Algoritmalara Giriş — MIT 6.006'dan

ML Builder için Türkçe Notlar

Phase 2

2026-06-07

İçindekiler

1 Önsöz 1

2 Bu kitap nedir? 3

3 Nasıl Okumalı 5

4 32 Ders 7

5 Notasyon 9

6 Builder Ekseni — Neden Bu Ders? 11

7 Yazım Kuralları 13

8 Algoritmalar ve Hesaplama 15

8.1 Bu Derste Ne Var? 15
8.2 Hesaplamalı Problem Nedir? 16
8.3 Problem = Girdi-Çıktı İlişkisi 16
8.4 Algoritma Nedir? 17
8.5 Doğum Günü Algoritması 18
8.6 Tümevarımla Doğruluk İspatı 20
8.7 Verimlilik: Zamanı Değil, İşlemi Say 22
8.8 Asimptotik Gösterim: O, Ω, Θ 22
8.9 Yaygın Çalışma Süresi Fonksiyonları 22
8.10 Hesaplama Modeli: word RAM 24
8.11 Bellek ve Word Boyutu 24
8.12 Temel İşlemler 25
8.13 Veri Yapıları: İlk Bakış 26
8.14 Bu Dersin Özeti 26
8.15 Kontrol Soruları 28
8.16 Egzersizler 28
8.17 Sonraki Ders İçin Hazırlık 29
8.18 Anahtar Kavramlar (Cheat Sheet) 29
8.19 Builder ve OMSCS Bağlantıları 30

9 Veri Yapıları ve Dinamik Diziler 31

9.1 Bu Derste Ne Var? 31
9.2 Arayüz mü, Veri Yapısı mı? 33
9.3 İki Temel Arayüz: Küme ve Dizi 33
9.4 Statik Dizi Arayüzü 33
9.5 Statik Dizi (Static Array) 34

9.6 Bellek Ayırma Modeli ve Word Boyutu . . . 34
9.7 Dinamik Dizi Arayüzü . . . 35
9.8 Bağlı Liste (Linked List) . . . 35
9.9 Veri Yapısı Zenginleştirme (Augmentation) . . . 36
9.10 Statik Dizi mi, Bağlı Liste mi? . . . 37
9.11 Dinamik Dizi (Dynamic Array) . . . 37
9.12 Dizi Büyütme ve Geometrik Seri . . . 39
9.13 Amortize Analiz (Amortized Analysis) . . . 39
9.14 Üç Veri Yapısı — Karşılaştırma . . . 42
9.15 Bu Dersin Özeti . . . 42
9.16 Kontrol Soruları . . . 44
9.17 Egzersizler . . . 44
9.18 Sonraki Ders İçin Hazırlık . . . 45
9.19 Anahtar Kavramlar (Cheat Sheet) . . . 45
9.20 Builder ve OMSCS Bağlantıları . . . 46

10 Problem Oturumu 1 **47**
10.1 Bu Problem Oturumu Ne Hakkında? . . . 47
10.2 Problem 1: Asimptotik Sıralama . . . 48
10.3 Problem 2: Sequence Arayüzünü Black Box Olarak Kullanma . . . 49
10.4 Problem 3: Çift Uçlu Dinamik Dizi . . . 50
10.5 Problem 4: Bağlı Listenin Son Yarısını Ters Çevirme . . . 51
10.6 Ne Öğrendik? . . . 53
10.7 Sonraki . . . 53

11 Kümeler ve Sıralama **55**
11.1 Bu Derste Ne Var? . . . 55
11.2 Arayüz Tekrarı: Küme Nedir? . . . 57
11.3 Küme (Set) Arayüzü . . . 57
11.4 Küme mi, Dizi mi? . . . 57
11.5 Sırasız Dizi ile Küme . . . 58
11.6 Sıralı Dizi ile Küme . . . 58
11.7 Sıralama Problemi ve Sözlük . . . 60
11.8 Permütasyon Sıralaması . . . 60
11.9 Seçmeli Sıralama (Selection Sort) . . . 61
11.10 Eklemeli Sıralama (Insertion Sort) . . . 63
11.11 Birleştirmeli Sıralama (Merge Sort) . . . 63
11.12 Sıralama Algoritmaları — Karşılaştırma . . . 66
11.13 Bu Dersin Özeti . . . 67
11.14 Kontrol Soruları . . . 67
11.15 Egzersizler . . . 68
11.16 Sonraki Ders İçin Hazırlık . . . 69
11.17 Anahtar Kavramlar (Cheat Sheet) . . . 69
11.18 Builder ve OMSCS Bağlantıları . . . 69

12 Hashing **71**
12.1 Bu Derste Ne Var? . . . 71
12.2 Set'i $O(\log n)$'den Hızlı Yapabilir miyiz? . . . 72

12.3 Karşılaştırma Modeli ve Alt Sınır . . . 72
12.4 Direct Access Array (Doğrudan Erişim Dizisi) . . . 74
12.5 Problem: Anahtar Evreni Çok Büyük . . . 74
12.6 Hash Fonksiyonu . . . 75
12.7 Çakışma (Collision) ve Çözümleri . . . 75
12.8 Hash Fonksiyonu Seçimi: Bölme Yöntemi . . . 75
12.9 Evrensel Hashing (Universal Hashing) . . . 77
12.10 Beklenen Zincir Uzunluğu . . . 79
12.11 Dinamik Hashing . . . 81
12.12 Bu Dersin Özeti . . . 81
12.13 Kontrol Soruları . . . 82
12.14 Egzersizler . . . 83
12.15 Sonraki Ders İçin Hazırlık . . . 83
12.16 Anahtar Kavramlar (Cheat Sheet) . . . 83
12.17 Builder ve OMSCS Bağlantıları . . . 84

13 Problem Oturumu 2 **87**
13.1 Bu Problem Oturumu Ne Hakkında? . . . 87
13.2 Problem 1: Yineleme Çözme — Master Theorem ve Özyineleme Ağacı . . . 87
13.3 Problem 2: Sonsuz Diziyi Arama . . . 89
13.4 Problem 3: Katmanlı Görüntü Editörü . . . 90
13.5 Problem 4: Tuğla Üfleme . . . 91
13.6 Ne Öğrendik? . . . 93
13.7 Sonraki . . . 93

14 Doğrusal Zamanlı Sıralama **95**
14.1 Bu Derste Ne Var? . . . 95
14.2 Önceki Ders: Hash Tablosunun Sınırı . . . 97
14.3 Sıralama Alt Sınırı: n log n . . . 97
14.4 Karşılaştırmanın Ötesi: Direct Access Array Sort . . . 99
14.5 Daha Büyük Aralık: Anahtarı Basamaklara Ayır . . . 99
14.6 Tuple Sort (Excel Tablo Sıralaması) . . . 100
14.7 Kararlı Sıralama (Stable Sort) . . . 100
14.8 Counting Sort . . . 103
14.9 Radix Sort . . . 103
14.10 Bu Dersin Özeti . . . 106
14.11 Kontrol Soruları . . . 106
14.12 Egzersizler . . . 108
14.13 Sonraki Ders İçin Hazırlık . . . 108
14.14 Anahtar Kavramlar (Cheat Sheet) . . . 109
14.15 Builder ve OMSCS Bağlantıları . . . 109

15 Problem Oturumu 3 **111**
15.1 Bu Problem Oturumu Ne Hakkında? . . . 111
15.2 Problem 1: Hash Tablosundan Dizi . . . 111
15.3 Problem 2: Critter Sort . . . 113
15.4 Problem 3: Küp Toplamı . . . 115
15.5 Problem 4: Poker — Kayan Pencere ve Frekans Tablosu . . . 116

15.6 Ne Öğrendik? . . . 118
15.7 Sonraki . . . 118

16 İkili Ağaçlar — Bölüm 1 **121**
16.1 Bu Derste Ne Var? . . . 121
16.2 Hedef: Tüm İşlemler O(log n) . . . 122
16.3 İkili Ağaç Nedir? . . . 122
16.4 Neden İki İşaretçi? . . . 124
16.5 Tanımlar: Alt Ağaç, Derinlik, Yükseklik . . . 124
16.6 Traversal (Geziş) Sırası . . . 125
16.7 subtree_first ve successor . . . 127
16.8 insert_after . . . 128
16.9 delete . . . 128
16.10 Sequence ve Set Olarak Ağaç . . . 130
16.11 Bu Dersin Özeti . . . 131
16.12 Kontrol Soruları . . . 131
16.13 Egzersizler . . . 132
16.14 Sonraki Ders İçin Hazırlık . . . 132
16.15 Anahtar Kavramlar (Cheat Sheet) . . . 133
16.16 Builder ve OMSCS Bağlantıları . . . 133

17 İkili Ağaçlar — Bölüm 2: AVL **135**
17.1 Bu Derste Ne Var? . . . 135
17.2 Geçen Dersten: O(h) İşlemler ve Bugünkü Hedef . . . 137
17.3 Küme Ağaçları = İkili Arama Ağaçları (BST) . . . 137
17.4 Dizi Ağaçları: subtree_at . . . 137
17.5 Alt Ağaç Zenginleştirme (Subtree Augmentation) . . . 138
17.6 Hangi Özellikler Tutulabilir? . . . 140
17.7 Ağaç Rotasyonu . . . 140
17.8 AVL / Yükseklik Dengesi . . . 142
17.9 Yükseklik Dengesi → Denge . . . 143
17.10 Yükseklik de Bir Alt Ağaç Özelliği . . . 143
17.11 Rotasyonlarla Dengeyi Koruma . . . 145
17.12 Bu Dersin Özeti . . . 145
17.13 Kontrol Soruları . . . 147
17.14 Egzersizler . . . 147
17.15 Sonraki Ders İçin Hazırlık . . . 148
17.16 Anahtar Kavramlar (Cheat Sheet) . . . 148
17.17 Builder ve OMSCS Bağlantıları . . . 149

18 Problem Oturumu 4 **151**
18.1 Bu Problem Oturumu Ne Hakkında? . . . 151
18.2 Problem 1: Sequence AVL — delete_at ve Çift Rotasyon . . . 151
18.3 Problem 2: En Güçlü Görüşler . . . 154
18.4 Problem 3: Müzayede — Çoklu Yapı ve Cross-Linking . . . 155
18.5 Problem 4: Receiver Roster — İç İçe Ağaçlar ve Rank Sorgusu . . . 158
18.6 Ne Öğrendik? . . . 158
18.7 Sonraki . . . 159

19 İkili Yığınlar (Binary Heaps) 161
19.1 Bu Derste Ne Var? . . . 161
19.2 1. Öncelik Kuyruğu Arayüzü . . . 162
19.3 2. Set AVL ile Çözüm . . . 162
19.4 3. Priority Queue Sort . . . 163
19.5 4. Üç Sıralama: Birleştirici Çerçeve . . . 163
19.6 5. Hedef: n log n + Yerinde (Complete Binary Tree) . . . 163
19.7 6. Complete Binary Tree ↔ Dizi . . . 165
19.8 7. Max-Heap Özelliği . . . 165
19.9 8. insert ve max_heapify_up . . . 167
19.10 9. delete_max ve max_heapify_down . . . 168
19.11 10. Yerinde Heapsort ve Doğrusal Build . . . 169
19.12 Bu Dersin Özeti . . . 169
19.13 Kontrol Soruları . . . 172
19.14 Egzersizler . . . 172
19.15 Sonraki Ders İçin Hazırlık . . . 173
19.16 Anahtar Kavramlar (Cheat Sheet) . . . 173
19.17 Builder ve OMSCS Bağlantıları . . . 174

20 Çizgeler ve Enine Arama (BFS) 175
20.1 Bu Derste Ne Var? . . . 175
20.2 1. Çizge Nedir? . . . 176
20.3 2. Çizgeler Her Yerde . . . 177
20.4 3. Basit Çizge ve |E| = O(V^2) . . . 178
20.5 4. Komşular ve Derece . . . 178
20.6 5. Çizge Veri Yapıları . . . 178
20.7 6. Yollar ve En Kısa Yol . . . 180
20.8 7. En Kısa Yol Ağacı . . . 181
20.9 8. Seviye Kümeleri ve BFS . . . 181
20.10 9. BFS Çalışma Süresi O(V + E) . . . 184
20.11 Bu Dersin Özeti . . . 184
20.12 Kontrol Soruları . . . 186
20.13 Egzersizler . . . 186
20.14 Sonraki Ders İçin Hazırlık . . . 187
20.15 Anahtar Kavramlar (Cheat Sheet) . . . 187
20.16 Builder ve OMSCS Bağlantıları . . . 188

21 Quiz 1 Gözden Geçirme 189
21.1 Bu Quiz Review Ne Hakkında? . . . 189
21.2 1. Büyük Dört ve İki Çözüm Yolu . . . 190
21.3 2. Üç Problem Tipi: White-box / Black-box / Modification . . . 190
21.4 3. Reduction Disiplini: Önce Arayüz, Sonra Verimlilik . . . 191
21.5 4. Sınav Stratejisi ve Kısmi Puan . . . 191
21.6 5. Kaçınılacak Tuzaklar (Downsides) . . . 193
21.7 6. Konu Tekrarı — Sıralama Algoritmaları . . . 193
21.8 7. Konu Tekrarı — Sequence Veri Yapıları . . . 194
21.9 8. Konu Tekrarı — Set Veri Yapıları ve Öncelik Kuyruğu . . . 194
21.10 Bu Quiz Review'in Özeti . . . 194

21.11 Quiz-tarzı Problemler . . . 195
21.12 Quiz Hazırlığı Egzersizleri . . . 198
21.13 Quiz 2 Öncesi Kapsam Genişlemesi . . . 199
21.14 Ders 1-12 Toplu Cheat Sheet . . . 199
21.15 Builder ve OMSCS Bağlantıları . . . 200

22 Derinlemesine Arama (DFS) 201
22.1 Bu Derste Ne Var? . . . 201
22.2 1. BFS'ten DFS'e: İki Arama Stratejisi . . . 202
22.3 2. Erişilebilirlik Problemi ve Ebeveyn Ağacı . . . 203
22.4 3. DFS Algoritması . . . 204
22.5 4. DFS Doğruluğu . . . 204
22.6 5. DFS Çalışma Süresi O(E) . . . 206
22.7 6. DFS En Kısa Yol Vermez . . . 206
22.8 7. Bağlılık ve Bağlı Bileşenler . . . 206
22.9 8. Yönlü Çevrimsiz Çizge (DAG) ve Topolojik Sıralama . . . 207
22.10 9. Bitiş Sırası → Topolojik Sıralama . . . 208
22.11 10. Çevrim Tespiti . . . 208
22.12 Bu Dersin Özeti . . . 211
22.13 Kontrol Soruları . . . 212
22.14 Egzersizler . . . 212
22.15 Sonraki Ders İçin Hazırlık . . . 213
22.16 Anahtar Kavramlar (Cheat Sheet) . . . 213
22.17 Builder ve OMSCS Bağlantıları . . . 214

23 Ağırlıklı En Kısa Yollar 215
23.1 Bu Derste Ne Var? . . . 215
23.2 1. Ağırlıksızdan Ağırlığa: Neden ve Nasıl . . . 216
23.3 2. Ağırlık Gösterimi . . . 217
23.4 3. Ağırlıklı Yol ve En Kısa Yol . . . 217
23.5 4. İki Tuzak: $+\infty$ ve $-\infty$ . . . 217
23.6 5. BFS'e İndirgenebilen Özel Durumlar . . . 219
23.7 6. Genel Manzara: DAG / Bellman-Ford / Dijkstra . . . 220
23.8 7. En Kısa Yol Ağacı: Mesafeden Ebeveyn . . . 220
23.9 8. DAG Relaxation: Mesafe Tahminleri ve Üçgen Eşitsizliği . . . 220
23.10 9. Relax Güvenlidir . . . 223
23.11 10. DAG Relaxation Algoritması ve Doğruluğu . . . 223
23.12 Bu Dersin Özeti . . . 225
23.13 Kontrol Soruları . . . 226
23.14 Egzersizler . . . 227
23.15 Sonraki Ders İçin Hazırlık . . . 227
23.16 Anahtar Kavramlar (Cheat Sheet) . . . 227
23.17 Builder ve OMSCS Bağlantıları . . . 228

24 Problem Oturumu 5 229
24.1 Bu Problem Oturumu Ne Hakkında? . . . 229
24.2 Problem 1: Çizge Yarıçapı ve Eksantriklik . . . 230
24.3 Problem 2: Router Gecikmesi ve Süpernode . . . 231

24.4 Problem 3: Potry Harter ve Büyülü Kapılar . . . 233
24.5 Problem 4: Purity Atlantic ve Sabitten Yararlanma . . . 234
24.6 Problem 5: Cep Küpü — Durum Çizgesi ve Ortada Buluşma . . . 235
24.7 Ne Öğrendik? . . . 236
24.8 Sonraki . . . 238

25 Bellman-Ford **239**
25.1 Bu Derste Ne Var? . . . 239
25.2 1. Bellman-Ford'un Hedefi . . . 240
25.3 2. Isınma: Yönsüz Çevrim ve İndirgeme . . . 241
25.4 3. En Kısa Yollar Basittir . . . 241
25.5 4. k-Kenar Mesafesi . . . 241
25.6 5. Tanık (Witness) ve $-\infty$ . . . 243
25.7 6. Her Negatif Çevrim Bir Tanık İçerir . . . 243
25.8 7. Graf Çoğaltma . . . 243
25.9 8. Graf Dönüşümü Örneği . . . 246
25.10 9. Bellman-Ford Algoritması . . . 246
25.11 10. Doğruluk ve Çalışma Süresi . . . 248
25.12 Bu Dersin Özeti . . . 248
25.13 Kontrol Soruları . . . 251
25.14 Egzersizler . . . 251
25.15 Sonraki Ders İçin Hazırlık . . . 252
25.16 Anahtar Kavramlar (Cheat Sheet) . . . 252
25.17 Builder ve OMSCS Bağlantıları . . . 253

26 Dijkstra **255**
26.1 Bu Derste Ne Var? . . . 255
26.2 1. Manzara: Üç Algoritma ve Dijkstra'nın Yeri . . . 256
26.3 2. Gözlem 1: Negatif Olmayan Ağırlık → Mesafe Artar . . . 257
26.4 3. Gözlem 2: Artan Sıra Bilinirse DAG Relaxation . . . 257
26.5 4. Dijkstra'nın Fikri . . . 257
26.6 5. Değiştirilebilir Öncelik Kuyruğu . . . 259
26.7 6. Dijkstra Algoritması . . . 259
26.8 7. Doğruluk: İki Gözlem . . . 262
26.9 8. Çalışma Süresi: Öncelik Kuyruğu Seçimi . . . 262
26.10 Bu Dersin Özeti . . . 265
26.11 Kontrol Soruları . . . 266
26.12 Egzersizler . . . 267
26.13 Sonraki Ders İçin Hazırlık . . . 267
26.14 Anahtar Kavramlar (Cheat Sheet) . . . 267
26.15 Builder ve OMSCS Bağlantıları . . . 268

27 Problem Oturumu 6 **269**
27.1 Bu Problem Oturumu Ne Hakkında? . . . 269
27.2 Problem 1: Dijkstra Elle ve Negatif Kenarın Kırdığı Varsayım . . . 270
27.3 Problem 2: Ağırlıklı Yarıçap ve Johnson . . . 271
27.4 Problem 3: Atniss ve Sensörler — Süpernode ve İkili Arama . . . 273
27.5 Problem 4: Ashley ve Critter'lar — Graf Çoğaltma ve Durum Makinesi . . . 274

27.6 Problem 5: Nakliye ve Darboğaz — Modifiye Dijkstra . . . 275
27.7 Ne Öğrendik? . . . 278
27.8 Sonraki . . . 278

28 Tüm-Çiftler En Kısa Yollar (Johnson) 279
28.1 Bu Derste Ne Var? . . . 279
28.2 1. APSP Problemi ve $\Theta(V^2)$ Çıktısı . . . 280
28.3 2. Naif Çözüm: V × SSSP . . . 281
28.4 3. Fikir: Yeniden Ağırlıklandırma . . . 282
28.5 4. Kötü Fikir: Her Kenara Sabit Ekle . . . 282
28.6 5. İyi Fikir: Potansiyel Dönüşümü . . . 282
28.7 6. Potansiyel Fonksiyon ve Telescoping . . . 284
28.8 7. Negatif-Olmama Koşulu = Üçgen Eşitsizliği . . . 284
28.9 8. Süpernode ile Potansiyeli Hesapla . . . 284
28.10 9. Johnson Algoritması ve Çalışma Süresi . . . 287
28.11 Bu Dersin Özeti . . . 288
28.12 Kontrol Soruları . . . 290
28.13 Egzersizler . . . 290
28.14 Sonraki Ders İçin Hazırlık . . . 291
28.15 Anahtar Kavramlar (Cheat Sheet) . . . 291
28.16 Builder ve OMSCS Bağlantıları . . . 292

29 Quiz 2 Gözden Geçirme 293
29.1 Bu Quiz Review Ne Hakkında? . . . 293
29.2 1. Quiz 2 Neyi Ölçer — Modelleme ve İndirgeme . . . 294
29.3 2. Çizge Algoritmaları Haritası . . . 295
29.4 3. SSSP Hiyerarşisi: BFS → DAG → Dijkstra → Bellman-Ford . . . 295
29.5 4. APSP ve Johnson . . . 295
29.6 5. Graf Değiştirme Stratejileri . . . 296
29.7 6. Sınav Taktiği ve Puan Kaybı . . . 296
29.8 Bu Quiz Review'in Özeti . . . 297
29.9 Quiz-tarzı Problemler . . . 297
29.10 Quiz Hazırlığı Egzersizleri . . . 302
29.11 Quiz 3 Öncesi Kapsam Genişlemesi . . . 302
29.12 Ders 13-21 Toplu Cheat Sheet (L9-L14 + PS5-6) . . . 302
29.13 Builder ve OMSCS Bağlantıları . . . 303

30 Dinamik Programlama 1: SRTBOT 305
30.1 Bu Derste Ne Var? . . . 305
30.2 1. Yeni Bölüm: Algoritmik Tasarım ve DP . . . 306
30.3 2. SRTBOT Çerçevesi . . . 307
30.4 3. Örnek: Merge Sort SRTBOT ile . . . 308
30.5 4. Fibonacci: Memoization'sız Üstel . . . 308
30.6 5. Memoization: Üstelden Polinoma . . . 308
30.7 6. Çalışma Süresi Formülü . . . 311
30.8 7. DAG En Kısa Yol DP Olarak . . . 311
30.9 8. Alt-Problem Tasarım Aracı: Prefix/Suffix/Substring . . . 313
30.10 9. Bowling Problemi: SRTBOT Uygulaması . . . 313

30.11 10. Bottom-Up DP ve Yerel Kaba Kuvvet . . . 313
30.12 Bu Dersin Özeti . . . 316
30.13 Kontrol Soruları . . . 316
30.14 Egzersizler . . . 317
30.15 Sonraki Ders İçin Hazırlık . . . 317
30.16 Anahtar Kavramlar (Cheat Sheet) . . . 318
30.17 Builder ve OMSCS Bağlantıları . . . 318

31 Dinamik Programlama 2: LCS, LIS, Oyunlar 321
31.1 Bu Derste Ne Var? . . . 321
31.2 1. DP 2/4: Üç Örnek ve Yeni Fikirler . . . 322
31.3 2. SRTBOT Hatırlatma . . . 323
31.4 3. LCS: Çoklu Girdi → Alt Problem Çarpımı . . . 323
31.5 4. LCS Recurrence: Eşit / Farklı Durumlar . . . 323
31.6 5. Parent Pointers ile Çözüm Kurtarma . . . 326
31.7 6. LIS: Naif Tanım Neden Çöker . . . 326
31.8 7. LIS: Alt Problem Kısıtı . . . 327
31.9 8. Değişen Para Oyunu: Substring + Genişletme . . . 327
31.10 9. İki Oyuncu: Max/Min Recurrence . . . 327
31.11 10. Subproblem Expansion İlkesi . . . 329
31.12 Bu Dersin Özeti . . . 330
31.13 Kontrol Soruları . . . 331
31.14 Egzersizler . . . 332
31.15 Sonraki Ders İçin Hazırlık . . . 332
31.16 Anahtar Kavramlar (Cheat Sheet) . . . 332
31.17 Builder ve OMSCS Bağlantıları . . . 333

32 Problem Oturumu 8 335
32.1 Bu Problem Oturumu Ne Hakkında? . . . 335
32.2 Problem 1: Tim the Beaver — Mutluluk DP . . . 336
32.3 Problem 2: Menix Edit Distance — Precomputation . . . 337
32.4 Problem 3: Saggy'nin Blok Kulesi — LIS-benzeri . . . 339
32.5 Problem 4: Princess Plum Izgarası — Yol Sayma . . . 341
32.6 Ne Öğrendik? . . . 343
32.7 Sonraki . . . 343

33 Dinamik Programlama 3: Floyd-Warshall, Parantezleme 345
33.1 Bu Derste Ne Var? . . . 345
33.2 1. DP 3/4: Subproblem Expansion . . . 347
33.3 2. Bellman-Ford DP Olarak . . . 347
33.4 3. Floyd-Warshall: Vertex-Prefix APSP . . . 348
33.5 4. Floyd-Warshall Recurrence ve $O(V^3)$ . . . 349
33.6 5. Floyd-Warshall vs Johnson . . . 349
33.7 6. Aritmetik Parantezleme: Kökü Tahmin Et . . . 350
33.8 7. Negatif Sayılar: Min/Max Genişletmesi . . . 352
33.9 8. Parantezleme Recurrence ve $O(n^3)$ . . . 352
33.10 9. Piyano Parmaklama: State = Parmak . . . 352
33.11 10. Genelleme: Çoklu Nota, Gitar . . . 354

33.12 Bu Dersin Özeti . . . 355
33.13 Kontrol Soruları . . . 355
33.14 Egzersizler . . . 356
33.15 Sonraki Ders İçin Hazırlık . . . 356
33.16 Anahtar Kavramlar (Cheat Sheet) . . . 357
33.17 Builder ve OMSCS Bağlantıları . . . 357

34 Dinamik Programlama 4: Pseudopolinom, Subset Sum 359
34.1 Bu Derste Ne Var? . . . 359
34.2 1. DP 4/4: Tamsayı Alt Problemler . . . 361
34.3 2. Rod Cutting: Tamsayı Alt Problem . . . 361
34.4 3. Rod Cutting Polinom mu? . . . 363
34.5 4. Subset Sum: Karar Problemi . . . 363
34.6 5. Subset Sum Recurrence: OR . . . 363
34.7 6. Subset Sum Polinom Değil: Pseudopolinom . . . 365
34.8 7. Pseudopolinom Tanımı ve Hiyerarşi . . . 367
34.9 8. DP Karakterizasyonu: Alt Problem Tipleri . . . 367
34.10 9. DP Karakterizasyonu: Constraint / Branching / Combination . . . 367
34.11 10. Dört Dersin Özeti . . . 370
34.12 Bu Dersin Özeti . . . 370
34.13 Kontrol Soruları . . . 370
34.14 Egzersizler . . . 371
34.15 Sonraki Ders İçin Hazırlık . . . 372
34.16 Anahtar Kavramlar (Cheat Sheet) . . . 372
34.17 Builder ve OMSCS Bağlantıları . . . 373

35 Hesaplama Karmaşıklığı: P, NP, NP-Tamlık 375
35.1 Bu Derste Ne Var? . . . 375
35.2 1. Karmaşıklık: Alt Sınır Tarafı . . . 377
35.3 2. P, EXP, R Hiyerarşisi . . . 377
35.4 3. Halting Problem: Uncomputable . . . 377
35.5 4. Çoğu Problem Çözülemez . . . 379
35.6 5. NP: Tanım 1 — Şanslı Algoritma . . . 380
35.7 6. NP: Tanım 2 — Doğrulayıcı . . . 380
35.8 7. P ≠ NP Konjektürü . . . 380
35.9 8. NP-hard ve NP-complete . . . 382
35.10 9. Reduction: İndirgeme ile Zorluk Kanıtı . . . 382
35.11 10. NP-complete Örnekleri . . . 382
35.12 Bu Dersin Özeti . . . 385
35.13 Kontrol Soruları . . . 385
35.14 Egzersizler . . . 387
35.15 Sonraki Ders İçin Hazırlık . . . 388
35.16 Anahtar Kavramlar (Cheat Sheet) . . . 388
35.17 Builder ve OMSCS Bağlantıları . . . 389

36 Problem Oturumu 9 391
36.1 Bu Problem Oturumu Ne Hakkında? . . . 391
36.2 Problem 1: Coin-Crafting — 0/1 Knapsack . . . 393

36.3 Problem 2: Tim the Beaver Kariyer Fuarı — Sınırsız Knapsack + Çanta Boşaltma 394
36.4 Problem 3: Protein Parsing — Precomputation ile Hızlandırma 395
36.5 Problem 4: Lazy Egg Drop — Minimax DP 396
36.6 Ne Öğrendik? 400
36.7 Sonraki 400

37 Quiz 3 Gözden Geçirme **401**
37.1 Bu Quiz Review Ne Hakkında? 401
37.2 SRTBOT Çerçevesi — Derin Tekrar 403
37.3 Quiz-tarzı Problemler (Spring ’18, Tam Çözüm) 404
37.4 Quiz Hazırlığı Egzersizleri 408
37.5 Sınav Stratejisi (Kapsam Notları) 408
37.6 Toplu Cheat Sheet — SRTBOT 409
37.7 Bu Quiz Review’in Özeti 409
37.8 Builder ve OMSCS Bağlantıları 410

38 Toparlanma ve Sonraki Dersler **411**
38.1 Bu Derste Ne Var? 411
38.2 1. 6.006’nın 4 Hedefi 411
38.3 2. Karmaşıklık — Final İçin (L19 Özeti) 413
38.4 3. Üç Ünite — Quiz 1: Veri Yapıları (Kara Kutular) 414
38.5 4. Üç Ünite — Quiz 2: Çizgeler 414
38.6 5. Üç Ünite — Quiz 3: DP = Uygulamalı Çizge 416
38.7 6. 6.046 — Doğal Uzantı 417
38.8 7. 6.046 — “Doğru/Verimli” Tanımını Gevşetmek 417
38.9 Bu Dersin Özeti 419
38.10 Kontrol Soruları 420
38.11 Egzersizler 421
38.12 Sonraki Ders İçin Hazırlık 421
38.13 Anahtar Kavramlar (Cheat Sheet) 421
38.14 Builder ve OMSCS Bağlantıları 422

39 Son Ders: Algoritmalar Her Yerde **425**
39.1 Bu Derste Ne Var? 425
39.2 1. Üç Geometrici + Jason’ın Origami Yolculuğu 427
39.3 2. Demaine — Computational Origami (6.849) 427
39.4 3. Demaine — Self-Assembly (Geometrik Hesaplama Modeli) 427
39.5 4. Demaine — İleri Veri Yapıları, Planar Çizgeler, Recreational 430
39.6 5. Solomon — Mesh Üzerinde En Kısa Yol (Dijkstra Neden Yanlış) 430
39.7 6. Solomon — Ray Casting & Stanford Bunny 430
39.8 7. Solomon — Politik Redistricting (Gerrymandering) 432
39.9 Bu Dersin Özeti 435
39.10 Kontrol Soruları 435
39.11 Egzersizler 436
39.12 6.006 Tamamlandı — Kurs Kapanışı 436
39.13 Anahtar Kavramlar (Cheat Sheet) 438
39.14 Builder ve OMSCS Bağlantıları 438

1 Önsöz

2 Bu kitap nedir?

Bu, **MIT 6.006 — Introduction to Algorithms** (Spring 2020) ders serisinin Türkçe ders notlarıdır. Hedef, videoları izlerken paralel okunabilecek; sonradan tek başına da yeterli olabilecek bir referans seti üretmek.

Kurs üç hoca tarafından, üç farklı sesle anlatılır:

- **Jason Ku** — akademik, tahta + marker, Sokratik; "önce soruyu sınıfa taşı".
- **Erik Demaine** — enerjik, görselleştiren; *"I love algorithms."*
- **Justin Solomon** — informal, problem oturumları ve çizge teorisi.

Serinin bir iddiası var: bir algoritma yazmak işin yalnızca **yarısıdır**. Diğer yarısı, o algoritmanın *doğru* ve *verimli* olduğunu — bir bilgisayara değil, bir **insana** ikna edecek netlikte — kanıtlamaktır. 6.006 boyunca her ders bu iki yarıyı birlikte öğretir.

> **i** Kaynak
>
> - **Seri:** MIT 6.006 Introduction to Algorithms, Spring 2020 (OCW)
> - **YouTube playlist:** 6.006 Spring 2020 (32 video)
> - **Hocalar:** Jason Ku, Erik Demaine, Justin Solomon
> - **Çeviri ve genişletme:** Phase 2 (TR + ML Builder / OMSCS köprüleri)

3 Nasıl Okumalı

Sıralı oku. Kurs kümülatiftir — her ünite bir öncekinin kurduğu kavramları kullanır. Üç büyük ünite vardır: **veri yapıları** (diziler, hashing, ağaçlar, yığınlar), **çizge algoritmaları** (BFS, DFS, en kısa yollar), ve **dinamik programlama**. Aralara serpiştirilmiş **problem oturumları** (PS) her bloğu pekiştirir; üç **quiz tekrarı** sınav öncesi sentez yapar.

Önerilen akış: önce videoyu izle, sonra ilgili dersi oku, en sonunda **bir problemi kendin çöz**. Bu set videoyu **destekler**, ikame etmez.

Pratik bir tavsiye

6.006'nın kalbi koddan çok **ispat ve iletişimdir**. Her dersteki tümevarım argümanını, asimptotik analizi ve "neden doğru" gerekçesini atlama. Bir algoritmayı tarif seviyesinde — sınıftaki bir arkadaşına anlatabileceğin netlikte — yazabiliyorsan, onu anlamışsındır.

4 32 Ders

Kurs 32 videodan oluşur: **21 ders (lecture) + 3 quiz tekrarı + 8 problem oturumu**. Aşağıdaki sıra videoların orijinal akışıdır.

#	Tip	Ders
1	L1	Algoritmalar ve Hesaplama
2	L2	Veri Yapıları ve Dinamik Diziler
3	PS1	Problem Oturumu 1
4	L3	Kümeler ve Sıralama
5	L4	Hashing
6	PS2	Problem Oturumu 2
7	L5	Doğrusal Zamanlı Sıralama
8	PS3	Problem Oturumu 3
9	L6	İkili Ağaçlar — Bölüm 1
10	L7	İkili Ağaçlar — Bölüm 2: AVL
11	PS4	Problem Oturumu 4
12	L8	İkili Yığınlar (Binary Heaps)
13	L9	Çizgeler ve Enine Arama (BFS)
14	Quiz 1 Review	Quiz 1 Gözden Geçirme
15	L10	Derinlemesine Arama (DFS)
16	L11	Ağırlıklı En Kısa Yollar
17	PS5	Problem Oturumu 5
18	L12	Bellman-Ford
19	L13	Dijkstra
20	PS6	Problem Oturumu 6
21	L14	Tüm-Çiftler En Kısa Yollar (Johnson)
22	Quiz 2 Review	Quiz 2 Gözden Geçirme
23	L15	Dinamik Programlama 1: SRTBOT
24	L16	Dinamik Programlama 2: LCS, LIS, Oyunlar
25	PS8	Problem Oturumu 8
26	L17	Dinamik Programlama 3: Floyd-Warshall, Parantezleme
27	L18	Dinamik Programlama 4: Pseudopolinom, Subset Sum
28	L19	Hesaplama Karmaşıklığı: P, NP, NP-Tamlık
29	PS9	Problem Oturumu 9
30	PS10	Quiz 3 Gözden Geçirme
31	L20	Toparlanma ve Sonraki Dersler
32	L21	Son Ders — Algoritmalar Her Yerde

Not: Phase 2 üretimi Ders 1 (Algoritmalar ve Hesaplama) ile başlar. Kalan dersler aynı şablonla eklenir.

5 Notasyon

- **Asimptotik sınırlar:** $O(\cdot)$ üst sınır, $\Omega(\cdot)$ alt sınır, $\Theta(\cdot)$ sıkı sınır.
- **Büyüme hiyerarşisi:** $1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^c \ll 2^n$ — soldan sağa hızlı büyür; polinom ve altı "verimli".
- **Girdi boyutu (n):** her zaman "eleman sayısı" değildir; bir çizge için genellikle $V + E$ (düğüm + kenar).
- **Çizge:** $G = (V, E)$ — düğüm kümesi V, kenar kümesi E.
- **Mesafe:** $\delta(s, v)$ — s'ten v'ye en kısa yol uzunluğu.

Tüm matematik MathJax 3 ile render ediliyor.

6 Builder Ekseni — Neden Bu Ders?

Her ders bu bağlantı katmanını taşır

Bu kursun pratik değeri ML değil, **mühendislik temeli ve ölçek**:

6.006 kavramı	İleriye / Geriye köprü
O / Θ / Ω, DP, çizge dili	**OMSCS CS 6515 (Graduate Algorithms)** ile birebir örtüşür
"Büyük N'de hangi karmaşıklık geçer"	Kompetitif programlama sezgisi
Hash tablosu, en kısa yol, öncelik kuyruğu	Sistem tasarımı: routing, cache, scheduler
`list`/`dict`'in gizli zaman maliyeti	**Geriye → Phase 1 Python** (`time.perf_counter` ile ampirik doğrulama)
Asimptotik mertebeler (log, exp)	**Geriye → Phase 1 Calculus** (büyüme oranları)
Randomized algoritma, beklenen-zaman	**Geriye → Phase 1 Stat 110** (hashing, quicksort)

7 Yazım Kuralları

- **Türkçe terminoloji + parantez içinde İngilizce orijinal** ilk geçtiğinde: "ikili ilişki (binary relation)", "tümevarım (induction)", "hesaplama modeli (word RAM)".
- **Hocalardan alıntılar** İngilizce orijinal hâliyle, blockquote içinde, zaman damgasıyla verilir.
- **Builder Notu** callout'ları ML köprüsünü (geriye + ileriye) ve OMSCS bağını taşır.
- **Öğretim kodu** (pseudocode, Python örnekleri) görünür kod bloklarında; **figürler** (çizge, büyüme eğrisi, bellek şeması) algoritmayı görselleştiren, kaynağı gizli üretici hücrelerdir.
- **Kontrol Soruları** collapse'lu — cevap kapalı başlar, okur kendi düşündükten sonra açar.
- **Egzersizler** cevapsız — en az bir kodlama/ispat egzersizi.

Bu kitap videoların yerine geçmez

Tek başına bu set yetmez — Ku, Demaine ve Solomon'un canlı anlatımının yerine geçemez. Önce videoyu izle, sonra ilgili dersi oku, son olarak bir problemi **kendin çöz**. Set videoyu **destekler**, ikame etmez.

8 Algoritmalar ve Hesaplama

Problem vs algoritma, tümevarımla doğruluk, asimptotik verimlilik ve word RAM modeli

ℹ Bölüm bilgisi

- **Ku’nun videosu:** YouTube — Lecture 1: Algorithms and Computation (≈46 dk)
- **OCW sayfası:** MIT 6.006 Lecture 1: Algorithms and Computation
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 1
- **Hocalar:** Jason Ku, Erik Demaine, Justin Solomon
- **Okuma süresi:** ≈22 dk

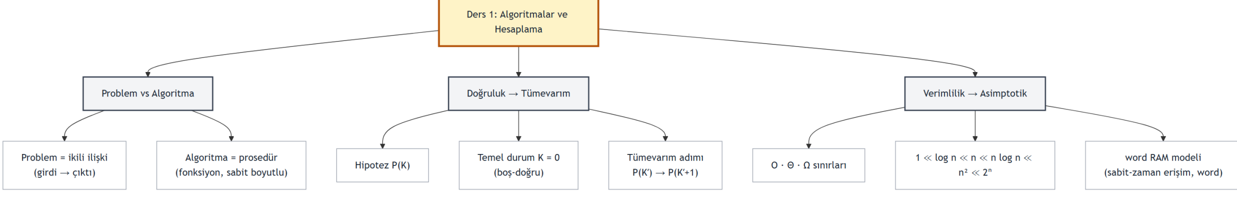
8.1 Bu Derste Ne Var?

6.006’nın ilk dersi, kursun *neyi* öğrettiğini tanımlar: hesaplamalı problemleri çözmeyi, çözümün **doğru** olduğunu ispatlamayı ve **verimli** olduğunu savunmayı. Jason Ku, kodlamadan çok düşünmeye ve iletişime odaklandığımızı vurgular — bu derste yazacağınız ispat, çoğu zaman yazacağınız koddan fazladır.

Üç temel kavram bu derste yan yana gelir:

1. **Problem vs Algoritma** — bir problem girdi-çıktı ilişkisidir; bir algoritma o ilişkiyi hesaplayan bir prosedürdür.
2. **Tümevarımla doğruluk** — bir algoritmanın *her* girdi için doğru çalıştığını sonlu bir argümanla ispatlama.
3. **Asimptotik verimlilik + hesaplama modeli** — zamanı saniyeyle değil, temel işlem sayısıyla ölçme; word RAM modeli.

> “*Really what the course is about is teaching you to solve computational problems.*” — Ku, 1:00



Şekil 8.1: Ders 1’in kavram haritası: problem vs algoritma ikili ilişkisinden, tümevarımla doğruluğa, oradan asimptotik verimlilik + word RAM hesaplama modeline.

Builder Notu — ML ve OMSCS Köprüleri

Bu ders ML değil, **mühendislik temeli** kurar. Phase 1 ve ileri hedeflerle köprüler:

- **Geriye → Python (Phase 1):** Ku, Python'ın `list`, `set`, `dict` yapılarının "bu modelde olmadığını" söyler (42:28). Python'da bedava görünen işlemlerin gerçek bir zaman maliyeti var — bu kurs o maliyeti görünür kılar.
- **Geriye → Calculus (Phase 1):** asimptotik mertebeler (log, lineer, polinom, üstel büyüme) — fonksiyonların büyüme hızını karşılaştırma.
- **İleriye → OMSCS CS 6515 (Graduate Algorithms):** bu dersin formal devamı; aynı dil ($O/\Theta/\Omega$, DP, çizge).
- **İleriye → kompetitif programlama ve sistem tasarımı:** "büyük N'de hangi karmaşıklık geçer" sezgisi; routing, cache, scheduler tasarımı.

Tek cümle: *Bu ders, "hızlı kod" ile "hızlı algoritma" arasındaki farkı ve bunu nasıl ispatlayacağını öğretir.*

8.2 Hesaplamalı Problem Nedir?

Ku derse bir soruyla başlar: *"What is a problem? What is an algorithm?"* — ve cevabı söylemek yerine sınıfı oraya taşır (Sokratik üslup). Kursun dört hedefi vardır:

1. Hesaplamalı problemi **çöz**.
2. Çözümün **doğru** olduğunu ispatla.
3. Çözümün **verimli** olduğunu savun.
4. Bunların hepsini başkalarına **iletişimle** aktar.

Ku'ya göre bu sınıfı diğer kodlama derslerinden ayıran şey budur: sadece bilgisayara değil, *insanlara* doğruluğu kanıtlamak. Bu yüzden derste bol bol yazı yazılır.

> *"we really concentrate on being able to prove that the things you're doing are correct and better than other things, and being able to communicate those ideas to others, and not just to a computer"*
> — Ku, 2:40

8.3 Problem = Girdi-Çıktı İlişkisi

Soyut tanım: bir **problem**, girdiler kümesi ile çıktılar kümesi arasında bir **ikili ilişkidir** (binary relation). Her girdi için hangi çıktıların *doğru* olduğunu belirtir — bu, girdiler ile çıktılar arasında iki-parçalı (bipartite) bir çizge gibidir (Şekil 8.2). Bir girdinin birden çok doğru çıktısı olabilir: örneğin "dizide 5 değerini içeren bir indeks ver" probleminde birden çok 5 varsa, o indekslerin herhangi biri doğrudur.

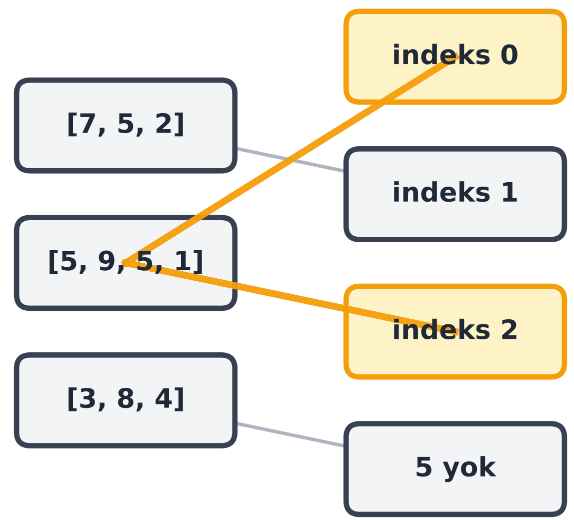
Pratikte bir problemi tek tek girdi-çıktı çiftleriyle değil, bir **yüklem (predicate)** ile tanımlarız: bir girdi-çıktı çifti verildiğinde, çıktının doğru olup olmadığını *kontrol edebiliriz*. Bu, sonsuz sayıda girdiyi sonlu bir kuralla ifade etmenin yoludur.

> *"what a problem is is a binary relation between these inputs and outputs."* — Ku, 3:48

Çalışılan Örnek — Doğum Günü Problemi (kurulum): Bir sınıftaki öğrenciler arasında, *herhangi iki* öğrencinin aynı doğum anına (gün + yıl + saat) sahip olup olmadığını soralım. Ku, algoritmanın yalnızca bu sınıf için değil, **keyfi boyutta** herhangi bir girdi için çalışmasını ister — 300 öğrenci de olabilir, bir milyar öğrenci de. İşte bu yüzden "sabit boyutlu" bir makine ile "keyfi boyutlu" girdiyi işlemek zorundayız.

Problem = ikili ilişki: bir girdinin birden çok doğru çıktısı olabilir

Girdiler (dizi) Çıktı(lar): 5'in indeksi



Amber: tek girdi → iki geçerli çıktı (5 dizide iki kez geçiyor)

Şekil 8.2: Problem bir **ikili ilişkidir**: sol sütun girdiler (örnek diziler), sağ sütun olası çıktılar (5'in indeksi). Kenarlar geçerli (girdi → çıktı) çiftleridir. `[7, 5, 2]` için tek doğru çıktı (indeks 1), `[3, 8, 4]` için "5 yok"; ama `[5, 9, 5, 1]` için **iki** geçerli çıktı vardır (indeks 0 ve indeks 2, amber vurgulu) — bir girdinin birden çok doğru cevabı olabilir, yani problem fonksiyon olmak zorunda değildir.

8.4 Algoritma Nedir?

Bir **algoritma**, problemin tersine, çıktıları önceden bilmez — o bir **prosedürdür**: sabit boyutlu bir makine ya da tarif ki, kendisine bir girdi verildiğinde bir çıktı *üretir*. Matematiksel olarak algoritma bir **fonksiyondur**: her girdiyi tek bir çıktıya eşler ve o çıktı, problemin tanımına göre doğru olmalıdır.

Kritik gerilim şudur: girdi keyfi büyüklükte olabilir (bir milyar öğrenci), ama algoritmamız **sabit boyutludur**. Sabit boyutlu bir kod, keyfi büyük girdiyi işleyebilmek için **döngü kurmalı veya özyineleme (recursion) yapmalıdır** — aynı kod satırlarını tekrar tekrar çalıştırarak. Bu gözlem, neden bir sonraki adımda tümevarıma ihtiyaç duyacağımızın da habercisidir.

> *"An algorithm is some kind of function that takes these inputs, maps it to a single output, and that output better be correct based on our problem."* — Ku, 10:19

> *"So it's just a procedure. You can think of it as like a recipe."* — Ku, 14:56

8.5 Doğum Günü Algoritması

Ku, bir öğrenciden algoritma önerisi alır ve onu formalize eder. Algoritma dört adımdan oluşur (somut bir izi için Şekil 8.3):

1. Bir **kayıt (record)** tut.
2. Öğrencileri bir sırayla **görüşmeye al** (interview).
3. Her görüşmede: doğum günü kayıtta var mı **kontrol et** → varsa bir **çift döndür**; yoksa öğrenciyi kayda **ekle**.
4. Herkes bittiğinde eşleşme bulunamadıysa "eşleşme yok" döndür.

> *"Maintain a record. Interview students in some order… check if birthday in record… return pair… Otherwise, add a new student to record."* — Ku, 12:11

Bu, problem setlerinde beklenen **tarif seviyesinde** açıklamadır: bir bilgisayara değil, sınıftaki bir arkadaşına anlatabileceğin netlikte sözel bir tanım.

> *"if you said this algorithm to any of your friends in this class… they would at least understand what it is that you're doing."* — Ku, 13:46

Şimdi elimizde bir algoritma var. Ama bu yeterli değil: Ku'nun dört hedefinden ikisi henüz eksik — bu algoritmanın **doğru** ve **verimli** olduğunu göstermek. Sonraki iki bölüm tam olarak bunu yapar.

Builder Notu — Çarpışma Tespiti ve Hash Fonksiyonları

Doğum günü algoritmasının "kayıt"ı aslında bir **hash kümesidir** (Python `dict`/`set`). "Doğum günü kayıtta var mı?" kontrolünün $O(1)$ olması, hash fonksiyonunun anahtarı sabit zamanda bir kovaya eşlemesinden gelir — bu, Ders 4'ün (Hashing) konusu.

- **Geriye → çarpışma matematiği:** Doğum günü *paradoksu* (sürpriz biçimde az sayıda öğrencide eşleşme olasılığının yükselmesi), hash tablolarındaki çarpışma analizinin (yük faktörü, beklenen çarpışma sayısı) ta kendisidir. Phase 1 Stat 110 olasılık dersiyle doğrudan köprü.
- **İleriye → feature hashing (ML):** Yüksek-kardinaliteli kategorik öznitelikleri sabit boyutlu bir vektöre indirgeyen "hashing trick", çarpışmaları *bilerek tolere eder* — kabul edilebilir çarpışma oranını yine bu doğum günü matematiği sınırlar.
- **İleriye → embedding lookup:** Bir token → indeks → embedding satırı erişimi, bu kayıt sözlüğünün ölçeklenmiş hâlidir: anahtar→değer $O(1)$ erişim. Karpathy'nin makemore serisinde embedding tablosu tam olarak budur.

Şekil 8.3: Doğum günü algoritmasının adım-adım izi. Öğrenciler sırayla görüşülür; her satır bir görüşme adımıdır ve o ana kadar **record** sözlüğünde biriken doğum günlerini kutucuklarla gösterir (kutuda gün + ilk gören öğrenci). Eşleşme bulunana kadar her adımda kayıt bir hücre **büyür** (slate kutular, kesik çizgili '+ kayıt' = yeni eklenen). Adım 5'te Can'ın doğum günü (03-14) daha önce Ali'de görüldüğü için **çarpışma** olur: çakışan önceki kayıt amber çerçeveyle, aktif sorgu dolu amber kutuyla vurgulanır ('EŞLEŞME!') ve algoritma orada **durur** — bu yüzden iz yalnızca 5 adım. record sabit-zamanlı sözlük araması sayesinde toplam maliyet $O(n)$.

8.6 Tümevarımla Doğruluk İspatı

Dört öğrenci için algoritmanın doğruluğunu elle deneyerek gösterebilirdik. Ama 300 (ya da bir milyar) öğrenci için tek tek denemek imkânsız. Sabit boyutlu bir kodun *keyfi büyük* bir girdide doğru çalıştığını göstermenin yolu **tümevarımdır (induction)** — bu yüzden 6.006 öncesi bir ispat/ayrık matematik dersi şarttır.

> *"Induction, right? … we write a constant sized piece of code that can take on any arbitrarily large size input."* — Ku, 16:23

Çalışılan Örnek — Doğum Günü Algoritmasının Doğruluğu

Tümevarımı kurmak için üç parça gerekir: hipotez, temel durum, tümevarım adımı. Bu üç parça Şekil 8.4 içinde bir arada görülür.

1) Tümevarım hipotezi (P(K)): İlk K öğrenci bir eşleşme içeriyorsa, algoritma K+1'inci öğrenciyi görüşmeden *önce* bir çift döndürmüştür.

> *"If first K students contain a match, algorithm returns a match before interviewing student K plus 1."* — Ku, 19:29

2) Temel durum (K = 0): Hiç öğrenci görüşülmeden önce hiçbir iş yapılmamıştır. İlk 0 öğrenci bir eşleşme *içeremez*, dolayısıyla hipotez boş-doğru (vacuously true) olarak sağlanır. Ku, en kolay temel durumun 2 veya 1 değil, 0 olduğunu vurgular.

> *"After interviewing 0 students, I haven't done any work… the first 0 can't have a match."* — Ku, 21:21

3) Tümevarım adımı: P(K) doğru varsayılır, P(K +1) gösterilir. İki durum vardır:

- **Durum A:** İlk K öğrenci zaten bir eşleşme içeriyorsa → tümevarım hipotezi gereği algoritma çoktan doğru çıktıyı döndürmüştür.
- **Durum B:** İlk K öğrenci eşleşme içermiyorsa ve K +1'inci öğrenci görüşülür → eğer ilk K +1 öğrenci içinde bir eşleşme varsa, bu eşleşme *zorunlu olarak* K +1'inci öğrenciyi içerir (aksi halde daha önce eşleşme olurdu). Algoritma yeni öğrencinin doğum gününü kayıtta arar; varsa çifti döndürür, yoksa öğrenciyi kayda ekler ve hipotez K +1 için yeniden kurulur.

Sonuç: K = n alındığında, n öğrencinin tümü görüşüldükten sonra bir eşleşme varsa döndürülmüş olur; yoksa algoritma "eşleşme yok" döndürür. İkisi de doğrudur.

> *"OK, so that's how we prove correctness."* — Ku, 24:52

Ku, bu seviyede bir formalliğin her zaman gerekmediğini, ama kalibrasyon için yeterli olduğunu belirtir:

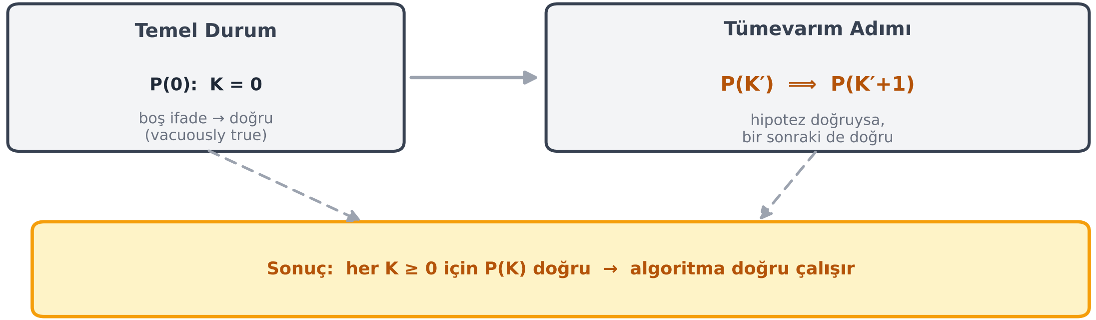
> *"This is a little bit more formal than we would ask you to do… but it's definitely sufficient."* — Ku, 24:57

Hedef çıta: bir başkası senin algoritmanı okuyup kodlayabilmeli.

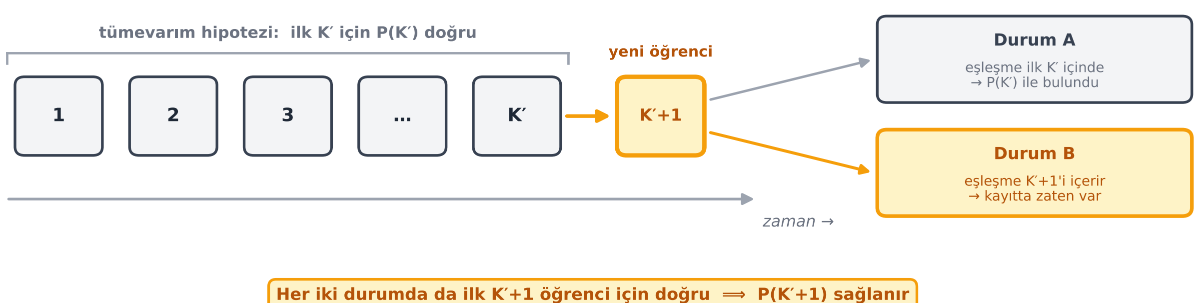
> *"if you communicated to someone else taking this class what your algorithm was, they would be able to code it up and tell a stupid computer how to do that thing."* — Ku, 25:10

Tümevarımla doğruluk ispatı — temel durum, adım ve durum ayrımı

Tümevarımın iki parçası: temel durum + tümevarım adımı



Tümevarım adımının özü: ilk K′+1 öğrenci içindeki eşleşme



Şekil 8.4: Tümevarımla doğruluk ispatının iskeleti. **Üst panel:** ispat iki parçadan oluşur — temel durum P(0) (K = 0 için ifade boş, dolayısıyla doğru / *vacuously true*) ve tümevarım adımı P(K) $\Longrightarrow$ P(K +1) (hipotez doğruysa bir sonraki de doğru); ikisi birleşince her K ≥ 0 için P(K) doğru → algoritma doğru çalışır. **Alt panel:** adımın özü — ilk K öğrenci için P(K) doğru kabul edilir (slate hipotez şeridi), sıraya yeni öğrenci K +1 eklenir (amber vurgu). İlk K +1 öğrenci içindeki eşleşme iki durumdan biridir: **Durum A** eşleşme ilk K içindedir (P(K) ile zaten bulunmuştur), **Durum B** eşleşme K +1'inci öğrenciyi içerir (o öğrencinin doğum günü kayıtta zaten vardır). Her iki durumda da ilk K +1 öğrenci için doğru $\Longrightarrow$ P(K +1) sağlanır.

8.7 Verimlilik: Zamanı Değil, İşlemi Say

Algoritma doğru; şimdi **verimli** olduğunu savunmalıyız. Verimlilik yalnızca "ne kadar hızlı çalışıyor" değil, "diğer olası yaklaşımlara *kıyasla* ne kadar hızlı" demektir.

> *"Efficiency just means not only how fast does this algorithm run, but how fast does it compare to other possible ways of approaching this problem?"* — Ku, 26:05

Akla ilk gelen ölçüm — kronometreyle süreyi ölçmek — işe yaramaz, çünkü süre **donanıma bağlıdır**: bir kol saati hesaplayıcısı ile IBM araştırma bilgisayarı aynı kodu çok farklı sürelerde çalıştırır. Makineyi denklemden çıkarmak için, her **temel işlemin** sabit zaman aldığını varsayar ve algoritmanın kaç temel işlem yaptığını sayarız.

> *"Instead, count fundamental operations."* — Ku, 27:57

Önemli nüans — girdi boyutu (n): Performans, girdinin *boyutuna* göre ölçülür. Ama n her zaman "eleman sayısı" değildir: $n \times n$ bir dizi için girdi boyutu n^2'dir; bir çizge için genellikle düğüm sayısı artı kenar sayısıdır ($V + E$). Doğru n'i seçmek analizin ilk adımıdır.

8.8 Asimptotik Gösterim: O, Ω, Θ

İşlem sayısını bir fonksiyonla ifade ettikten sonra, onu **asimptotik gösterimle** sadeleştiririz — sabitleri ve düşük dereceli terimleri atıp, girdi büyüdükçe baskın olan büyüme mertebesine bakarız. Üç temel sınır vardır:

- O **(Big-O)** — *üst sınır*: "çalışma süresi en fazla bu mertebede büyür."
- Ω **(Omega)** — *alt sınır*: "çalışma süresi en az bu mertebede büyür."
- Θ **(Theta)** — *sıkı sınır*: hem üstten hem alttan aynı mertebeyle sınırlı (O ve Ω birlikte).

> *"We have big O notation, which corresponds to upper bounds. We will have omega, which corresponds to lower bounds. And we have theta, which corresponds to both."* — Ku, 30:54

Ku, asimptotik gösterimin formal tanımının ertesi gün **recitation'da** (problem oturumu) işleneceğini söyler — bu kurs, ders ile problem oturumunu bilinçli olarak ayırır. Bizim için pratik sezgi şudur: Θ "tam olarak bu hızda", O "en kötü ihtimalle bu hızda" demektir.

8.9 Yaygın Çalışma Süresi Fonksiyonları

Ku, sık karşılaşılan büyüme mertebelerini tahtada en yavaştan en hızlıya sıralar:

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^c \ll 2^n$$

Bu sıralamada **polinom** zamana kadar olan her şey (n^c, c sabit) bu derste "verimli" sayılır. Üstel (2^n) ise pratikte kullanılamaz.

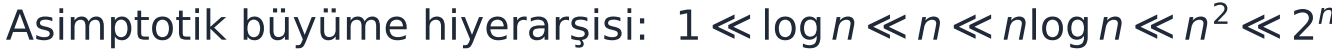
> *“this right here is what we mean by efficient, in this class, usually… generally what we mean is polynomial.”* — Ku, 33:45

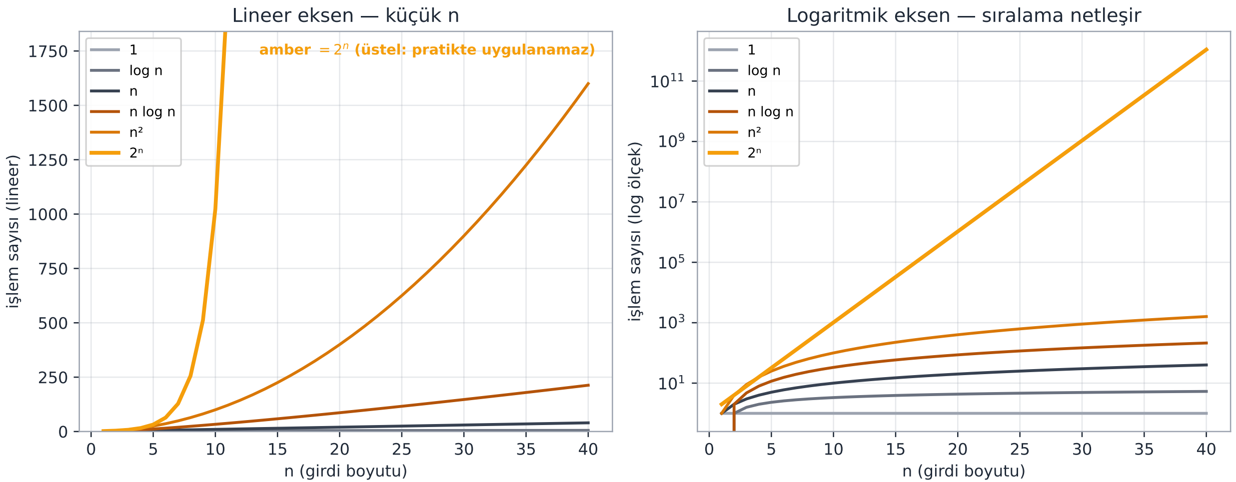
Ku, fonksiyonları $n = 1000$’e kadar çizdiğinde önemli bir sezgi verir: log eğrisi o kadar yavaş büyür ki neredeyse sabit gibi görünür; üstel ise neredeyse dik bir çizgi gibi yukarı fırlar.

> *“So log is almost just as good as constant.”* — Ku, 35:52

> *“So this is crap. This is really good.”* — Ku, 36:07

Burada “crap” (berbat) üstel için, “really good” (çok iyi) ise polinom/altı içindir. Builder sezgisi: bir algoritmanın 2^n mi yoksa n^c mi olduğu, $n = 50$’de bile dünyaları ayırır — biri evrenin yaşından uzun sürer, diğeri milisaniyeler.





Şekil 8.5: Asimptotik büyüme hiyerarşisi $1 \ll \log n \ll n \ll n \log n \ll n^2 \ll 2^n$ iki panelde. **Sol (lineer eksen, $n \leq 40$):** 2^n daha $n \approx 11$’de görünür tavanı aşıp patlar; polinomlar ($n, n \log n, n^2$) tabanda okunur kalır. **Sağ (logaritmik y):** log ölçek tüm eğrileri birbirinden ayırır ve sıralamayı netleştirir — düz çizgiler farklı eğimlerle dizilir. Amber $= 2^n$ (üstel, pratikte uygulanamaz); slate→amber tonlaması yavaştan hızlıya iyi büyümeyi gösterir.

Builder Notu — Big-O ve Model Karmaşıklığı

Bu büyüme hiyerarşisi (Şekil 8.5) ML’de doğrudan karar verir:

- **İleriye → transformer dikkati (attention):** Self-attention, dizinin uzunluğu n’de $O(n^2)$’dir — her token her tokenla eşleşir. n^2 ile $n \log n$ arasındaki uçurum, uzun-bağlam modellerinin neden alt-karesel dikkat aradığını (linear attention; FlashAttention’ın bellek tasarrufu) açıklar.
- **İleriye → neden gradient descent:** Tüm parametre kombinasyonlarını denemek üstel (2^n) — imkânsız. Gradient descent her adımda yalnızca polinom iş yapar; “polinom = verimli” disiplini, kaba-kuvvet aramayı optimizasyonla değiştirmeyi zorunlu kılar.
- **Geriye → SVD rank kesimi (Phase 1 / 18.065):** Bir $m \times n$ matrisi rank-k’ya kesmek, çarpımı $O(mn \cdot \min(m, n))$’den $O(mnk)$’ya indirir. k’yı seçmek, “karşılayabildiğin büyüme mertebesi”

> için doğruluğu takas etmektir — bu dersin asimptotik disiplininin lineer cebire taşınmış hâli.

8.10 Hesaplama Modeli: word RAM

Temel işlemleri "saymak" için, bilgisayarın *neyi* sabit zamanda yapabildiğini tanımlayan bir **hesaplama modeli** gerekir. 6.006'nın kullandığı model **word RAM**'dir (word = kelime, RAM = random access memory).

> *"a machine called a word RAM, which we use for its theoretical brevity."* — Ku, 37:21

İki kavram:

- **Random access memory (RAM):** Bellekteki herhangi bir konuma **sabit zamanda** rastgele erişebiliriz. Bellek, devasa bir bit dizisidir (1'ler ve 0'lar); CPU küçük bir miktar bilgiyi tutup işleyebilir ve bellekten istediği adresi getirip geri yazabilir.

  "Random access memory– it means that I can randomly access different places in memory in constant time." — Ku, 37:44

- **Word (kelime):** CPU'nun bellekten bir seferde alıp üzerinde işlem yapabildiği sabit boyutlu bit öbeği. Bu, modelin "sabit zaman" varsayımının temel birimidir — bir sonraki bölümün konusu.

8.11 Bellek ve Word Boyutu

Modern bilgisayarlar belleği **byte** (8 bitlik öbekler) düzeyinde adresler — her 8 bit için bir adres vardır. CPU bir adres verip o adresteki **word**'ü (kelimeyi) içeri alır, işler ve geri yazar.

> *"A word is how big of a chunk that the CPU can take in from memory at a time and operate on."*
> — Ku, 39:40

Word boyutu neden önemli? Çünkü bir adresi CPU'da saklayabilmek için adresin bir word'e sığması gerekir. Ku kendi gençliğindeki **32-bit** word'lerle bugünkü **64-bit** word'leri karşılaştırır:

- **32 bit** → 2^{32} farklı adres → yaklaşık 4 GB bellek sınırı (eski disklerin neden bölümlere ayrıldığının sebebi).
- **64 bit** → 2^{64} adres → yaklaşık 18 exabyte; pratikte sınırsız (Google'ın tüm sunucularındaki veri ≈ 10 exabyte mertebesinde).

> Builder Notu — GPU'da O(1) Erişim ve Bellek Bandwidth
>
> word RAM'in "her adrese sabit zamanda eriş" varsayımı, gerçek donanımda **kırılır** — ve bu kırılma ML performansının kalbidir:
>
> - **İleriye → cache ve coalesced erişim:** Önbellek hiyerarşisi yüzünden *sıralı* (bitişik) erişim, *rastgele* erişimden kat kat hızlıdır. GPU'da iş parçacıkları bitişik adresleri okuduğunda erişim "coalesced" olur ve tam bant genişliği kullanılır; saçılmış (scatter/gather) erişim bant-genişliği darboğazına takılır.
> - **İleriye → bellek-bağlı çekirdekler:** matmul ve attention gibi birçok ML çekirdeği işlem değil

Bellek (RAM)
indeksli word dizisi — her kutu 1 word (w bit)
adres 0 0xA3F0
adres 1 0x0042
adres 2 0xFFFF
adres 3 0x1C8E
adres 4 0x0000
adres 5 0xCAFE
CPU
read(adres 3)
bir word'ü sabit zamanda oku/yaz
$O(1)$ — word boyutundan bağımsız
Word boyutu ↔ adreslenebilir bellek
32-bit $2^{32} \approx$ 4 GB
64-bit $2^{64} \approx$ 18 EB
w-bit word → adres alanı 2^w;
n girdi sığsın diye $w \geq \log_2 n$

Şekil 8.6: Word RAM modeli şeması: bellek, indeksli bir **word** dizisidir (her kutu w bitlik bir word). CPU, herhangi bir adresteki bir word'ü **sabit zamanda** ($O(1)$, word boyutundan bağımsız) okur/yazar — amber okla vurgulanan `read(adres 3)` işlemi. Sağdaki yan not word boyutunu adreslenebilir bellekle bağlar: 32-bit word $2^{32} \approx 4$ GB, 64-bit word $2^{64} \approx 18$ EB adres alanı sunar. n girdinin sığması için word $w \geq \log_2 n$ bit olmalıdır.

bellek-bandwidth bağlıdır (roofline modeli, aritmetik yoğunluk). "İşlem say" modeli bu trafiği eksik sayar — gerçek hız çoğu zaman taşınan byte miktarına bağlıdır.

- **Geriye → word = pointer boyutu:** 64-bit word, dev tensörleri adresleyebilmenin nedenidir; darboğaz adresleme değil, o adreslere veriyi *taşıma* hızıdır.

8.12 Temel İşlemler

word RAM modelinde CPU'nun **sabit zamanda** yapabildiği işlemler şunlardır:

- İki word'ü **karşılaştırma**.
- **Tam sayı aritmetiği**, **mantıksal işlemler**, **bit düzeyi işlemler** (sonuncusunu bu derste pek kullanmayız).
- Bir bellek adresinden bir word **okuma** ve **yazma**.

> *"I can either do integer arithmetic, logical operations, bitwise operations… And I can read and write from an address in memory, a word in constant time."* — Ku, 41:53

Kritik kısıt: CPU her seferinde yalnızca **sabit miktarda** bilgi (genelde iki word) üzerinde işlem yapar. Lineer miktarda veriyi (n öğe) işlemek istiyorsak — örneğin hepsini okumak — bu **lineer zaman** $O(n)$ alır, çünkü her parçayı ayrı ayrı okumak zorundayız. Sabit zaman ile lineer zaman arasındaki bu ayrım, tüm algoritma analizinin tohumudur.

8.13 Veri Yapıları: İlk Bakış

Dersin ilk yarısı (ilk sekiz ders) **veri yapıları** üzerinedir: CPU gibi sabit miktarda değil, *büyük* miktarda veriyi saklayıp üzerinde verimli işlemler desteklemek. İlk örnek **statik dizidir** (static array).

> *"Python has a lot of really interesting data structures, like a list, and a set, and a dictionary… that are actually not in this model."* — Ku, 44:07

Bu cümle bu kursun en önemli uyarısıdır. Python'da `list`, `set`, `dict` bedava ve sihirli görünür — ama word RAM modelinde *yokturlar*. Aralarında çok sayıda kod katmanı vardır ve o arayüzün ne kadar zaman aldığı her zaman belli değildir.

Builder Notu — Gizli Maliyet ve Sistem Tasarımı

- **Geriye → Python (Phase 1):** `lst.insert(0, x)` bir satır görünür ama $O(n)$'dir (her elemanı kaydırır); `dict[key]` ortalama $O(1)$'dir ama bunun arkasında hashing vardır (Ders 4). Bu kurs o gizli maliyeti açar.
- **Ampirik doğrulama:** `time.perf_counter` ile farklı n değerlerinde süre ölçüp asimptotik tahmini test edebilirsin — teori ile ölçümü buluşturmak builder disiplinidir.
- **İleriye → sistem tasarımı:** "hangi işlem sabit, hangisi lineer" ayrımı, bir veri yapısını (hash map mı, dizi mi, ağaç mı) seçerken verdiğin her kararın temelidir.

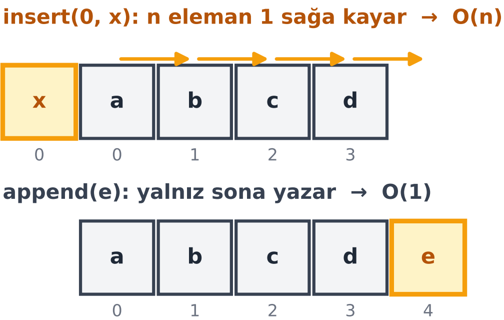
Egzersiz 4'teki ölçüm deneyi bu uçurumu *ampirik olarak* doğrular — aşağıdaki Egzersizler bölümünde kodu var.

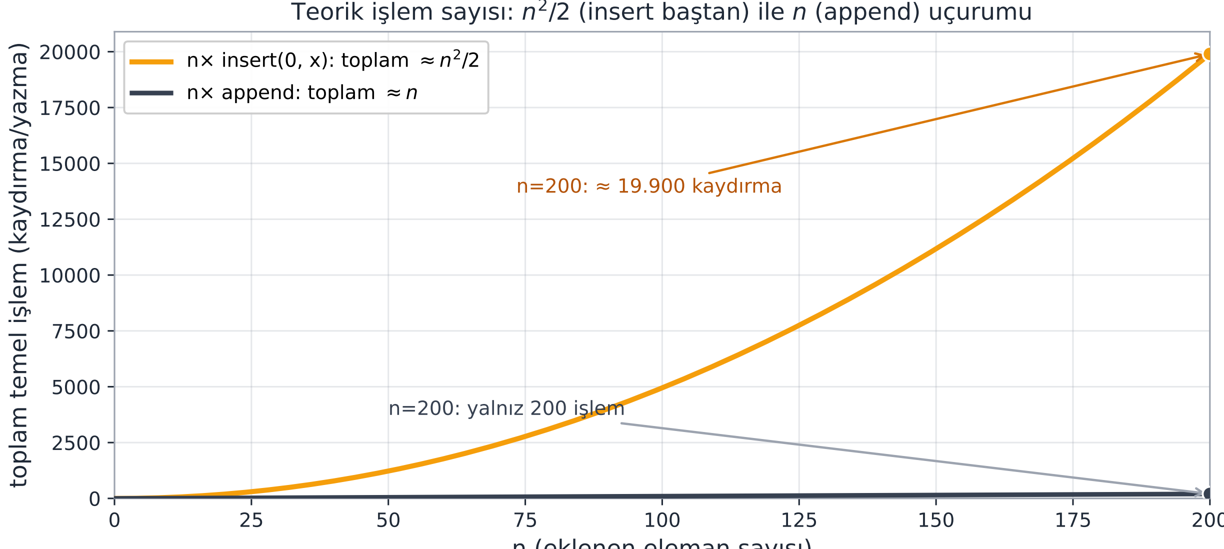
8.14 Bu Dersin Özeti

1. **Problem**, girdiler ile çıktılar arasında bir ikili ilişkidir; **algoritma**, o ilişkiyi hesaplayan bir fonksiyon/prosedürdür.
2. Bir algoritmanın doğruluğu, keyfi büyük girdiler için **tümevarımla** ispatlanır (hipotez + temel durum + tümevarım adımı).
3. Verimlilik, saniyeyle değil, **temel işlem sayısıyla** ölçülür — donanımdan bağımsız olmak için.
4. **Asimptotik gösterim**: O (üst sınır), Ω (alt sınır), Θ (sıkı sınır).
5. Büyüme hiyerarşisi: $1 \ll \log n \ll n \ll n \log n \ll n^2 \ll 2^n$. Polinom ve altı "verimli", üstel kullanılamaz.
6. **word RAM** modeli: belleğe sabit-zaman rastgele erişim; bir word, CPU'nun bir kerede işlediği bit öbeği.
7. Python'ın `list`/`dict`/`set` yapıları word RAM modelinde doğrudan yoktur — bedava görünen işlemlerin gizli zaman maliyeti vardır.

! Tek Bir Cümle

Bir algoritmayı anlamak, onun *ne* yaptığını değil; **neden doğru** ve **ne kadar hızlı** olduğunu — ve bunu başkasına nasıl kanıtlayacağını bilmektir.





Şekil 8.7: Python `list` gizli maliyeti: `insert(0, x)` baştan ekleme, mevcut tüm n elemanı bir konum sağa kaydırır (amber oklar) → $O(n)$; `append` yalnız sona tek hücre yazar → $O(1)$. Alttaki teorik işlem-sayısı eğrisi DETERMİNİSTİKtir (zaman ölçümü değil): n kez baştan ekleme toplam $\approx n^2/2$ kaydırma yaparken, n kez `append` toplam $\approx n$ işlemde kalır. $n = 200$'de uçurum ≈ 19.900'e karşı 200.

8.15 Kontrol Soruları

Soru 1: Problem ile algoritma arasındaki fark nedir?

Cevap: Bir **problem**, girdiler ile çıktılar arasında bir ikili ilişkidir — *ne* hesaplanacağını söyler (hangi girdiye hangi çıktı doğru). Bir **algoritma** ise o ilişkiyi gerçekleştiren bir prosedürdür — *nasıl* hesaplanacağını söyler. Problem çıktıları bilir (ilişkiyi tanımlar); algoritma çıktıları üretir. Matematiksel olarak algoritma, her girdiyi tek bir (doğru) çıktıya eşleyen bir fonksiyondur.

Soru 2: n = 1.000.000 girdi için O(n) bir algoritma mı yoksa O(n^2) bir algoritma mı tercih edilir? Neden?

Cevap: $O(n)$ tercih edilir. $n = 10^6$ için $O(n) \approx 10^6$ işlem (milisaniyeler), $O(n^2) \approx 10^{12}$ işlem (mertebe olarak dakikalar–saatler) demektir. Asimptotik fark, n büyüdükçe pratik bir uçuruma dönüşür. Ku'nun deyişiyle: polinom "verimli", ama düşük dereceli polinom her zaman daha iyidir.

Soru 3: Doğum günü algoritmasının tümevarım ispatında temel durum neden K = 1 değil de K = 0 seçilir?

Cevap: $K = 0$, ispatın *en kolay* ve *en sağlam* tabanıdır: hiç öğrenci görüşülmediğinde hiçbir iş yapılmamıştır ve ilk 0 öğrenci bir eşleşme içeremez, dolayısıyla hipotez **boş-doğru** (vacuously true) olarak sağlanır. $K = 1$ veya $K = 2$ de işe yarar ama gereksiz yere durum kontrolü gerektirir. Daha zayıf (daha kolay) temel durum, daha temiz bir ispattır.

Soru 4: Python'da lst.insert(0, x) neden sabit zaman O(1) değildir? Bu, word RAM modeliyle nasıl ilişkilidir?

Cevap: `lst.insert(0, x)`, listenin başına ekleme yapar; bunun için mevcut tüm n elemanı bir konum sağa kaydırmak gerekir → $O(n)$. Tek satırlık "bedava" görünen bu işlem, word RAM modelinde lineer sayıda word okuma/yazma anlamına gelir. Ku'nun uyarısı tam buna işaret eder: Python yapıları modelde doğrudan yoktur; gerçek maliyeti görmek için arayüzün altına bakmak gerekir.

8.16 Egzersizler

Egzersiz 1. Şu problemi girdi-çıktı ilişkisi olarak tanımla: "bir tam sayı dizisinde en büyük elemanı bul." Girdi kümesi, çıktı kümesi ve doğruluğu kontrol eden yüklemi (predicate) yaz.

Egzersiz 2. Doğum günü algoritmasının çalışma süresini analiz et. "Kayıtta var mı?" kontrolü her seferinde tüm kaydı lineer taramayla yapılırsa toplam süre ne olur? Bu kontrol $O(1)$ olsaydı (örneğin hash ile) toplam süre ne olurdu?

Egzersiz 3. Tümevarım adımındaki "Durum B"yi kendi cümlelerinle yeniden ifade et: ilk K +1 öğrenci içinde bir eşleşme varsa, bu eşleşme neden *zorunlu olarak* K +1'inci öğrenciyi içerir?

Egzersiz 4. Python'da aşağıdaki deneyi çalıştır; baştan ekleme (`insert(0, x)`) ile sona ekleme (`append`) sürelerini farklı n değerlerinde karşılaştır ve asimptotik farkı gözlemle:

```
import time

def measure(n):
    a, b = [], []
    t0 = time.perf_counter()
    for i in range(n):
        a.insert(0, i)      # bas: her ekleme O(n) -> toplam O(n^2)
    t1 = time.perf_counter()
    for i in range(n):
        b.append(i)         # son: amortize O(1) -> toplam O(n)
    t2 = time.perf_counter()
    print(n, "insert(0):", t1 - t0, "append:", t2 - t1)

for n in (1000, 2000, 4000, 8000):
    measure(n)
```

Egzersiz 5. Statik bir dizide "i'inci elemana eriş" işlemi neden $O(1)$'dir (word RAM modelinde)? Bu, bir bağlı listede (linked list) neden $O(1)$ *değildir*? Ders 2'nin dinamik dizi konusuna bir köprü kur.

8.17 Sonraki Ders İçin Hazırlık

Ders 2 (L2): Veri Yapıları ve Dinamik Diziler

Erik Demaine ile dersin ilk veri yapısı bölümüne giriyoruz. Statik dizinin sınırını (sabit boyut) aşan **dinamik dizi** (Python `list`'in altında yatan yapı) ve **bağlı liste** ele alınır; sona eklemenin neden **amortize** $O(1)$ olduğunu göreceğiz — bu dersteki "gizli maliyet" sorusunun ilk cevabı.

Ders 2 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle Egzersiz 4'ü (Python ölçümü) çöz.
- `dizi (sequence)` arayüzünü düşün: hangi işlemler $O(1)$, hangileri $O(n)$ olmalı?
- Ana cümleyi tekrar oku: *"Bir algoritmayı anlamak, onun ne yaptığını değil; neden doğru ve ne kadar hızlı olduğunu bilmektir."*

8.18 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Problem	Girdi-çıktı ikili ilişkisi (binary relation)	Böl. 2
Algoritma	İlişkiyi hesaplayan prosedür/fonksiyon	Böl. 3
Tümevarımla doğruluk	Hipotez + temel durum ($K = 0$) + tümevarım adımı	Böl. 5

Kavram	Tanım	Sayfada
Verimlilik	Temel işlem sayısı (saniye değil)	Böl. 6
O / Ω / Θ	Üst / alt / sıkı asimptotik sınır	Böl. 7
Büyüme hiyerarşisi	$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll 2^n$	Böl. 8
Polinom = verimli	n^c sınırı; üstel (2^n) kullanılamaz	Böl. 8
word RAM	Sabit-zaman rastgele erişimli hesaplama modeli	Böl. 9
Word	CPU'nun bir kerede işlediği bit öbeği	Böl. 10
Temel işlemler	Aritmetik, mantık, oku/yaz; her biri $O(1)$	Böl. 11
Statik dizi	İlk veri yapısı; Python list/dict modelde yok	Böl. 12

8.19 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, ileri algoritma ve sistem mühendisliğinin temel dilini kurar — köprülerin özeti:

1. **Asimptotik sezgi** → kompetitif programlama: "$n = 10^5$ için $O(n^2)$ geçer mi?" sorusunun cevabı bu derste oturur.
2. **word RAM modeli** → sistem tasarımı: hangi işlemin gerçekten sabit zaman olduğunu bilmek, veri yapısı seçiminin temelidir (hash map, dizi, ağaç).
3. **Tümevarımla doğruluk** → OMSCS CS 6515: graduate algoritma derslerinde her algoritma ispatla sunulur; bu dersin disiplini oraya taşınır.
4. **İndirgeme (reduction) sezgisi** → "bilinen bir probleme indirge" yaklaşımı, hem 6.006'nın hem de ileri derslerin çekirdek stratejisidir.
5. **Python gizli maliyet** → üretim kodu: `list.insert(0, x)` veya `x in list` gibi "masum" çağrıların asimptotik bedelini görmek, ölçeklenen sistemlerde performans hatalarını önler.
6. **`time.perf_counter` ile ampirik doğrulama** → teori ile ölçümü buluşturmak; bir builder, asimptotik tahmini deneyle test eder.

! Tek bir şey alıp gideceksen

Bir algoritma yazmak yarısıdır; diğer yarısı onun *doğru* ve *verimli* olduğunu — başkasını ikna edecek netlikte — kanıtlamaktır. 6.006 boyunca her ders bu iki yarıyı birlikte öğretir.

9 Veri Yapıları ve Dinamik Diziler

Arayüz vs veri yapısı, statik dizi / bağlı liste / dinamik dizi, geometrik seri ve amortize analiz

Bölüm bilgisi

- **Demaine'in videosu:** YouTube — Lecture 2: Data Structures and Dynamic Arrays (≈50 dk)
- **OCW sayfası:** MIT 6.006 Lecture 2: Data Structures and Dynamic Arrays
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 2
- **Hocalar:** Erik Demaine (Jason Ku, Justin Solomon)
- **Okuma süresi:** ≈24 dk

9.1 Bu Derste Ne Var?

Ders 1 problem ile algoritma arasındaki farkı kurdu. Bu ders, aynı ayrımı veri tarafında yapar: **arayüz (interface)** ile **veri yapısı (data structure)** arasındaki fark. Erik Demaine, dersin ilk veri yapıları bölümüne enerjik bir girişle başlar.

Üç temel kavram bu derste yan yana gelir:

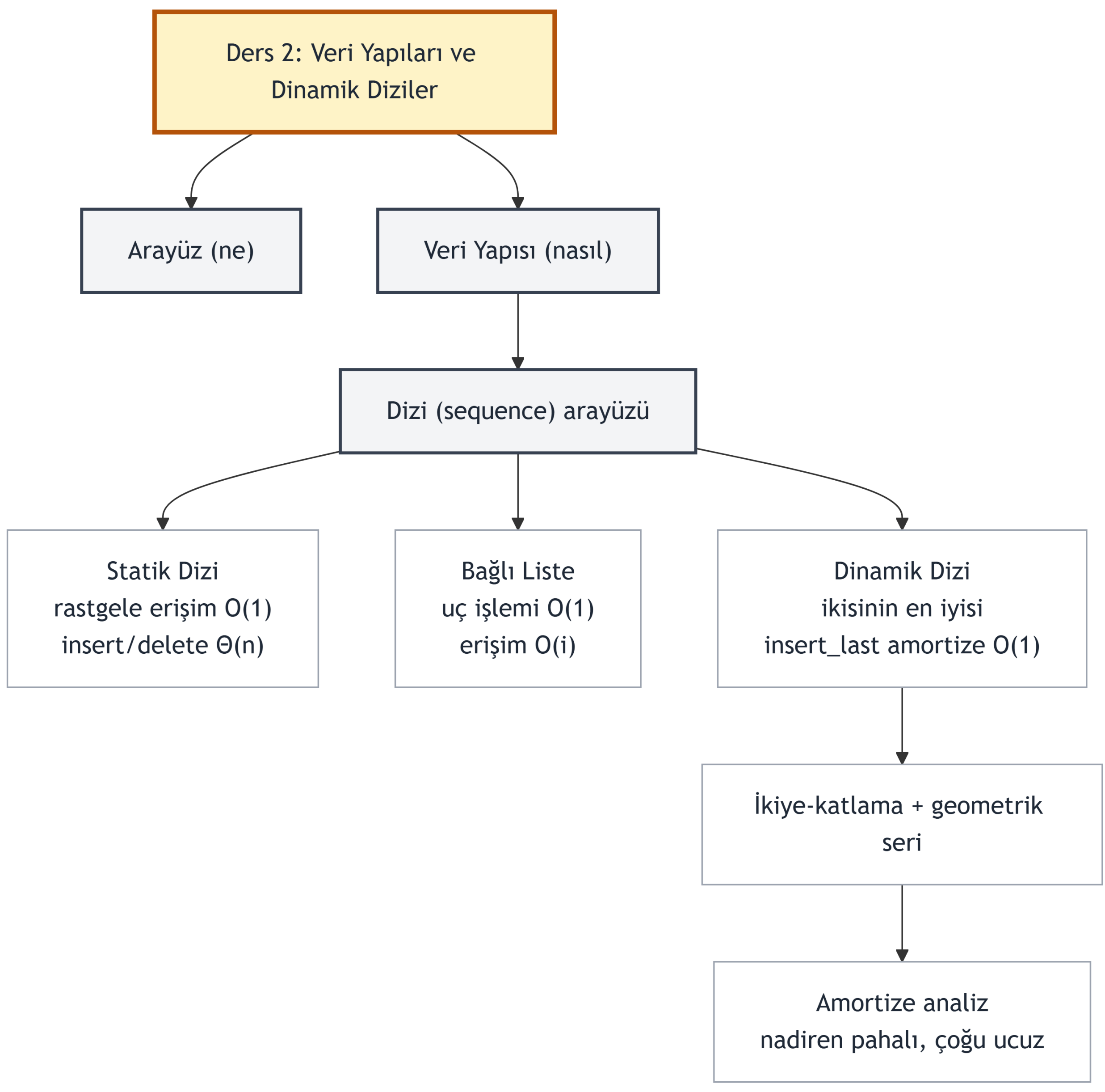
1. **Arayüz vs veri yapısı** — arayüz *ne* yapılacağını (desteklenen işlemler), veri yapısı *nasıl* yapılacağını (gerçekleştirim) söyler.
2. **Dizi (sequence) arayüzü** — n öğeyi sıralı tutma; statik dizi, bağlı liste ve dinamik dizi ile çözülür.
3. **Amortize analiz** — dinamik dizinin (Python `list`) neden sona eklemede amortize $O(1)$ olduğunun ispatı.

> *"an interface says what you want to do. A data structure says how you do it."* — Demaine, 1:18

Builder Notu — Arayüz/Implementasyon Ayrımı ve PyTorch

Demaine'in arayüz/veri yapısı ayrımı, modern ML kütüphanelerinin temel tasarım desenidir — aynı sözleşme (interface), değişebilir gerçekleştirim (implementation):

- **İleriye → PyTorch `nn.Module`:** Bir katmanın `forward()` arayüzü *ne* hesaplanacağını söyler; arkadaki tensör depolaması, bellek düzeni ve çekirdek (kernel) seçimi *nasıl* yapılacağını. Aynı `nn.Linear` API'si CPU, CUDA veya MPS backend'iyle çalışır — arayüz sabit, veri yapısı değişir.
- **Geriye → Python (Phase 1):** Demaine açıkça söyler, bu giriş dersleri "Python'ın nasıl gerçekleştirildiğini" anlatır. Python `list` = dinamik dizi; bu ders onun iç mekanizmasını açar.
- **Geriye → word RAM (Ders 1):** statik dizinin $O(1)$ erişimi doğrudan Ders 1'in word RAM



Şekil 9.1: Ders 2'nin kavram haritası: arayüz (ne) ile veri yapısı (nasıl) ayrımından, dizi arayüzünü çözen üç gerçekleştirime — statik dizi (rastgele erişim O(1)), bağlı liste (uç işlemi O(1)), dinamik dizi (ikisinin en iyisi).

modeline dayanır (ardışık bellek + rastgele erişim).

Tek cümle: *Aynı arayüzü farklı veri yapıları çözer; doğru seçim, hangi işlemin ne sıklıkta çağrıldığına bağlıdır.*

9.2 Arayüz mü, Veri Yapısı mı?

Demaine, bir programcının **API**, eski kuşak bir algoritmacının **ADT** (abstract data type) dediği şeyi **arayüz** olarak adlandırır. Ayrım nettir:

- **Arayüz** = *ne*: hangi veriyi saklayabilirsin, hangi işlemleri destekler, bu işlemler ne anlama gelir. Bu bir **spesifikasyondur** (problem ifadesinin veri karşılığı).
- **Veri yapısı** = *nasıl*: verinin somut gösterimi ve işlemleri destekleyen **algoritmalar**. Algoritmalar tam da burada devreye girer.

> *"The same problem can be solved by many different data structures."* — Demaine, 3:23

Veriyi sadece saklamak sıkıcıdır (bir dosyaya atarsın). İşi *ilginç* kılan, o veri üzerindeki **işlemlerdir** — ve farklı veri yapıları bazı işlemleri diğerlerinden hızlı destekler. Doğru veri yapısı, ne için kullanacağına bağlıdır.

9.3 İki Temel Arayüz: Küme ve Dizi

6.006 iki ana arayüze odaklanır:

- **Küme (set):** öğeleri *değerlerine* (anahtar/key) göre tutar — sıralı tutma, bir değeri arama. "Set" bir matematikçi için başka, bir Python programcısı için başka şey ifade eder.
- **Dizi (sequence):** öğeleri *belirli bir sırada* tutar — örneğin 5, 2, 9, 7 sayılarını bu sırayla. Python'da bunu bir `list` ile saklarsın; sırayı korur.

Bu ders **dizi** arayüzüne odaklanır (küme arayüzüne sonda değinilir, Ders 3-4'te derinleşir). İki temel araç vardır: **diziler (arrays)** ve **işaretçiler (pointers / bağlı yapılar)**. Bugün ikisini de göreceğiz.

9.4 Statik Dizi Arayüzü

En basit dizi arayüzü **statik dizidir**: öğe sayısı n değişmez (ama öğelerin kendisi değişebilir). Öğeler $x_0, x_1, \dots, x_{n-1}$ olarak etiketlenir (Python'daki gibi 0'dan başlar). Desteklenen işlemler:

- **build(x):** verilen öğelerden n boyutlu yeni bir dizi kur.
- **length():** sabit n değerini döndür.
- **iter():** öğeleri x_0'dan x_{n-1}'e sırayla üret (en az lineer zaman).
- **get_at(i):** i'inci öğeyi (x_i) döndür.
- **set_at(i, x):** i'inci öğeyi yeni bir değerle değiştir.

`build` ve `iter` doğası gereği tüm öğelere dokunur → $\Theta(n)$. Asıl ilginç olan, dizinin *ortasına* dinamik erişim: get_at ve set_at.

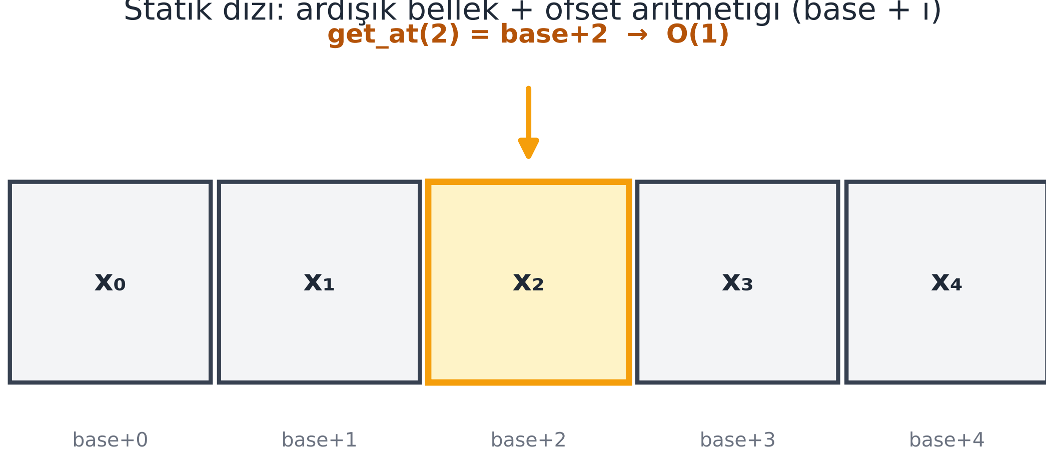
9.5 Statik Dizi (Static Array)

Bu arayüzün doğal çözümü **statik dizidir** (Python'da `list` denir; Demaine "array" demeyi tercih eder). Python'da aslında statik dizi yoktur — yalnızca dinamik diziler vardır — ama kavramı anlamak gerekir.

Statik dizi, belleğin **ardışık (consecutive)** bir parçasıdır. Ders 1'in word RAM modeline dayanır: bellek, w-bitlik word'lerden oluşan bir dizidir ve herhangi bir word'e sabit zamanda rastgele erişilebilir. Bir diziye i indeksinden erişmek, basit bir **ofset aritmetiğidir** (Şekil 9.2): dizinin başlangıç adresi (Python'da `id(array)`) artı i.

> *"as long as I store my array consecutively in memory, I can access the array in constant time."*
> — Demaine, 11:26

Bu yüzden statik dizi, get_at ve set_at işlemlerini $O(1)$ zamanda çözer. length de $O(1)$'dir (n'i adresle birlikte saklarız). build ve iter $\Theta(n)$'dir. Bu çalışma süreleri **optimaldir** — basit ama bu modele giderek daha çok ihtiyaç duyacağız.



Şekil 9.2: Statik dizi: öğeler $x_0 \ldots x_4$ **ardışık bellekte** yan yana durur. i'inci öğenin adresi basit bir **ofset aritmetiğiyle** bulunur: `base + i`. Bu yüzden `get_at(i)` herhangi bir indekse tek hamlede ulaşır — $O(1)$ **rastgele erişim** (amber vurgulu x_2). Bu, Ders 1'in word-RAM modeline dayanır: bir adres tek word'e sığar ve bellek hücresine sabit zamanda erişilir. Bedeli, dizinin boyutunun sabit olmasıdır.

9.6 Bellek Ayırma Modeli ve Word Boyutu

build'in nasıl çalıştığını tanımlamak için **bellek ayırma modeli** gerekir. En temiz varsayım: n boyutlu bir diziyi $\Theta(n)$ zamanda ayırabiliriz.

> *“the cleanest one is just to assume that you can allocate an array of size n in theta n time.”* — Demaine, 13:09

Neden sabit değil de lineer? Çünkü ayrılan bellek başlangıçta belirsizdir; onu 0’larla başlatmak lineer maliyet ister. Bu modelin güzel bir yan etkisi: kullandığın yer (space), kullandığın zamandan (time) fazla olamaz. Sonsuz boyutlu bir dizi ayırıp birkaç öğe kullanmak gerçekçi değildir — bellek ayırmanın bir bedeli olmalıdır.

Word boyutu ($w \geq \log n$): Dizi erişiminin sabit zaman olması için, word boyutu w en az $\log n$ kadar olmalıdır. Sebep: n öğeyle çalışıyorsak, en azından onları **adresleyebilmeliyiz** — “i’inciyi ver” diyebilmek için i sayısını bir word’e sığdırmalıyız. Bu yüzden $w \geq \log n$.

> *“the word size has to grow with n… at least as fast as log n.”* — Demaine, 15:13

Bu, gerçek makineler (sabit boyut) ile teori (ölçeklenebilirlik) arasında köprü kurar: gerçekte daha büyük girdi için daha çok RAM alırsın; teoride de w, n ile birlikte asimptotik olarak büyür.

9.7 Dinamik Dizi Arayüzü

Statik arayüze iki **dinamik işlem** ekleriz: dizinin ortasına **insert_at** ve ortasından **delete_at**. insert_at, set_at’ten farklıdır: hiçbir bilgiyi silmez; eklenen konumdan sonraki tüm öğeler indekslerini bir kaydırır ($n' = n + 1$).

İndeksleme inceliği önemlidir: insert_at(2, x)’ten sonra get_at(2) yeni öğeyi, get_at(3) eski x_2’yi döndürür. Bu izlemesi zordur ama uçlarda (baş/son) ekleme/silme yaparken çok kullanışlıdır. Bu yüzden özel durumları tanımlarız: insert/delete **first** ve **last**, get/set **first** ve **last**.

> *“insert_last doesn’t change the indices of any of the old items.”* — Demaine, 19:59

Kritik gözlem: insert_last eski öğelerin indekslerini değiştirmez (güzel özellik); insert_first hepsini $+1$ kaydırır. Tüm dinamik dizi arayüzünü sabit zamanda çözmek **imkânsızdır** (ispatlanabilir), ama özel durumları — yalnızca uçlardan ekleme/silme — verimli çözebiliriz. İşte bu yüzden özel durumlar ilginçtir.

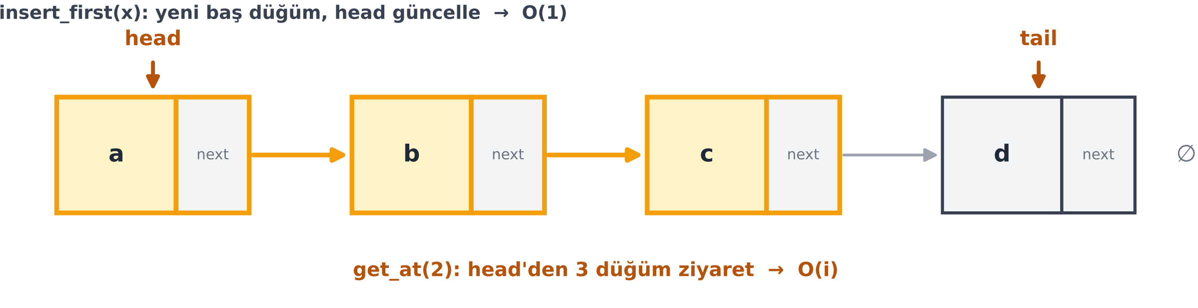
9.8 Bağlı Liste (Linked List)

Dinamik diziyi çözen ilk veri yapısı **bağlı listedir**. Öğeleri **düğümlerde (node)** saklarız; her düğümde bir **item** alanı ve bir **next** (sonraki) işaretçisi vardır — iki sınıf değişkenli bir nesne gibi. Düğümler next işaretçileriyle birbirine bağlanır; sıra, bu işaretçilerle örtük olarak temsil edilir (Şekil 9.3). Veri yapısı bir **head** (baş) işaretçisiyle temsil edilir.

Bu, **işaretçi tabanlı** bir veri yapısıdır (statik dizi ise dizi tabanlıydı). İşaretçiler tek bir word’e sığar, bu yüzden word RAM modelinde sabit zamanda dereference edilebilir (işaret edilen düğüme bakılabilir). Düğümler bellekte rastgele sırada durur — işaretçiler aslında dev bellek dizisindeki indekslerdir.

> *“linked list, insert- and delete_first are constant time.”* — Demaine, 30:41

Bağlı listenin **uçlarda** dinamik olması güçlüdür: yeni bir baş öğe eklemek (insert_first) sabit zamandır — yeni düğüm ayır, next'ini eski başa bağla, head'i güncelle. Ancak **rastgele erişim** zayıftır: i'inci öğeye ulaşmak için i kez next takip etmek gerekir → get_at/set_at $O(i)$, en kötü durumda $\Theta(n)$. Statik dizinin tam tersi: dizi rastgele erişimde iyi, bağlı liste uçlarda dinamiklikte iyidir.



Şekil 9.3: Bağlı liste: her düğüm bir **(item | next)** kutusudur; sıra yalnız `next` işaretçileriyle örtük tutulur (bellekte düğümler dağınık olabilir). `head` ilk, `tail` son düğüme işaret eder (§8 augmentation) → **insert_first** ve **insert_last** $O(1)$. Rastgele erişim zayıftır: `get_at(2)` head'den `next` zincirini i kez takip eder — burada 3 düğüm ziyareti (amber zincir) → $O(i)$. Son düğümün `next`'i $\emptyset$ (None).

9.9 Veri Yapısı Zenginleştirme (Augmentation)

Küçük bir bulmaca: bağlı listede get_last'ı sabit zamanda nasıl çözeriz? Cevap: son öğeye bir **işaretçi** sakla (genelde **tail** denir). Bu, **veri yapısı zenginleştirmedir** (augmentation): yapıya ekstra bilgi ekleyip onu her zaman güncel tutmak.

> *"data structure augmentation, where we add some extra information to the data structure"* — Demaine, 33:09

tail işaretçisiyle get_last ve insert_last $O(1)$ olur. delete_last ise daha zordur — onun için **çift bağlı liste** (doubly linked list, her düğümde bir de prev işaretçisi) gerekir. Zenginleştirmenin bedeli: yapıyı değiştiren her işlemde ekstra bilgiyi (örneğin tail) güncel tutmak zorundasın.

Builder Notu — Augmentation ve Veritabanı İndeksleri

Augmentation — "yapıya ekstra bilgi ekle, onu güncel tut, bir sorguyu hızlandır" — sistem mühendisliğinin en yaygın takasıdır:

- **İleriye → veritabanı indeksleri:** Bir tabloya B-ağacı veya hash indeksi eklemek tam olarak augmentation'dır: aramayı $O(n)$ tam-taramadan $O(\log n)$'e (veya $O(1)$'e) indirir. Bedeli, tail'i güncel tutmak gibi — her `INSERT/UPDATE/DELETE` indeksi de güncellemek zorundadır (yazma daha pahalı, okuma çok daha ucuz).
- **Sezgi → skip-list / B-tree:** tail ve prev işaretçileri, daha zengin augmentation'ların (skip-list'in

atlama katmanları, B-ağacının iç düğüm anahtarları) ilk basamağıdır; hepsi "ekstra yapısal bilgi sakla, navigasyonu hızlandır" fikrinin türevidir.
- **İleriye → ML feature store:** Önceden hesaplanmış (materialized) öznitelik tabloları, çıkarım anında yeniden hesaplamayı önlemek için tutulan augmentation'lardır — güncel tutma maliyeti (cache invalidation), tail güncellemesiyle aynı disiplini ister.

9.10 Statik Dizi mi, Bağlı Liste mi?

İki veri yapısı **birbirini tamamlar**:

- **Statik dizi:** get_at/set_at sabit zaman $O(1)$ (rastgele erişimde mükemmel), ama insert/delete her yerde $\Theta(n)$ (kaydırma veya yeniden ayırma + kopyalama gerekir). Sona ekleme bile zordur, çünkü ayrılan dizi büyütülemez — yeni bir dizi ayırıp her şeyi kopyalamak $\Theta(n)$'dir.
- **Bağlı liste:** insert/delete_first sabit zaman $O(1)$ (uçlarda dinamiklikte iyi), ama get_at/set_at $O(i)$, en kötü $\Theta(n)$ (rastgele erişimde kötü).

Demaine'in deyişiyle: diziler statik rastgele erişim için, bağlı listeler dinamik uç işlemleri için iyidir. Hiçbiri *her ikisini* birden iyi yapmaz. Hedefimiz, ikisinin de güçlü yanlarını birleştiren bir yapı. Bu tercihin gerçek donanımdaki bir başka boyutu da **bellek yerelliğidir** (Şekil 9.4): aynı asimptotik tarama, ardışık dizide bağlı listeden çok daha hızlı koşar.

9.11 Dinamik Dizi (Dynamic Array)

Çözüm **dinamik dizidir** — ve bu tam olarak Python'ın `list` dediği şeydir.

> *"another way to describe what these introductory lectures are about is telling you about how Python is implemented."* — Demaine, 34:19

Fikir, statik dizinin katı kuralını **gevşetmektir**: ayrılan dizinin boyutu tam n olmak zorunda değil; *kabaca* n olsun. Algoritma dilinde "kabaca" = sabit çarpanları at. Diziyi boyutu $\Theta(n)$ ve n'den büyük-eşit tutarız (örneğin $2n$).

> *"the size of the array is theta n– probably also greater than or equal to n."* — Demaine, 36:11

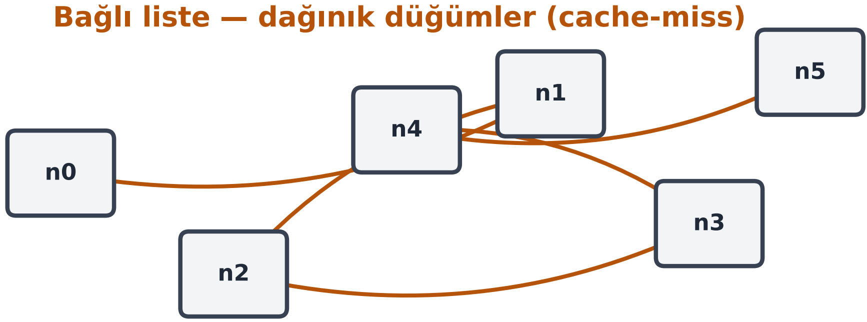
Yapı, bir dizi artı bir **length** (gerçek öğe sayısı) tutar. length, dizinin gerçek boyutu **size**'dan küçük-eşittir; ilk length konum veriyi, kalanı boş yeri temsil eder. insert_last basittir: `a[length] = x`, sonra length'i artır → $O(1)$. Ama bir sorun var: length = size olduğunda boş yer kalmaz. İşte o zaman diziyi büyütmek gerekir — sonraki bölümün konusu.

Bellek yerelliği: ardışık dizi (cache-dostu) vs dağınık liste (cache-miss)

Statik dizi — ardışık bellek (cache-dostu)

tek atlamayla sıralı erişim · yüksek yerellik

Bağlı liste — dağınık düğümler (cache-miss)



her next uzun bir bellek sıçraması · düşük yerellik

Aynı sayıda öğe: dizi tek ardışık blok (önbellek satırı), liste her next'te uzak bir adrese sıçrar

Şekil 9.4: Dizilerin taramada bağlı listeyi yenmesinin donanım sebebi: **bellek yerelliği**. Üstte statik dizi tek **ardışık blok** — öğeler bitişik adreslerde, bir önbellek satırı ($O(1)$ erişim, amber); sıralı tarama tek atlamayla ilerler, yüksek yerellik. Altta bağlı liste düğümleri bellekte **dağınık**; her `next` işaretçisi uzak bir adrese sıçrar (amber oklar), her erişim önbellek-ıskası (cache-miss) riski taşır. Aynı asimptotik $\Theta(n)$ tarama, gerçek donanımda dizide çok daha hızlıdır — düşük yerellik bağlı listeyi cezalandırır.

9.12 Dizi Büyütme ve Geometrik Seri

length = size olduğunda diziyi büyütmek gerekir. Ne kadar? İki doğal seçenek:

- **Sabit ekleme (size + 5):** Kötü. Her 5 adımda bir lineer yeniden ayırma → işlem başına hâlâ lineer zaman. Statik dizinin "her seferinde yeniden ayır" sorununu sadece sabit çarpanla iyileştirir.
- **Sabit çarpan (2 × size):** İyi. Diziyi her dolduğunda **iki katına** çıkar.

Çalışılan Örnek — Doubling'in Toplam Maliyeti

Boş bir diziden başlayıp n kez insert_last yapalım (size = 1 başlasın). Yeniden boyutlandırmalar $n = 1, 2, 4, 8, 16, \ldots$ değerlerinde, yani **2'nin kuvvetlerinde** olur (çünkü ikiye katlıyoruz; Şekil 9.5). Her resize, ayırma + kopyalama nedeniyle lineerdir. Toplam resize maliyeti:

$$1 + 2 + 4 + 8 + \cdots + (\approx n) = \sum_i 2^i \quad (i = 0\text{'dan} \approx \log_2 n\text{'e})$$

Bu bir **geometrik seridir**. Kimliği: $1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1$. Geometrik seriler **son terim tarafından domine edilir**:

> *"Geometric series are dominated by the last term"* — Demaine, 45:46

Son terim $2^{\log_2 n} = n$ olduğundan, toplam $\Theta(n)$'dir. Yani n işlem için toplam yeniden boyutlandırma maliyeti yalnızca $\Theta(n)$ — işlem başına "kabaca sabit".

Builder Notu — İkiye-Katlama Her Yerde

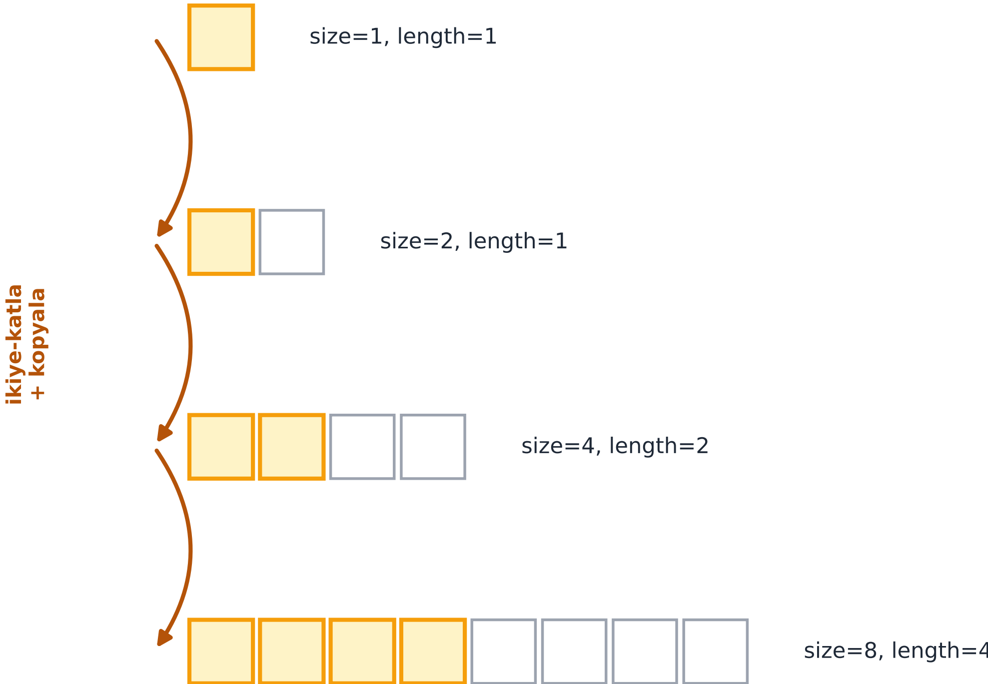
Demaine'in "dolunca iki katına çık" stratejisi, tek bir veri yapısının özelliği değil — ölçeklenen sistemlerin tekrar eden bir motifidir:

- **İleriye → PyTorch `DataLoader` ve `torch.cat`:** Boyutu önceden bilinmeyen bir tampona (buffer) tensör biriktirirken, kapasitenin geometrik büyütülmesi (her `torch.cat` realloc'unda sabit çarpan) toplam kopya işini $\Theta(n)$'de tutar; sabit ekleme olsa $\Theta(n^2)$'ye patlardı.
- **İleriye → exponential backoff:** Başarısız bir isteği yeniden denerken bekleme süresini ikiye katlamak (1s, 2s, 4s, 8s), aynı geometrik seri sezgisidir — toplam bekleme son terimle domine olur, sistem yükü sınırlı kalır.
- **İleriye → batch-size / LR taraması:** Hiperparametre ararken batch boyutunu veya öğrenme oranını ikiye katlayarak (32, 64, 128, …) geometrik bir ızgara taramak, lineer ızgaraya göre çok daha geniş bir aralığı logaritmik sayıda denemeyle kapsar — "son terim toplamı domine eder" disiplini.

9.13 Amortize Analiz (Amortized Analysis)

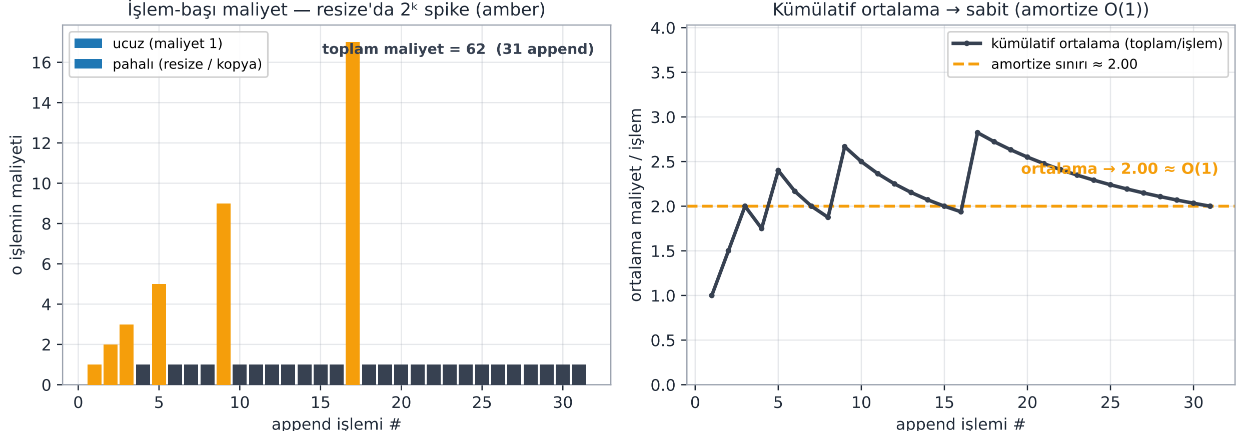
Bu "kabaca sabit" kavramı **amortizasyondur**. Tanım: bir işlem $T(n)$ **amortize zaman** alır, eğer *herhangi* k işlem en fazla $k \cdot T(n)$ toplam zaman alıyorsa.

> *"Amortized means a particular kind of averaging– averaging over the sequence of operations."*
> — Demaine, 47:42



Şekil 9.5: Dinamik dizi büyütme (doubling): dizi dolarken ayrılan boyut **1 → 2 → 4 → 8** olarak ikiye katlanır. Her satır o anki diziyi gösterir — dolu amber hücreler **length** (gerçek öğe sayısı), boş slate hücreler **size − length** (ayrılan ama henüz kullanılmayan yer). Soldaki amber oklar her resize'da **ayır + kopyala** adımını gösterir: `length == size` olunca yeni (iki kat) dizi ayrılır ve eski öğeler taşınır ($\Theta(\text{length})$). Toplam ayrılan + kopyalanan iş bir geometrik seridir: $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ — son terim toplamı domine eder, bu yüzden n append toplam $\Theta(n)$'dir ve `insert_last` amortize $O(1)$ olur.

Doubling'de n işlem toplam $\Theta(n)$ zaman aldığından, insert_last **sabit amortize** (constant amortized) zamandır. Sezgi: tek tek işlemler bazen pahalıdır (resize anında lineer), ama çoğu ucuzdur (sabit). Pahalı işlemin maliyetini, onu mümkün kılan tüm ucuz işlemlere "dağıtırız" — bu, işlem dizisi üzerinde ortalamadır (Şekil 9.6); delete_last için resize gerekmez (length'i azalt, biter), o da sabit amortizedir.



Şekil 9.6: Amortize analiz (İMZA): bir dinamik diziye boş başlangıçtan $n = 31$ kez `append`. **Sol:** her işlemin maliyeti — çoğu yalnız 1 (tek slot yazımı, slate çubuklar); `length == size` olduğunda ikiye-katlama tetiklenir ve o işlemin maliyeti o anki boyut kadar sıçrar (amber çubuklar, append #1, 2, 3, 5, 9, 17'de değerler $1, 2, 3, 5, 9, 17$ — yani o anki boyutu kopyalama + 1 slot yazımı; resize boyutları $1, 2, 4, 8, 16$ (2^k) olduğundan ilk hariç maliyet $2^k + 1$'e sıçrar). **Sağ:** kümülatif ortalama (toplam maliyet / işlem sayısı, slate çizgi) her resize'da yukarı sıçrayıp ardından düşer ve sabit bir değere — tam olarak $62/31 = 2{,}00$ (amber kesik çizgi) — düzleşir. Nadiren pahalı resize'ların maliyeti tüm adımlara yayıldığı için `insert_last` **amortize** $O(1)$'dir; geometrik seri toplamı $\sum_i 2^i = 2^{k+1} - 1 \approx 2n$ bu sabiti garanti eder.

> **Builder Notu — Amortize Düşünme ve Mini-batch/Checkpoint**
>
> "Nadiren pahalı, çoğu zaman ucuz; maliyeti diziye yay" analizi, eğitim ve altyapı kodunun her yerinde aynı biçimde karşına çıkar:
>
> - **İleriye → checkpoint kaydetme:** Bir modeli her N adımda bir diske yazmak pahalıdır (tüm ağırlıkları serialize et), ama bu maliyet aradaki N ucuz eğitim adımına yayıldığında *adım başına* amortize maliyet ihmal edilebilir kalır — tam olarak resize'ın append'lere yayılması gibi.
> - **İleriye → gradient accumulation:** Birkaç mini-batch'in gradyanını biriktirip seyrek aralıklarla bir optimizer adımı atmak, pahalı `step()` çağrısının maliyetini birikim adımlarına amortize eder.
> - **İleriye → rehash ve GC:** Hash tablosu yük faktörü eşiği aşınca yeniden hash'leme ($\Theta(n)$) ve çöp toplama (garbage collection) duraklamaları, doubling resize'la birebir aynı amortize argümanına dayanır: nadir, pahalı, ama dizi boyunca sabit ortalama.

9.14 Üç Veri Yapısı — Karşılaştırma

Demaine dersi bu özet tabloyla kapatır (Θ'lar atılmış, çalışma süreleri; Şekil 9.7):

İşlem	Statik Dizi	Bağlı Liste	Dinamik Dizi
get_at / set_at	$O(1)$	$O(n)$	$O(1)$
insert / delete_first	$O(n)$	$O(1)$	$O(n)$
insert / delete_last	$O(n)$	$O(n)$	$O(1)$ amortize
insert / delete_at	$O(n)$	$O(n)$	$O(n)$

Dinamik dizi, rastgele erişimde statik dizinin hızını ($O(1)$ get_at) ve sona eklemede neredeyse bağlı listenin esnekliğini ($O(1)$ amortize insert_last) birleştirir — ikisinin en iyisi.

Builder Notu — Veri Yapısı Seçimi = Tensor Layout

Maliyet matrisinin verdiği ders — "her yapı bir takas; sıcak yoldaki işleme göre seç" — derin öğrenmede **tensör bellek düzeni** seçiminde birebir tekrar eder:

- **İleriye → contiguous tensor vs Python list:** Bir batch'i Python `list`'inde tutmak (dağınık nesneler, bağlı liste sezgisi) ile tek bir contiguous tensörde tutmak (ardışık dizi sezgisi) arasındaki fark, tam olarak bu matristeki rastgele erişim/tarama takasıdır — vektörize işlemler ardışık belleği ister.
- **İleriye → `.contiguous()` çağrısı:** `transpose`/`permute` sonrası bir tensör mantıksal olarak doğru ama bellekte dağınıktır; `.contiguous()` onu yeniden ardışık düzene kopyalar ($\Theta(n)$, tıpkı resize kopyası gibi) — çünkü birçok CUDA çekirdeği ardışık düzen olmadan çalışmaz veya yavaşlar.
- **İleriye → GPU bellek yerelliği:** Şekil 9.4'nin cache argümanı GPU'da daha da keskindir; coalesced (bitişik) erişim tam bant genişliği kullanır, dağınık erişim çekirdeği bellek-bağlı yapar.

"Hangi işlem sıcak yolda?" sorusu, layout kararıyla doğrudan performansa çevrilir.

9.15 Bu Dersin Özeti

1. **Arayüz** *ne* yapılacağını (işlemler), **veri yapısı** *nasıl* yapılacağını (gerçekleştirim) söyler.
2. İki temel arayüz: **küme** (değere/anahtara göre) ve **dizi** (sıraya göre).
3. **Statik dizi**: ardışık bellek; get_at/set_at $O(1)$, build/iter $\Theta(n)$.
4. **Bağlı liste**: düğüm + next işaretçisi; insert/delete_first $O(1)$, erişim $O(i)$.
5. **Dinamik dizi** (Python `list`): boyut $\Theta(n)$; ikiye-katlama ile insert_last amortize $O(1)$.
6. **Geometrik seri** son terimle domine olur: $1 + 2 + \cdots + 2^k = \Theta(2^k) = \Theta(n)$.
7. **Amortize analiz**: k işlem en fazla $k \cdot T(n)$ zaman alır; nadiren pahalı, ortalama ucuz.

! Tek Bir Cümle

Python `list`'in her "ucuz" append'i, arada bir yapılan pahalı ikiye-katlamanın işlem dizisine **amortize** edilmiş hâlidir — sihir değil, geometrik seri.

Veri yapısı seçimi = işlem profili: her sütun bir takas (Demaine §13)

	Statik Dizi	Bağlı Liste	Dinamik Dizi
get_at / set_at	O(1)	O(n)	O(1)
insert / delete_first	O(n)	O(1)	O(n)
insert / delete_last	O(n)	O(n)	O(1) amortize
insert / delete_at	O(n)	O(n)	O(n)

Şekil 9.7: Demaine'in kapanış maliyet matrisi: 4 işlem ailesi × 3 veri yapısı. Yeşilimsi hücreler $O(1)$ (en iyi, sabit zaman), amber hücreler $O(n)$ (yavaş, lineer), amber-çerçeveli açık hücre ise $O(1)$ amortize (neredeyse en iyi). Hiçbir sütun her satırda kazanmaz: **statik dizi** rastgele erişimde ($O(1)$ get_at/set_at) üstündür ama her ekleme/silme $O(n)$ kaydırma ister; **bağlı liste** uç işlemlerini ($O(1)$ insert/delete_first ve tail augmentation'la last) hızlandırır ama rastgele erişim $O(n)$'e düşer; **dinamik dizi** rastgele erişim $O(1)$'i sona ekleme $O(1)$ amortize ile birleştirip ikisinin en iyisini sunar — bu yüzden Python `list`'in temelidir. Veri yapısı seçimi, hangi işlemlerin sıcak yolda olduğuna bağlı bir takastır.

9.16 Kontrol Soruları

Soru 1: Arayüz (interface) ile veri yapısı (data structure) arasındaki fark nedir?

Cevap: **Arayüz**, *ne* yapılacağını söyler: hangi veriyi saklayabilirsin, hangi işlemler desteklenir ve bu işlemler ne anlama gelir (bir spesifikasyon). **Veri yapısı**, *nasıl* yapılacağını söyler: verinin somut gösterimi ve işlemleri destekleyen algoritmalar. Aynı arayüz (örneğin "dizi") birden çok veri yapısıyla (statik dizi, bağlı liste, dinamik dizi) çözülebilir; her birinin farklı çalışma süreleri vardır.

Soru 2: Çok sık get_at ama nadiren ekleme gereken bir uygulamada hangi veri yapısı uygundur? Tersine, sürekli baştan ekleme gerekiyorsa?

Cevap: Sık rastgele erişim için **statik dizi** veya **dinamik dizi** uygundur (get_at $O(1)$). Sürekli baştan ekleme için **bağlı liste** uygundur (insert_first $O(1)$; dizilerde $O(n)$ çünkü kaydırma gerekir). Eğer hem sık get_at hem sık insert_first gerekiyorsa, tek bir basit yapı ikisini birden $O(1)$ yapamaz — bu ileri veri yapılarının (örneğin dengeli ağaçlar, Ders 6-7) motivasyonudur.

Soru 3: Dinamik dizi dolduğunda neden boyutu size + 1 değil de 2 × size yapılır?

Cevap: size + 1 (ya da sabit ekleme, örn. +5), her birkaç eklemede bir lineer yeniden-ayırma gerektirir → işlem başına hâlâ **lineer** zaman. 2 × size ise yeniden boyutlandırmaları 2'nin kuvvetlerinde seyrekleştirir; n eklemenin toplam resize maliyeti geometrik seri olduğu için $\Theta(n)$'dir → işlem başına **sabit amortize**. Sabit çarpanla büyütme, amortize sabit zamanın anahtarıdır.

Soru 4: Python'da list.append() amortize O(1) ne demektir? Tek bir append neden bazen yavaş olabilir?

Cevap: "Amortize $O(1)$", *herhangi* k append'in toplamda en fazla $k \cdot O(1) = O(k)$ zaman aldığı anlamına gelir — işlem başına ortalama sabit. Ama tek bir append, dizi tam dolduğu anda denk gelirse, yeni (iki kat) dizi ayırıp tüm öğeleri kopyalar → o tek işlem $O(n)$. Bu nadir pahalı işlemin maliyeti, onu izleyen birçok ucuz append'e dağıtılır; dolayısıyla *ortalama* sabit kalır.

9.17 Egzersizler

Egzersiz 1. Statik dizi arayüzünün beş işlemini (build, length, iter, get_at, set_at) listele ve her birinin statik dizi gerçekleştiriminde çalışma süresini yaz. Hangileri $\Theta(n)$, hangileri $O(1)$?

Egzersiz 2. Bağlı listede insert_last'ı $O(1)$ yapmak için hangi zenginleştirme (augmentation) gerekir? Aynı zenginleştirmeyle delete_last neden hâlâ zordur ve bunu çözmek için ne gerekir?

Egzersiz 3. Dinamik dizide n öğe ekleyip sonra hepsini delete_last ile silersen, dizi boyutu hâlâ $\Theta(n)$ kalır (oysa $n = 0$). Bu bellek israfını önlemek için "küçültme" (shrinking) kuralını tasarla. (İpucu: dizi ¼ dolduğunda yarıya indir — neden   veya ½ değil?)

Egzersiz 4. Python'da bir listeye n kez append yaparken iç kapasitenin ne zaman büyüdüğünü gözlemle:

```python
import sys

lst = []
prev = -1
for i in range(64):
    cap = sys.getsizeof(lst)
    if cap != prev:
        print("uzunluk", len(lst), "-> kapasite baytlari", cap)
        prev = cap
    lst.append(i)
```

Kapasite hangi uzunluklarda zıplıyor? Bu, ikiye-katlama (doubling) sezgisini doğruluyor mu?

Egzersiz 5. Küme (set) arayüzü, dizi (sequence) arayüzünden nasıl ayrılır? Öğeleri *değerine/anahtarına* göre aramak (sıraya göre değil), neden farklı bir veri yapısı gerektirir? Ders 3-4'e (sıralama ve hashing) bir köprü kur.

9.18 Sonraki Ders İçin Hazırlık

Ders 3 (L3): Kümeler ve Sıralama

Justin Solomon ile **küme (set)** arayüzüne ve onu çözmenin ilk adımı olan **sıralamaya** (insertion, selection, merge sort) geçiyoruz. Bu derste değindiğimiz "değere göre arama" sorusu orada merkeze oturur. (Not: ders akışında araya **Problem Oturumu 1** girer — bu dersin kavramlarını pekiştiren problemler.)

Ders 3 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle Egzersiz 4'ü (Python kapasite gözlemi) çöz.
- Üç veri yapısı tablosunu (Şekil 9.7) ezberden çizebilecek hâle gel.
- Ana cümleyi tekrar oku: *"Aynı işi farklı veri yapıları farklı maliyetlerle yapar."*

9.19 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Arayüz vs veri yapısı	Ne (spesifikasyon) vs nasıl (gerçekleştirim)	Böl. 1
Dizi (sequence)	Öğeleri belirli bir sırada tutan arayüz	Böl. 2-3
Statik dizi	Ardışık bellek; get_at/set_at $O(1)$	Böl. 4
Bellek ayırma modeli	n boyutlu dizi $\Theta(n)$ zamanda ayrılır	Böl. 5
Word boyutu	$w \geq \log n$ (n öğeyi adresleyebilmek için)	Böl. 5

Kavram	Tanım	Sayfada
Bağlı liste	Düğüm + next; insert/delete_first $O(1)$	Böl. 7
Augmentation	Ekstra bilgi (tail) ile $O(1)$ get_last	Böl. 8
Dinamik dizi	Python list; boyut $\Theta(n)$, ikiye-katlama	Böl. 10
Geometrik seri	$1+2+\cdots+2^k = \Theta(2^k) = \Theta(n)$	Böl. 11
Amortize analiz	k işlem en fazla $k \cdot T(n)$ zaman	Böl. 12

9.20 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, ileri algoritma ve sistem mühendisliğinin temel dilini kurar — köprülerin özeti:

1. **Amortize analiz** → altyapı: log-structured storage, hash tablosu rehashing, garbage collection — “nadiren pahalı, çoğu zaman ucuz” her yerde aynı analiz.
2. **Dinamik dizi** → Python `list`, C++ `vector`, Java `ArrayList`’in iç mekanizması; kapasite büyütme stratejisi.
3. **Arayüz / veri yapısı ayrımı** → API tasarımı: aynı sözleşme (interface), değişebilir backend (implementation).
4. **word RAM + işaretçiler** → bellek modeli ve cache yerelliği; dizi (ardışık) ile bağlı liste (dağınık) arasındaki gerçek performans farkı.
5. **Augmentation** → veritabanı indeksleri: ekstra bir yapı (B-ağacı, hash) ekleyip sorguyu hızlandırma — aynı augmentation fikri.
6. **Geometrik seri sezgisi** → ikiye-katlama/yarıya-indirme stratejileri: connection pool, buffer büyütme, exponential backoff.

! Tek bir şey alıp gideceksen

Aynı işi farklı veri yapıları farklı maliyetlerle yapar; mühendislik, “hangi işlem ne sıklıkta çağrılıyor?” sorusuna göre doğru yapıyı seçmektir. Python `list`’in zarafeti, bu seçimi amortize analizle senin için çoktan çözmüş olmasıdır.

10 Problem Oturumu 1

Ders 1-2'nin uygulaması: asimptotik sıralama, sequence black box, çift uçlu dinamik dizi ve yerinde bağlı liste ters çevirme

ℹ Oturum bilgisi

- **Ku'nun videosu:** YouTube — Problem Session 1 (≈60 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 1
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 3 (Problem Oturumu 1)
- **Hoca:** Jason Ku
- **Okuma süresi:** ≈26 dk

10.1 Bu Problem Oturumu Ne Hakkında?

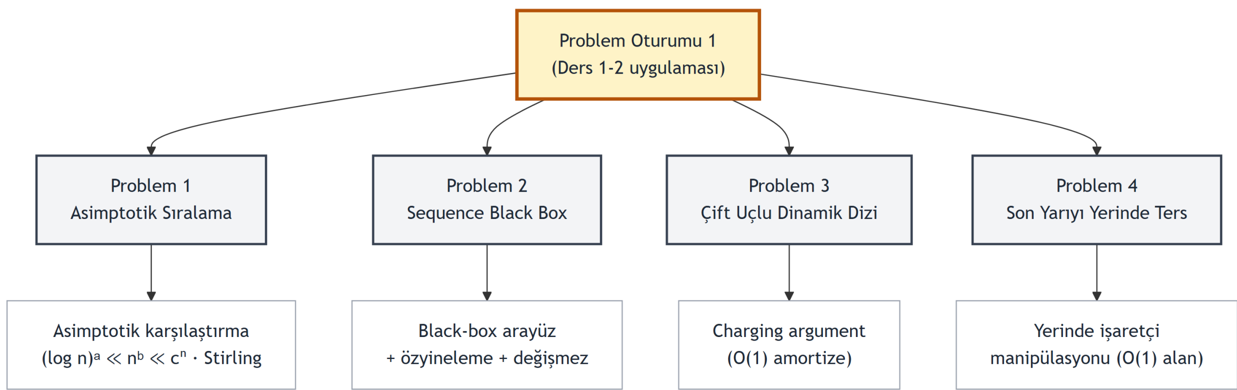
6.006'da iki tür öğretim vardır: **ders (lecture)** temel materyali (veri yapıları, algoritmalar) sunar; **problem oturumu (problem session)** ise o materyalin *uygulamasını* gösterir. Jason Ku'nun deyişiyle, problemlerin "hissi" temel materyalden çok farklıdır — problemlere yaklaşmanın, ancak çalışarak öğrenilen püf noktaları vardır.

Bu ilk oturum, Ders 1-2'nin kavramlarını uygular: **asimptotik analiz** (Problem 1), **sequence arayüzü ve özyineleme** (Problem 2), **dinamik dizi** (Problem 3) ve **bağlı liste manipülasyonu** (Problem 4). Dört problem birer "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir. Ortak araçların haritası Şekil 10.1 içinde özetlenir.

Bir genel uyarı tüm oturuma damga vurur: çözümü **kod değil, kelimelerle** anlat. Ku, kafasında kod derleyemediğini söyler; problem setlerinde de net sözel açıklama beklenir.

> *"the problem sets that you will work on… applications of that material… there's usually a much different feel between those problems then the underlying foundational material."* — Ku, 1:04

> *"I want you to explain in words to me, and we want you to explain in words in your LaTeX submissions what it is the algorithm is doing."* — Ku, 30:37



Şekil 10.1: Problem Oturumu 1'in kavram haritası: dört problem (üst sıra) ve her birinin öğrettiği taşınabilir araç (alt sıra). Problem 1 asimptotik karşılaştırma (Stirling + hiyerarşi), Problem 2 black-box arayüz disiplini + özyineleme, Problem 3 charging argument (amortize), Problem 4 yerinde işaretçi manipülasyonu kazandırır. Hepsi Ders 1-2 temeline dayanır.

10.2 Problem 1: Asimptotik Sıralama

İfade. Bir fonksiyon listesini asimptotik büyüme hızına göre (yavaştan hızlıya) sırala. Asimptotik olarak *eşit* olan fonksiyonlar küme parantezi {} içinde gruplanır. Örneğin n, $\sqrt{n}$, $n+\sqrt{n}$ için cevap: $\sqrt{n}$, sonra $\{n,\ n+\sqrt{n}\}$.

> **Yaklaşım — Tanıdık Forma İndirge, Sonra Hiyerarşiyi Uygula**
>
> Her fonksiyonu **tanıdık bir forma** (polinom-benzeri) çevir, sonra şu hiyerarşiyi uygula: logaritmik faktörler polinom faktörlerden, polinomlar da üstellerden yavaş büyür. Karmaşık ifadeler (faktöriyel, binom katsayısı) için **Stirling yaklaşımı** kullanılır. Bir fonksiyonu sınıflandıramıyorsan, onu faktöriyel/binom tanımına aç, Stirling uygula, sadeleştir — ortaya polinom-benzeri bir form çıkar ve karşılaştırma kolaylaşır.

Çözüm. Üç temel araç bu problemin tüm varyantlarını çözer:

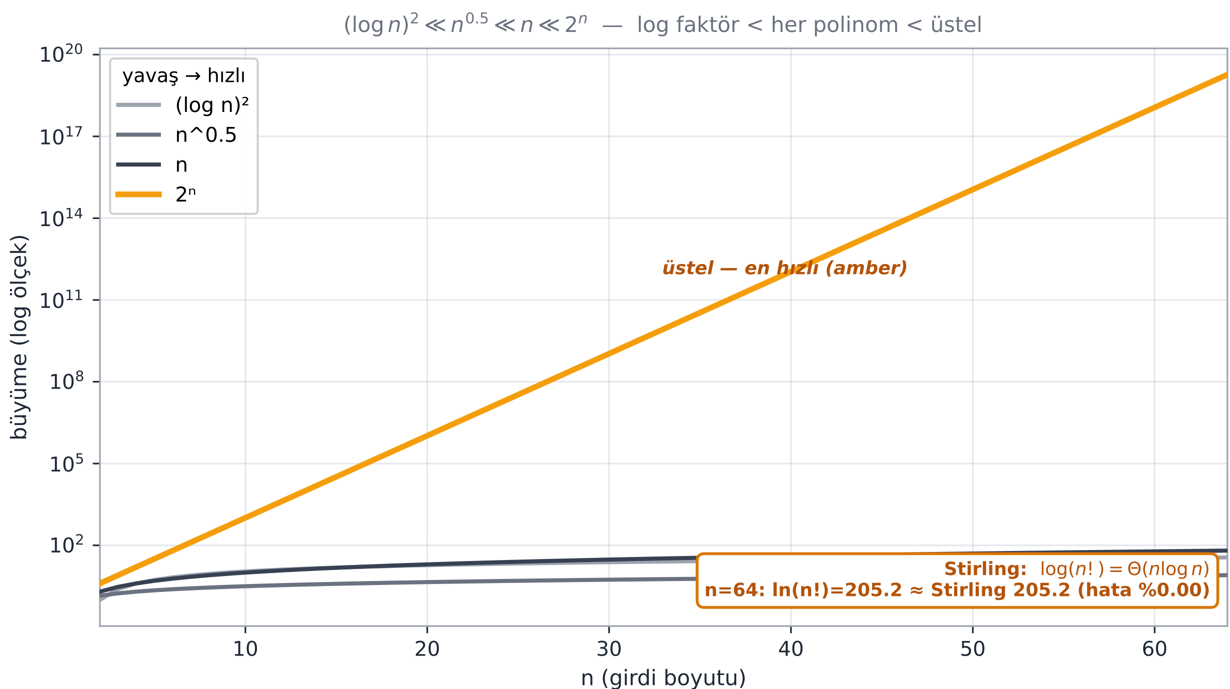
- **Log–polinom kimliği:** Herhangi pozitif a, b için $(\log n)^a \ll n^b$. Yani logaritmanın herhangi bir kuvveti, herhangi bir polinomdan kesinlikle (little-o anlamında) yavaştır. İspat yöntemi: iki fonksiyonun oranının limitini (kolaylık için logaritmasını) $n \to \infty$ için al.

  "this log n to any power is asymptotically less than any polynomial for any positive a and b." — Ku, 7:11

- **Stirling yaklaşımı:** $n! \approx \sqrt{2\pi n}\,(n/e)^n$. Bu, asimptotik değil *gerçek* bir limit eşitliğidir. Logaritması alınınca $\log(n!) = \Theta(n \log n)$ çıkar — sınıfta sık kullanılır.
- **Binom katsayısı:** $\binom{n}{k} = n!/(k!\,(n-k)!)$. İki örnek: $C(n,3) = n(n-1)(n-2)/6 = \Theta(n^3)$; $C(n, n/2)$, Stirling ile sadeleştirilince $\Theta(2^n/\sqrt{n})$ verir.

Bu araç kutusunun görsel özeti — kesin hiyerarşi ve Stirling'in $\log(n!)$ üzerindeki etkisi — Şekil 10.2 içinde gösterilir.

Karmaşıklık. Bu bir *analiz* problemidir (çalışma süresi değil): araç, fonksiyonları kapalı, karşılaştırılabilir forma indirgemek. Sonuç bir sıralamadır, bir koşma zamanı değil.



Şekil 10.2: Problem 1 araç kutusu: log-y ekseninde **kesin** asimptotik hiyerarşi $(\log n)^2 \ll n^{0.5} \ll n \ll 2^n$. Logaritmik faktörler (en açık slate) her polinomun ($n^{0.5}$, n — orta/koyu slate) altında kalır; polinomlar da üstelin (amber, **en hızlı**) altındadır — bu sıralama *her* pozitif a, b ve $c > 1$ için geçerlidir. Üstel eğri log eksende düz bir doğru olur ($\log 2^n = n \log 2$) ve n büyüdükçe diğerlerinin hepsini ezici biçimde geride bırakır; üç yavaş eğri tabanda kümelenir çünkü hepsi üstelin yanında ihmal edilebilir kalır. Sağ-alttaki kutu **Stirling** sonucunu özetler: $\log(n!) = \Theta(n \log n)$ — $\sqrt{2\pi n}\,(n/e)^n$ yaklaşımının logaritması $n \log n - n$ baskın terimini verir; $n = 64$ için $\ln(64!) = 205{,}17$ ile Stirling $205{,}17$ yalnız **%0,0006** sapar (asimptotik değil, gerçek bir limit eşitliği).

10.3 Problem 2: Sequence Arayüzünü Black Box Olarak Kullanma

İfade. Sana bir **sequence** veri yapısı veriliyor ve içini göremiyorsun (black box). Yalnızca dört işlemi var, hepsi sabit zaman: `insert_first`, `insert_last`, `delete_first`, `delete_last` (delete'ler sildikleri öğeyi döndürür). Bu işlemleri kullanarak (a) `swap_ends` ve (b) `shift_left(D, K)` yaz.

Yaklaşım — Arayüzü Onurlandır, Özyinelemeyle Değişmez Koru

İçeriği değil, yalnızca dış işlemleri kullan — bu, arayüz/gerçekleştirim ayrımının pratiğidir. Sabit-zamandan uzun işlemler için **döngü veya özyineleme** kullan; Ku özyinelemeyi tercih eder çünkü her adımda yalnızca *sabit miktarda* bilgiyle (bir değişmez + küçük durum analizi) uğraşmak argümanı kolaylaştırır.

> *"if I break it down so that I solve a slightly smaller problem recursively, and then do a constant amount of work and maintain some invariant, then it's very easy to argue things about it."* — Ku, 40:03

Çözüm.

(a) **`swap_ends`** — ilk ve son öğeyi değiştir: her ikisini sil, geçici değişkenlerde tut, ters yerlere geri ekle.

```
swap_ends(D):
    x1 = D.delete_first()
    x2 = D.delete_last()
    D.insert_first(x2)
    D.insert_last(x1)
```

(b) **`shift_left(D, K)`** — ilk K öğeyi sona taşı (K'inci öğe son olur, $K+1$'inci ilk olur). Özyinelemeli:

```
shift_left(D, K):
    if K < 1 or K > len(D) - 1:   # taban durum / sınır kontrolü
        return
    x = D.delete_first()
    D.insert_last(x)
    shift_left(D, K - 1)          # bir küçük örnek
```

Her çağrı bir öğeyi öne alır ve K'yi 1 azaltır; taban durumda ($K < 1$) durur. Değişmez korunduğu için doğruluk kolayca tümevarımla görülür. Bu black-box disiplini, bir sonraki problemde (Problem 3) tam tersini yapmanın — yani arayüzün *altındaki* veri yapısını tasarlamanın — neden değer taşıdığını da gösterir.

Karmaşıklık. `swap_ends` sabit sayıda $O(1)$ işlem → $O(1)$. `shift_left`, K kez sabit iş → $O(K)$.

10.4 Problem 3: Çift Uçlu Dinamik Dizi

İfade. "İki dünyanın en iyisi" bir veri yapısı tasarla: en kötü durumda $O(1)$ **indeksleme** (dizi gibi) *ve* sequence'in **her iki ucunda** $O(1)$ **amortize** ekleme/silme. Tek başına dinamik dizi sonda amortize $O(1)$ ama başta $O(n)$'dir; bağlı liste iki uçta $O(1)$ ama indeksleme $O(n)$'dir.

Yaklaşım — Sıfırdan Tasarla ya da Hazıra İndirge

İki yol vardır: (1) dinamik diziyi sıfırdan yeniden tasarla (çift uçlu); (2) hazır bir dinamik diziye **indirgeme** (reduction) yap. Her ikisinin de çekirdeği aynı sezgidir: pahalı bir yeniden-inşadan önce

mutlaka lineer sayıda ucuz işlem garanti et — bu, **charging argument**'ın özüdür.

Çözüm.

Yöntem 1 — Çift uçlu dinamik dizi (dynamic deque). Diziyi her *iki* uçta da fazladan boş yerle (her birinde lineer miktarda) tut. Böylece hem önden hem sondan eklerken, lineer sayıda işlem yapana kadar yeniden boyutlandırma gerekmez. Yeniden boyutlandırırken (büyütme veya küçültme) her iki uçta yine lineer fazla yer bırak — bu, charging argument'ın çalışmasını garanti eder: bir pahalı yeniden-inşadan önce mutlaka lineer sayıda ucuz işlem yapılmış olur (silmede de israfı önlemek için $\frac{1}{4}$ doluluğa inince küçült).

> *"I have to do a linear number of cheap things before I have to do an expensive thing again."* — Ku, 1:00:09

Yöntem 2 — Dinamik diziye indirgeme. İki dinamik dizi kullan; birini **ters** yönde gör. Saklanacak sequence'i ortadan ikiye böl; her yarı bir dinamik dizidir (biri normal, biri ters). Önden/sondan işlemler artık birer dinamik dizi ucu işlemi olur (biraz indeks aritmetiğiyle). Tek incelik: bir dizi **boşalırsa**, diğerini ikiye böl ve iki yeni diziye yeniden dağıt — bu yeniden-inşanın maliyeti, biriken amortize bütçeden karşılanır.

Karmaşıklık. İndeksleme $O(1)$ **en kötü durum** (saf ofset aritmetiği); her iki uçta ekleme/silme $O(1)$ **amortize**; alan, saklanan öğe sayısında lineer ($O(n)$).

10.5 Problem 4: Bağlı Listenin Son Yarısını Ters Çevirme

İfade. $2n$ düğümlü tek yönlü (singly) bir bağlı liste verilir. Son n düğümün sırasını **yerinde** (in-place) ters çevir. Yeni düğüm oluşturma yok; sabit (constant) ek alandan fazlasını kullanma — yani öğeleri bir diziye kopyalayıp geri yazamazsın, sadece mevcut düğümlerin işaretçilerini değiştirebilirsin.

Yaklaşım — Üç Adıma Böl, Önceki Düğümü Hatırla

Üç adıma böl: (1) n'inci düğümü bul, (2) $n+1$'den $2n$'e kadar olan düğümlerin `next` işaretçilerini ters çevir, (3) uçları yeniden bağla. İşaretçi ters çevirirken, "önceki" düğümü hatırlamak gerekir; aksi halde liste kopar ve düğümlere erişim kaybolur (garbage-collected dillerde sızıntı, diğerlerinde memory leak). Adım-adım işaretçi izi Şekil 10.3 içinde gösterilir.

Çözüm.

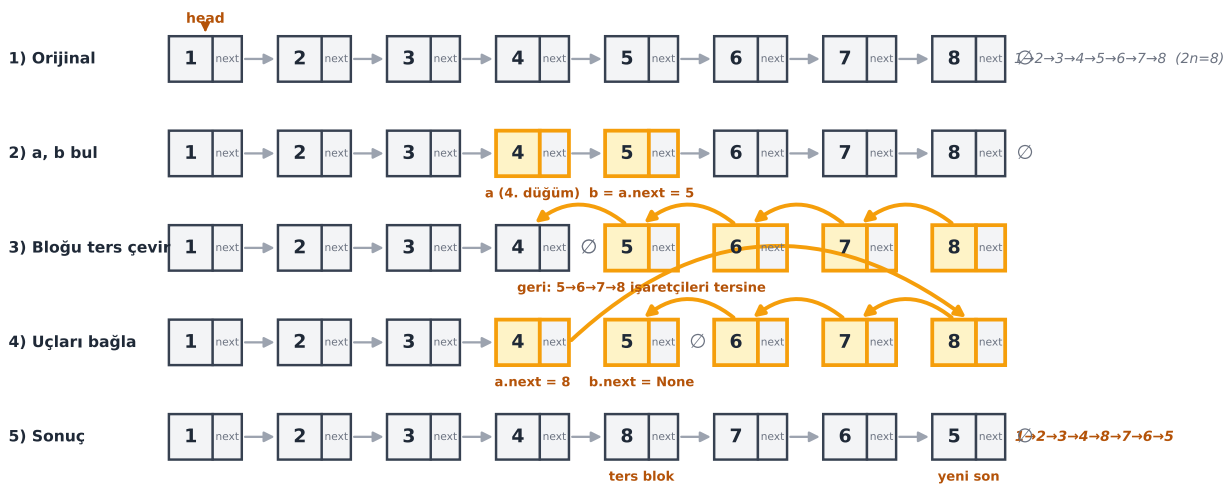
1. **n'inci düğümü bul:** $n = \text{size}/2$. Baştan $n-1$ kez `next` takip et → `a` = n'inci düğüm. `b = a.next` (ters çevrilecek bloğun başı).
2. **İşaretçileri ters çevir:** İki imleç tut — `x` (yeniden bağlanacak düğüm) ve `x_p` (ondan önceki). `x_p, x = a, b` ile başla. n kez: `x_n = x.next` (sonrakini sakla), `x.next = x_p` (geriye bağla), sonra `x_p, x = x, x_n` (kaydır).
3. **Uçları temizle:** Döngü sonunda `x_p` ters bloğun yeni başı (`c`) olur. `a.next = c` (ilk blok yeni başa bağlanır), `b.next = None` (eski baş, artık son, sonlanır).

```
reorder(L):
    n = L.size // 2
    a = L.head
    for _ in range(n - 1):
        a = a.next
    b = a.next
    x_p, x = a, b
    for _ in range(n):
        x_n = x.next
        x.next = x_p
        x_p, x = x, x_n
    a.next = x_p      # x_p = c
    b.next = None
```

Şekil 10.3'in beş aşaması, bu pseudocode'un $2n = 8$ düğümlük bir liste üzerinde nasıl yürüdüğünü — pivot a'nın bulunması, bloğun ters çevrilmesi ve uçların yeniden bağlanması — somut olarak izler.

Karmaşıklık. n'inci düğümü bulma $O(n)$, ters çevirme $O(n)$, temizlik $O(1)$ → toplam $O(n)$; ek alan $O(1)$ (yalnızca birkaç imleç).

Son yarıyı yerinde ters çevirme: 2n=8 düğüm, son n=4 (amber = ters blok işaretçileri)



Şekil 10.3: Son yarıyı **yerinde** ters çevirme izi ($2n = 8$ düğüm, son $n = 4$). **(1)** Orijinal liste $1 \to \cdots \to 8$, `head`. **(2)** $a = 4$. düğüm (pivot), $b = a.\text{next} = 5$ (ters çevrilecek bloğun başı). **(3)** Bloğun (5,6,7,8) `next` işaretçileri tersine çevrilir — **amber** geri-oklar artık $8 \to 7 \to 6 \to 5 \to 4$ yönünde; a'nın `next`'i geçici olarak $\emptyset$. **(4)** Uçlar yeniden bağlanır: `a.next = 8` ve `b.next = None` (eski baş artık son). **(5)** Sonuç $1 \to 2 \to 3 \to 4 \to 8 \to 7 \to 6 \to 5$ — ek alan $O(1)$ (yalnız birkaç imleç), zaman $O(n)$. İmleç ters çevrilirken "önceki" düğümü hatırlamak şarttır, yoksa liste kopar.

Builder Notu — Yerinde Algoritmalar ve Memory Leak

İşaretçi ters çevirmede bir düğüme referans kaybedersen, garbage-collected bir dilde (Python) GC onu toplar; C gibi dillerde bu bir **memory leak**'tir. Yerinde (in-place) algoritmalar, gömülü sistemler ve düşük-bellek ortamlarında kritiktir: ekstra dizi ayırmaya bütçe yoktur.

- **İleriye → ML / tensör işlemleri:** PyTorch'taki _ son ekli işlemler (`add_`, `relu_`) tam olarak bu sezgidir — yeni tensör ayırmadan mevcut belleği yerinde değiştirmek; büyük modellerde bellek tasarrufunun anahtarı.
- **İleriye → embedded / firmware:** Mikrodenetleyicilerde yığın (heap) ayırma çoğu zaman yasaktır; veri yapıları yerinde dönüştürülür.

Tek cümle: *Yerinde dönüşüm, "yeni alan ayırma — var olanı yeniden bağla" disiplinidir.*

10.6 Ne Öğrendik?

Altı Taşınabilir Araç

Bu oturum, Ders 1-2'nin teorisini dört somut problemde uyguladı ve altı taşınabilir araç kazandırdı:

1. **Asimptotik karşılaştırma araçları:** $(\log n)^a \ll n^b \ll c^n$ hiyerarşisi; faktöriyel/binom için Stirling yaklaşımı ($\log(n!) = \Theta(n \log n)$).
2. **Black box / arayüz disiplini:** Bir veri yapısını yalnızca dış işlemleriyle kullanıp üzerine yeni işlemler kurmak.
3. **Özyineleme stratejisi:** Sabit iş + bir küçük örneğe indirgeme; değişmezle kolay doğruluk argümanı.
4. **Charging argument (amortize):** Pahalı bir işlemden önce lineer sayıda ucuz işlem garanti ederek $O(1)$ amortize elde etme.
5. **Yerinde işaretçi manipülasyonu:** Sabit ek alanla bağlı liste yeniden bağlama; referans kaybı = sızıntı.
6. **İletişim kuralı:** Algoritmayı koddan önce *kelimelerle* anlat.

Dört problemin zaman, ek alan ve anahtar sonuç açısından maliyet özeti Şekil 10.4 içinde tek bakışta toplanır.

10.7 Sonraki

Ders 4 (L3) — Kümeler ve Sıralama

Sırada **Ders 4 (L3): Kümeler ve Sıralama** var — Justin Solomon ile **küme (set)** arayüzüne ve onu çözmenin ilk adımı olan **sıralamaya** (insertion, selection, merge sort) geçiyoruz. Bu oturumda gördüğün **asimptotik karşılaştırma araçları**, sıralama algoritmalarının çalışma sürelerini analiz ederken doğrudan işine yarayacak.

PS1 dört problemin maliyet özeti: bir analiz aracı + üç veri-yapısı garantisi

Problem	Zaman	Ek Alan	Anahtar Sonuç
P1 · Asimptotik Sıralama	**— (analiz)**	**—**	$(\log n)^a \ll n^b \ll c^n$, $\log(n!)=\Theta(n \log n)$
P2 · Sequence Black Box	**swap_ends O(1)** **shift_left O(K)**	**O(1)**	yalnız 4 uç işlemi · özyineleme
P3 · Çift Uçlu Dinamik Dizi	**index O(1) en-kötü** **iki-uç O(1) amortize**	**O(n)**	ofset aritmetiği + charging
P4 · Son Yarıyı Yerinde Ters	**O(n)**	**O(1) (yerinde)**	3 imleç · yeni düğüm yok

O(1) — sabit (en iyi) · O(1) amortize / yerinde — neredeyse en iyi · O(n) / O(K) — girdiye bağlı · analiz / yok

Şekil 10.4: PS1'in dört probleminin maliyet özeti tek bakışta. Satırlar dört problem, sütunlar **Zaman · Ek Alan · Anahtar Sonuç**. Hücreler maliyet sınıfına göre renklenir: yeşilimsi hücreler $O(1)$ sabit (en iyi), amber-çerçeveli açık hücreler $O(1)$ amortize / yerinde (neredeyse en iyi), dolu amber hücreler $O(n)$ / $O(K)$ girdiye bağlı, nötr hücreler ise çalışma-süresi olmayan analiz. **P1** bir araç-kutusudur (çalışma süresi yok): asimptotik hiyerarşi $(\log n)^a \ll n^b \ll c^n$ ve $\log(n!) = \Theta(n \log n)$. **P2** yalnız dört $O(1)$ uç işlemiyle `swap_ends`'i $O(1)$, `shift_left`'i özyinelemeyle $O(K)$ kurar — $O(1)$ ek alan. **P3** ofset aritmetiğiyle $O(1)$ **en-kötü** indeksleme ile charging argument'a dayanan iki-uçta $O(1)$ **amortize** ekleme/silmeyi birleştirir (alan $O(n)$). **P4** son yarıyı yalnız üç imleçle, yeni düğüm üretmeden çevirir: zaman $O(n)$, ek alan $O(1)$ **yerinde**. Amber çerçeve, amortize ve yerinde kazanımları işaretler.

11 Kümeler ve Sıralama

Küme arayüzü, sırasız/sıralı dizi tradeoff'u, permutation/selection/insertion/merge sort ve substitution analizi

Bölüm bilgisi

- **Solomon'un videosu:** YouTube — Lecture 3: Sets and Sorting (≈53 dk)
- **OCW sayfası:** MIT 6.006 Lecture 3: Sets and Sorting
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 4 (L3)
- **Hoca:** Justin Solomon (Jason Ku, Erik Demaine)
- **Okuma süresi:** ≈25 dk

11.1 Bu Derste Ne Var?

Ders 2 **dizi (sequence)** arayüzünü çözdü — öğeleri *sıraya* göre tutmak. Bu ders ikinci temel arayüze geçer: **küme (set)** — öğeleri *anahtarına* göre tutmak. Justin Solomon, set'i çözmenin doğal yolunun **sıralama** olduğunu gösterir ve dersin yarısını sıralama algoritmalarına ayırır.

Üç temel kavram bu derste yan yana gelir:

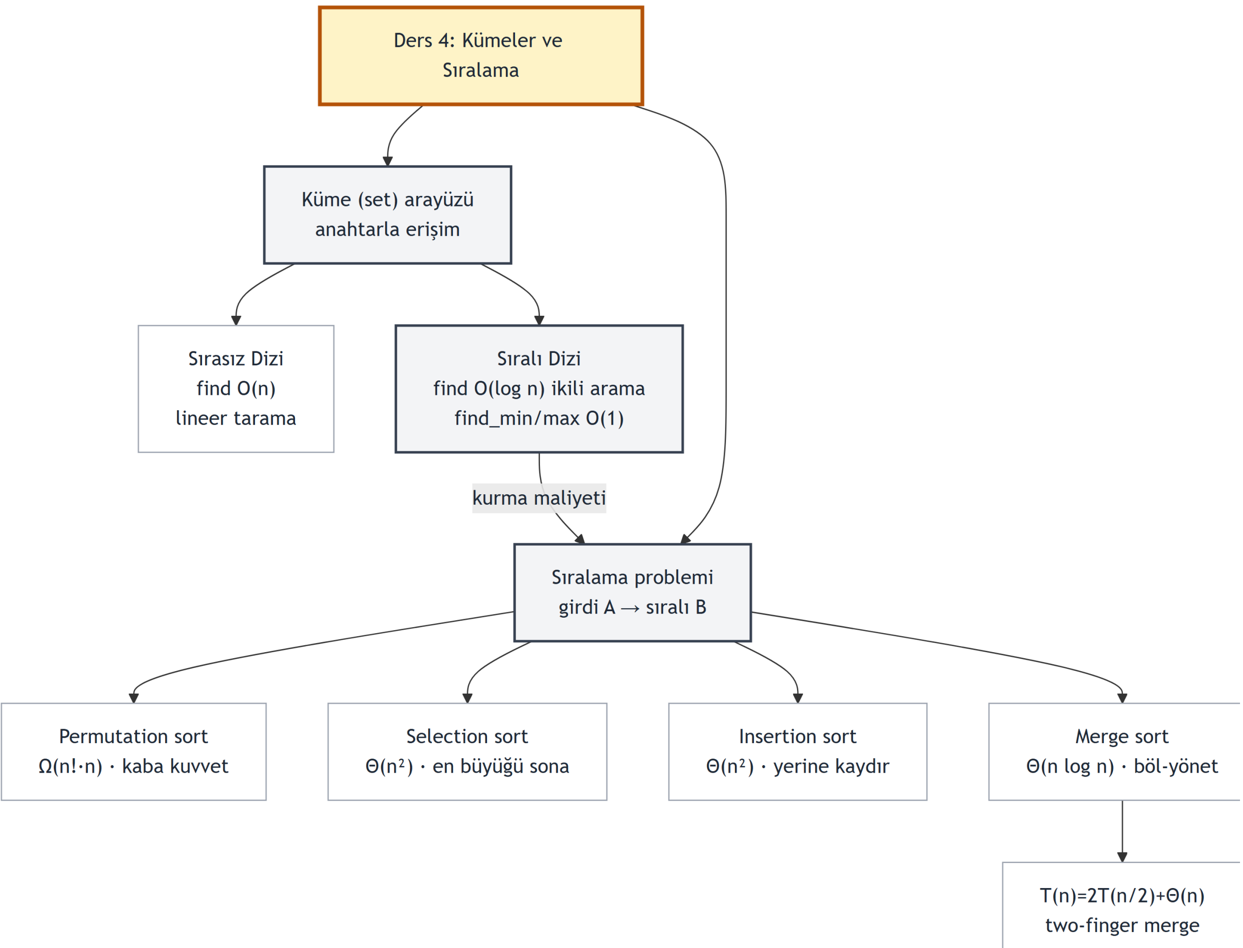
1. **Küme arayüzü** — anahtarla (key) erişilen bir koleksiyon: find, insert, delete, find_min/max.
2. **Sırasız vs sıralı dizi tradeoff'u** — sıralı dizide arama $O(\log n)$ ama sıralamanın kendisi $O(n \log n)$ maliyetlidir.
3. **Sıralama algoritmaları** — permutation, selection, insertion ve merge sort; özyineleme + substitution ile çalışma süresi analizi.

> *"there's an object called an interface, which is just a program specification."* — Justin, 1:42

Builder Notu — Set Arayüzü ve Python dict/Veritabanı

Bu dersin **küme (set)** arayüzü, her veri katmanının altında yatan "anahtarla eriş" desenidir — Python'dan veritabanlarına:

- **Geriye → Python (Phase 1):** Python `set/dict` tam da bu set arayüzünün gerçekleştirimidir; Solomon "bu çirkin detayları sizin için hallederler" der. `key in d` ortalama $O(1)$ görünür, ama o sabit zamanın *nasıl* elde edildiği (Ders 4 / Hashing) burada başlayan sorudur.
- **Geriye → Calculus (Phase 1):** n^2 ile $n \log n$ büyüme farkı — bir milyar öğede biri saatler, diğeri saniyeler. Asimptotik mertebeleri karşılaştırma disiplini doğrudan fonksiyon büyümesi sezgisidir.



Şekil 11.1: Ders 4'ün kavram haritası: küme arayüzünden iki dizi gerçekleştirimine (sırasız dizi → find $O(n)$ / sıralı dizi → find $O(\log n)$), oradan sıralama problemine ve dört sıralama algoritmasına (permutation $\Omega(n!\cdot n)$, selection $\Theta(n^2)$, insertion $\Theta(n^2)$, merge $\Theta(n \log n)$).

- **İleriye → veritabanı/arama:** sıralı/indeksli veri, ikili aramayla $O(\log n)$ sorgu; her veritabanı indeksinin (B-tree) temeli.
- **İleriye → OMSCS / böl-yönet:** merge sort, "böl ve yönet + substitution" kalıbının ilk örneği; CS 6515'te (Graduate Algorithms) her yerde.

Tek cümle: *Bir kümeyi sıralı tutmak, aramayı $O(\log n)$'e indirir — ama bu hızın bedeli, baştaki $O(n \log n)$'lik sıralamadır.*

11.2 Arayüz Tekrarı: Küme Nedir?

Ders 1'de tanıtılan ayrım burada da merkezdedir: **arayüz** bir *program spesifikasyonudur* (hangi işlemler destekleniyor); **veri yapısı** o arayüzü perde arkasında *gerçekleştiren* somut yapıdır.

> *"there's an object called an interface, which is just a program specification… a data structure, which is a way to actually implement an interface."* — Justin, 1:42

Bir **küme (set)** "bir yığın şey"dir: içine öğe eklersin ve "bu öğe var mı?" diye sorarsın. Her öğe bir **anahtarla (key)** ilişkilidir — örneğin sınıftaki öğrencilerin ID numarası. Aramayı anahtarla yaparsın; bulunca öğenin geri kalan bilgisine (isim, vb.) erişirsin.

11.3 Küme (Set) Arayüzü

Set arayüzü şu işlemleri destekler:

- **build(A):** bir iterable `A`'dan küme kur.
- **len():** içindeki öğe sayısı.
- **find(k):** anahtarı `k` olan öğeyi döndür (yoksa null).
- **insert(x) / delete(k):** dinamik işlemler — kümeyi değiştirir.
- **find_min() / find_max():** en küçük / en büyük anahtarlı öğe.
- **find_prev(k) / find_next(k):** sıralı komşular.

Burada **x** tüm nesneyi (ID + isim + …), **k** ise aramada kullanılan anahtarı temsil eder. Bu bir *arayüzdür* — nasıl gerçekleştirildiği henüz söylenmedi.

11.4 Küme mi, Dizi mi?

İki temel arayüzün farkı, öğeleri *neye göre* tuttuğudur:

- **Dizi (sequence):** öğeler **dışsal (extrinsic)** bir sıraya göre — ilk, onuncu, son. Pozisyonla erişim.
- **Küme (set):** öğeler **içsel (intrinsic)** anahtarlarına göre. Değerle (key) erişim.

Aynı veri her iki arayüzle de tutulabilir, ama sorular farklıdır: dizi "5. öğe ne?" diye sorar, küme "anahtarı 42 olan öğe var mı?" diye sorar.

11.5 Sırasız Dizi ile Küme

En basit gerçekleştirim: öğeleri **sırasız bir dizide** (unordered array), gelişigüzel sıralamayla sakla. Kurması kolaydır — büyük bir dizi al, her şeyi içine at.

> *"an unordered array is a perfectly reasonable way to implement this set interface. And then searching that array it will take linear time"* — Justin, 12:00

Ama her arama, diziyi baştan sona taramayı gerektirir: `find(k)` en kötü durumda $O(n)$. Aslında tüm işlemler (build, insert, delete, find_min) lineer zamandır. Basit ama yavaş. Şekil 11.2 bu sırasız gerçekleştirimi, hemen ardından gelen sıralı alternatifle yan yana gösterir.

11.6 Sıralı Dizi ile Küme

Daha iyisi: öğeleri **anahtara göre sıralı** bir dizide tut. Artık:

- **find_min() / find_max():** ilk / son öğe → $O(1)$.
- **find(k): ikili arama (binary search)** ile diziyi her adımda yarıya böl → $O(\log n)$.

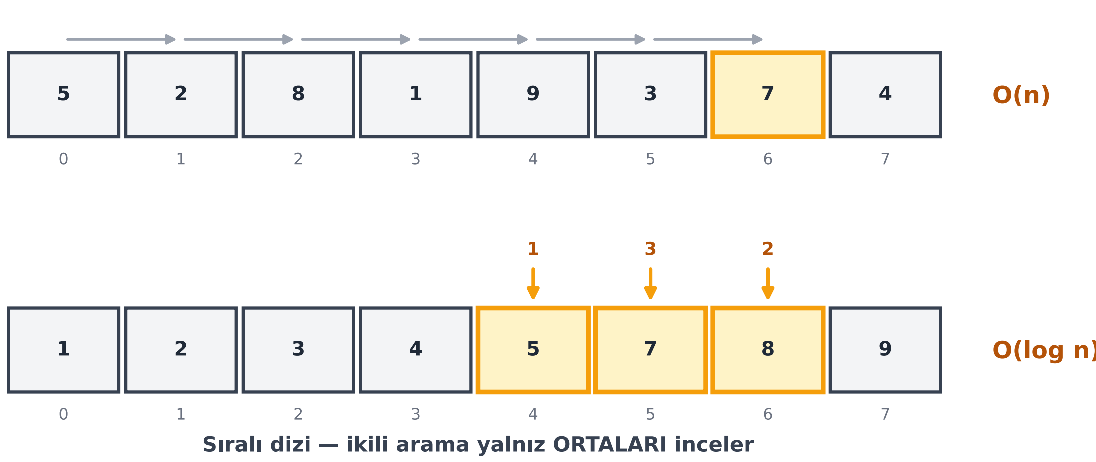
> *"if my array is sorted, how long does it take for me to search for any given element? … Log n time."* — Justin, 16:24

Bir milyar öğede sırasız dizi ~bir milyar işlem, sıralı dizi yalnızca ~30 işlem ($\log_2 10^9 \approx 30$) ister. Ama bir bedel var: sıralı diziyi **elde etmek** = sıralama, ve sıralama $O(n \log n)$ zaman alır. Dersin geri kalanı bu sıralamayı çözer. Şekil 11.2 iki gerçekleştirimi karşılaştırır; Şekil 11.3 ise ikili aramanın yarıya-bölme mekaniğini tek tek izler.

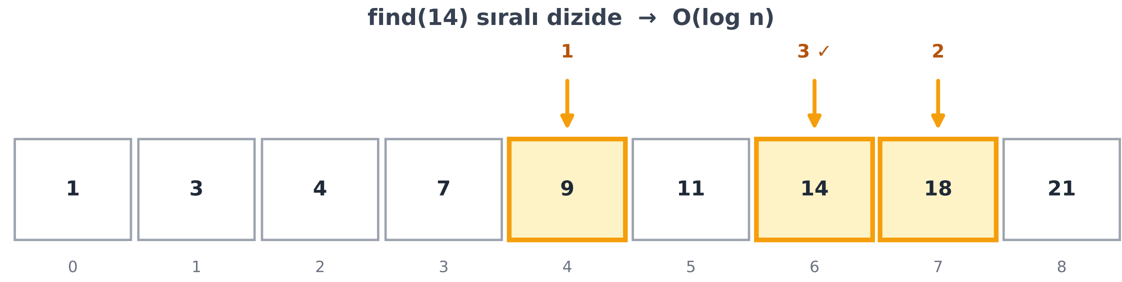
Builder Notu — O(log n) ve Veritabanı İndeksleri (B-tree)

"Sıralı dizi + ikili arama = $O(\log n)$ arama" sezgisi, her ölçeklenebilir veri sisteminin kalbindedir:

- **İleriye → veritabanı indeksleri:** Bir tabloya indeks eklemek, anahtarları **sıralı** bir yapıda (B-tree, sıralı dizinin disk-dostu çok-yollu genellemesi) tutmak demektir; `WHERE id = 42` sorgusu tam tamına ikili aramadır → milyarlarca satırda ~30 disk erişimi. İndekssiz sorgu ise sırasız diziye eşdeğer: tam tablo taraması, $O(n)$.
- **İleriye → arama motoru posting list'leri:** Ters indekslerdeki belge ID listeleri sıralı tutulur; iki listenin kesişimi (AND sorgusu) tam da two-finger merge mantığıyla, sıralılıktan faydalanarak yapılır.
- **Bedel-fayda dengesi:** İndeks "ücretsiz" değildir — onu *kurmak* ve *güncel tutmak* (her insert'te sıralı yapıyı koruma) maliyettir, tıpkı sıralı diziyi kurmanın $O(n \log n)$ bedeli gibi. "Çok okunan, az yazılan" veride indeks kazandırır.



Şekil 11.2: Küme arayüzünü diziyle gerçekleştirmenin iki yolu ve **arama** maliyetinin ayrışması. Üstte **sırasız dizi**: `find(7)` öğeyi bulana dek baştan sona tarar (soluk slate okları), öğe nerede olursa olsun en kötü durumda her hücreye bakılır → $O(n)$; `find_min/max` de aynı şekilde tüm diziyi tarar. Altta **aynı veri sıralı**: `find(7)` **ikili aramayla** yalnızca ortaları inceler (amber, 1-2-3 sırasıyla: index 4 → 6 → 5), her adım aralığı yarıya böler → $O(\log n)$; üstelik `find_min` = ilk öğe ve `find_max` = son öğe **uçlardan** doğrudan okunur → $O(1)$. Bedel: sıralı diziyi *kurmak* bir kez $O(n \log n)$ sıralama gerektirir — ama sonraki her arama logaritmik kalır (Şekil 11.3 bunu ayrıntılandırır).



Şekil 11.3: Sıralı dizide **ikili arama** (binary search): aralık $[lo, hi)$ her adımda **ortadan** ikiye bölünür. Ortadaki öğe ($A[mid]$) hedefle karşılaştırılır; küçükse sağ yarıya, büyükse sol yarıya inilir, eşitse bulunur. Burada `find(14)` üç adımda çözülür: 1. adım indeks 4 (değer $9 < 14 \rightarrow$ sağa), 2. adım indeks 7 (değer $18 > 14 \rightarrow$ sola), 3. adım indeks 6 (değer 14 ✓, bulundu). İncelenen ortalar **amber** vurgulu ve sıra-numaralı; elenen yarılar soluklaşır. Her adım arama uzayını yarıladığından maliyet $O(\log n)$'dir — $n = 10^9$ için bile yalnızca $\log_2 10^9 \approx 30$ adım yeter (sırasız dizide aynı arama $O(n)$, yani milyarlarca işlem olurdu).

11.7 Sıralama Problemi ve Sözlük

Sıralama problemi: girdi, n anahtardan oluşan bir dizi `A`; çıktı, aynı öğelerin sıralı bir dizisi `B`. İki terim algoritmaları ayırır:

- **Yıkıcı (destructive):** yeni bir `B` dizisi ayırmak yerine, `A`'nın üzerine sıralı hâlini yazar (C++ ve Python sort'u böyledir).
- **Yerinde (in-place):** yıkıcı *ve* ek bellek kullanmaz (sabit $O(1)$ fazlası dışında — döngü sayaçları gibi).

> *"a destructive algorithm is one that just overwrites A with a sorted version of A… in place, meaning that they also don't use extra memory"* — Justin, 20:13

11.8 Permütasyon Sıralaması

En basit (ve en kötü) sıralama: bir listenin sıralı hâli, onun bir permütasyonudur. O hâlde **tüm permütasyonları listele**, her birinin sıralı olup olmadığını kontrol et.

> *"list every possible permutation, and then just double check which one's in the right order."* — Justin, 22:04

n öğenin $n!$ permütasyonu vardır; her birinin sıralılığını kontrol etmek $\Theta(n)$'dir. Toplam $\Omega(n! \cdot n)$ — $n!$'den bile kötü. (Burada Ω kullanılır çünkü permütasyonları üretmenin maliyeti *en az* bu kadardır — bir alt sınır.) Pratikte tamamen kullanılamaz; ama doğruluğu ispatlaması çok kolaydır.

11.9 Seçmeli Sıralama (Selection Sort)

Daha akıllı bir algoritma. 6.006'da onu — alışılmadık biçimde — **özyinelemeli** yazarız (doğruluk ve süre analizini kolaylaştırmak için; pratikte iki döngüyle yazılır).

Önce bir yardımcı: `prefix_max(A, i)` — `A`'nın 0'dan `i`'ye kadarki en büyük öğesinin indeksini bulur. Solomon'un "derin gözlemi": 0..i aralığındaki en büyük öğe ya **tam i'dedir** ya da **i'den önceki** bir indekstedir.

> *"the biggest element from 0 to i… Either it's at index i… or it has index less than i. … Another term for this is induction."* — Justin, 31:15

```
def prefix_max(A, i):
    if i == 0:
        return 0
    j = prefix_max(A, i - 1)      # 0..i-1'in en buyugu (tumevarim hipotezi)
    if A[i] > A[j]:
        return i
    return j
```

Seçmeli sıralama: `i = n-1`'den başla; 0..i'nin en büyüğünü `prefix_max` ile bul, indeks `i`'ye **swap** et, sonra 0..i−1'i özyinelemeli sırala (somut bir iz için Şekil 11.4).

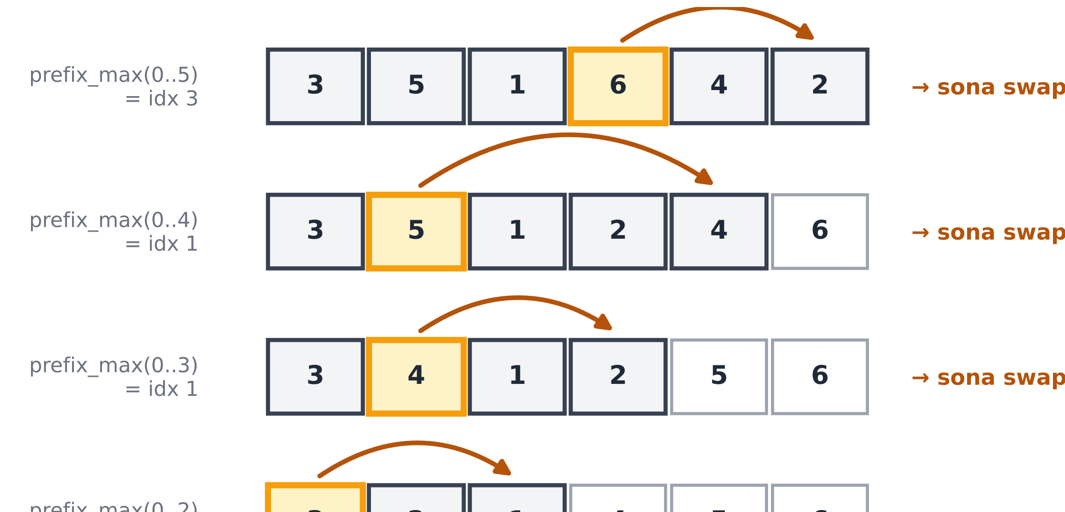
```
def selection_sort(A, i):
    if i > 0:
        j = prefix_max(A, i)
        A[i], A[j] = A[j], A[i]   # en buyugu sona koy
        selection_sort(A, i - 1)  # solu sirala
```

Çalışılan Örnek — substitution ile çalışma süresi. `prefix_max` için: $S(n) = S(n-1) + \Theta(1)$. $S(n) = cn$ tahmin et, yerine koy: $cn = c(n-1) + \Theta(1) \to \Theta(1) = c$, sabit. Yani $S(n) = \Theta(n)$. Seçmeli sıralama için: $T(n) = T(n-1) + \Theta(n)$ (her çağrı bir `prefix_max` = $\Theta(n)$). $T(n) = cn^2$ tahmin et: $cn^2 = c(n-1)^2 + \Theta(n) = cn^2 - 2cn + c + \Theta(n) \to \Theta(n) = 2cn - c$, tutarlı. Yani $T(n) = \Theta(n^2)$. Şekil 11.6 bu n seviyeli yineleme ağacını, merge sort'un $\log n$ seviyesiyle yan yana koyar.

Builder Notu — Yineleme Analizi Disiplini

`prefix_max` ve `selection_sort`'taki "form tahmin et → yerine koy → doğrula" (substitution) hareketi, her özyinelemeli maliyeti kapalı bir formüle bağlamanın temel disiplinidir:

- **İleriye → ML kernel maliyetleri:** Bir transformer'da self-attention'ın $O(n^2)$ maliyeti (her token diğer her token'a bakar) tam bu tür bir sayımla türetilir — ve neden uzun bağlamın pahalı olduğunun, FlashAttention/lineer-attention gibi optimizasyonların *neden* gerektiğinin cevabıdır.
- **İleriye → OMSCS CS 6515:** Graduate algoritma derslerinde her dinamik programlama / böl-yönet çözümünün ardından bir yineleme bağıntısı yazılır ve substitution (veya master teoremi) ile çözülür; bu dersin alıştırması oraya doğrudan taşınır.
- **Sezgi:** Selection sort'un $T(n) = T(n-1) + \Theta(n)$'i, "her adım problemi yalnız bir azaltır" demektir; bu yüzden $n + (n-1) + \cdots + 1 = \Theta(n^2)$. Yineleme bağıntısı, performansın *neden*ini koddan önce gösterir.



Şekil 11.4: Seçmeli sıralama izi (Solomon §8): dizi **sağdan sola** sıralanır. Her dış adımda `prefix_max(0..i)` aralığın **en büyük** öğesini bulur (amber vurgulu) ve onu pozisyon i'ye **swap** eder (amber yay) — böylece sağ uçta giderek büyüyen bir **sıralı kuyruk** (soluk hücreler) oluşur. Burada $[3, 5, 1, 6, 4, 2]$ dört geçişte tamamen sıralanır: $6 \rightarrow 5 \rightarrow 4 \rightarrow 3$ sırayla sona yerleşir, en altta sonuç $[1, 2, 3, 4, 5, 6]$. n adımın her biri $\Theta(n)$'lik bir `prefix_max` taraması içerir → $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

11.10 Eklemeli Sıralama (Insertion Sort)

Seçmeli sıralamanın "tersi": önce 0..i−1'i sırala, sonra `i`'inci öğeyi sıralı kısma doğru yerine **kaydır**. O da özyinelemeli yazılır ve yine $\Theta(n^2)$'dir (her ekleme en kötü durumda lineer kaydırma). Solomon zaman darlığından detayını atlar; argüman seçmeli sıralamayla aynıdır.

11.11 Birleştirmeli Sıralama (Merge Sort)

İlk *gerçekten verimli* sıralama: **böl ve yönet**. Diziyi ortadan ikiye böl, her yarıyı özyinelemeli sırala, sonra iki sıralı yarıyı **birleştir (merge)**.

Birleştirmenin püf noktası **iki parmak (two-finger)** tekniğidir: her sıralı listenin sonuna birer parmak koy, büyük olanı sonuca al, o parmağı sola kaydır. İki yarı zaten sıralı olduğundan, parmağın soluna hiç bakmana gerek kalmaz → birleştirme $\Theta(n)$ (lineer).

> *"I'm going to take two sorted lists. And I'm going to make a new sorted list, which is twice as long, by using two fingers… it takes linear time to merge."* — Justin, 44:40

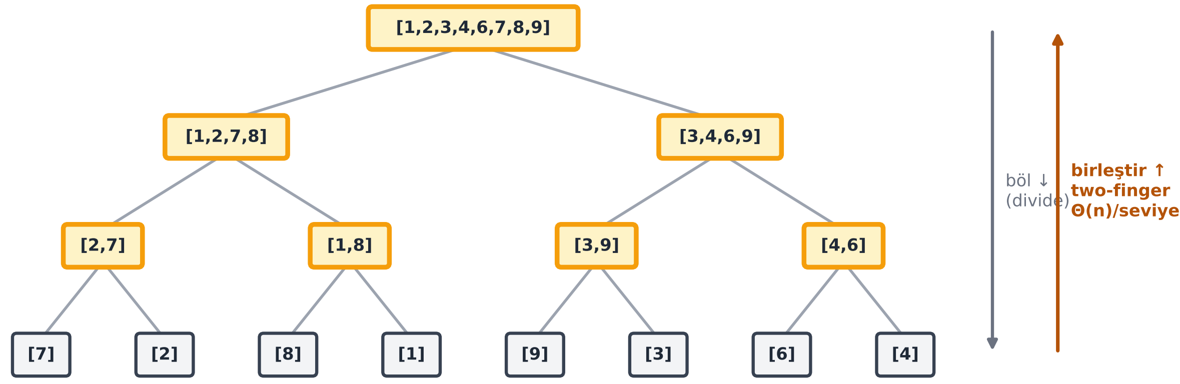
```
def merge_sort(A, a, b):            # A[a:b] araligini sirala
    if b - a > 1:
        c = (a + b) // 2            # orta (center)
        merge_sort(A, a, c)         # solu sirala
        merge_sort(A, c, b)         # sagi sirala
        merge(A, a, c, b)           # iki sirali yariyi birlestir (two-finger)
```

Çalışılan Örnek — substitution ile çalışma süresi. Merge sort iki kez yarı boyutta çağrılır + $\Theta(n)$ birleştirme: $T(n) = 2 \cdot T(n/2) + \Theta(n)$. $T(n) = cn \log n$ tahmin et, yerine koy: $cn \log n = 2 \cdot c(n/2) \cdot \log(n/2) + \Theta(n) = cn(\log n - \log 2) + \Theta(n) = cn \log n - cn \log 2 + \Theta(n)$. İki taraftan $cn \log n$ gider; $\Theta(n) = cn \log 2$, tutarlı ($\log 2$ sabit). Yani $T(n) = \Theta(n \log n)$ — sıralı diziyi vaat edilen sürede elde ettik. Şekil 11.5 bu böl-yönet akışını ve tek bir two-finger merge adımını birlikte gösterir.

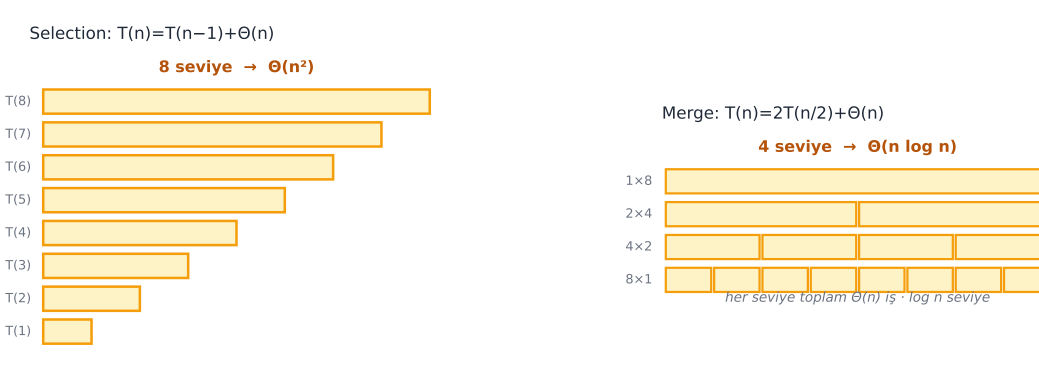
Builder Notu — Böl-Yönet ve OMSCS / quicksort-FFT

Merge sort'un $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ yinelemesi, *tüm* verimli özyinelemeli algoritmaların habercisidir:

- **İleriye → OMSCS CS 6515 çekirdek paradigması:** Graduate algoritmada **böl-ve-yönet** ayrı bir ünitedir; quicksort, ikili arama, en yakın nokta çifti ve hepsi aynı "böl → özyinele → birleştir" kalıbını ve master teoremini paylaşır.
- **İleriye → FFT ($\Theta(n \log n)$):** Hızlı Fourier Dönüşümü, polinomu çift/tek katsayılara böler, her yarıyı özyinelemeli dönüştürür, sonra birleştirir — merge sort'un birebir aynı yineleme ağacı. ML'de hızlı evrişim (convolution) ve sinyal işleme buna dayanır.
- **Pratik nüans:** Merge sort kararlıdır ama $\Theta(n)$ ek alan ister; quicksort yerinde ve genelde daha hızlıdır ama en kötü durumda $\Theta(n^2)$. "Hangi $\Theta(n \log n)$ sıralama" kararı bu uzay/kararlılık/en-kötü-durum dengeleriyle verilir.



Şekil 11.5: Birleştirmeli sıralama = **böl-yönet**. Üstte özyineleme ağacı: dizi her seviyede tam ortadan **ikiye bölünür** (aşağı, böl), $\lceil \log_2 n \rceil + 1$ seviye sonra yapraklara (tek öğe, zaten sıralı; slate) iner. Sonra **aşağıdan yukarı** her düğüm iki sıralı çocuğunu **two-finger merge** ile birleştirir (amber, yukarı ok). Altta tek bir two-finger adımı: iki sıralı liste $[1, 2, 7, 8]$ ve $[3, 4, 6, 9]$ üstündeki iki parmak (i, j) karşılaştırılır, küçük olan ($\min(1, 3) = 1$) sonuca alınır ve o parmak kaydırılır — her öğeye tam bir kez dokunulur, bu yüzden birleştirme bir seviyede $\Theta(n)$. $\log n$ seviye $\times$ seviye-başına $\Theta(n)$ iş: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.



Şekil 11.6: Neden seçmeli sıralama $\Theta(n^2)$ ama birleştirmeli sıralama $\Theta(n \log n)$ — fark yineleme ağacının **seviye sayısındadır** ($n = 8$). **Sol (selection):** $T(n) = T(n-1) + \Theta(n)$ — her adım problemi yalnız **BİR** azaltır, alt-problem boyutu $8, 7, 6, \ldots, 1$ olur; bu yüzden 8 seviye (genel: n seviye). Çubuk genişlikleri o seviyedeki alt-problem boyutudur ($T(8)$ en uzun, $T(1)$ en kısa). n seviye $\times$ ortalama $\Theta(n)$ iş $\Rightarrow$ $\Theta(n^2)$. **Sağ (merge):** $T(n) = 2T(n/2) + \Theta(n)$ — her seviye problemi **YARIYA** böler, alt-problem sayısı $1, 2, 4, 8$ ve boyutları $8, 4, 2, 1$ olur; bu yüzden yalnız $\lceil \log_2 8 \rceil + 1 = 4$ seviye (genel: $\log n$ seviye). Her seviyede toplam genişlik (toplam iş) sabit $\Theta(n)$ kalır — alt-problemler küçülürken sayıları artar. $\log n$ seviye $\times \Theta(n)$/seviye $\Rightarrow$ $\Theta(n \log n)$. İki algoritma da seviye başına $\Theta(n)$ iş yapar; tek fark seviye sayısı: n'e karşı $\log n$ — $n = 10^6$'da bu, bir milyon kata karşı ≈ 20 kat demektir.

11.12 Sıralama Algoritmaları — Karşılaştırma

Algoritma	Çalışma süresi	Yerinde mi?	Yöntem
Permutation sort	$\Omega(n! \cdot n)$	Hayır	Kaba kuvvet (tüm permütasyonlar)
Selection sort	$\Theta(n^2)$	Evet	En büyüğü seç, sona koy, özyinele
Insertion sort	$\Theta(n^2)$	Evet	Solu sırala, yeni öğeyi yerine kaydır
Merge sort	$\Theta(n \log n)$	Hayır ($\Theta(n)$ ek alan)	Böl ve yönet, two-finger merge

Şekil 11.7 aynı dört satırı, çalışma süresini büyümeye göre renklendirerek görselleştirir.

Dört sıralama algoritması: çalışma süresi · yerinde mi? · yöntem

Algoritma	Çalışma süresi	Yerinde?	Yöntem
Permutation	**Ω(n!·n)**	**Hayır**	Tüm permütasyonlar (kaba kuvvet)
Selection	**Θ(n²)**	**Evet**	En büyüğü seç, sona koy, özyinele
Insertion	**Θ(n²)**	**Evet**	Solu sırala, yeni öğeyi kaydır
Merge	**Θ(n log n)**	**Hayır**	Böl-yönet, two-finger merge

Şekil 11.7: Dört sıralama algoritmasının karşılaştırması (§11): her satır bir algoritma, sütunlar **çalışma süresi**, **yerinde mi?** ve **yöntem**. Çalışma süresi büyümeye göre renklenir — permütasyon sıralama $\Omega(n! \cdot n)$ (tüm permütasyonları deneyen kaba kuvvet, en kötü; koyu amber); seçmeli ve eklemeli sıralama $\Theta(n^2)$ (amber); birleştirmeli sıralama $\Theta(n \log n)$ (yeşilimsi-slate = en iyi). "Yerinde mi?" sütunu ek alan kullanımını gösterir: seçmeli/eklemeli **Evet** (kopya üstünde $O(1)$ ek alanla swap), permütasyon ve birleştirmeli **Hayır** (merge $\Theta(n)$ ek alan ister). $n = 10^6$ ölçeğinde n^2 ile $n \log n$ arasındaki fark yaklaşık bir milyon kattır — bu yüzden büyük veride birleştirmeli sıralama seçilir.

Builder Notu — n^2 vs n log n Ölçek Sezgisi

Tablodaki $\Theta(n^2)$ ile $\Theta(n \log n)$ arasındaki fark, soyut değil — pratik bir uçurumdur:

- **Ölçek hesabı:** $n = 10^6$'da $n^2 = 10^{12}$, ama $n \log n \approx 10^6 \times 20 = 2 \times 10^7$. Oran ≈ 50.000 kat; saniyeler ile günler arasındaki fark. $n = 10^9$'da uçurum bir milyon kata yaklaşır.
- **İleriye → ML veri ön-işleme:** Milyonlarca örneği sıralamak (deduplication, tokenizasyon istatistikleri, nearest-neighbor için ön-sıralama) gerçek bir maliyettir; "hangi sıralama / hangi veri yapısı" kararı eğitim pipeline'ının darboğazını belirler.
- **Disiplin:** Bir builder, n'in büyüklük mertebesini görür görmez hangi karmaşıklığın "geçeceğini" tahmin eder: $n \leq 10^4$ ise n^2 tolere edilebilir; $n \geq 10^6$ ise $n \log n$ (veya daha iyisi) zorunludur. Bu tablo o sezginin temelidir.

11.13 Bu Dersin Özeti

1. **Küme (set)** arayüzü öğeleri **anahtara** göre tutar: find, insert, delete, find_min/max.
2. **Sırasız dizi** ile küme: tüm işlemler $O(n)$ (find = lineer tarama).
3. **Sıralı dizi** ile küme: find_min/max $O(1)$, find $O(\log n)$ (ikili arama) — ama kurma maliyeti $O(n \log n)$.
4. **Sıralama sözlüğü:** yıkıcı (destructive), yerinde (in-place).
5. **Permutation sort** $\Omega(n! \cdot n)$ — kullanılamaz kötü örnek.
6. **Selection/Insertion sort** $\Theta(n^2)$; özyineleme + substitution ile analiz.
7. **Merge sort** $\Theta(n \log n)$: böl-yönet + lineer two-finger birleştirme; $T(n) = 2T(n/2) + \Theta(n)$.

Tek Bir Cümle

Sıralama, bir kümeyi "anahtarla aranabilir" kılmanın bedelidir; merge sort bu bedeli böl-yönet ile $O(n \log n)$'e indirir.

11.14 Kontrol Soruları

Soru 1: Sırasız dizi ile sıralı dizi, küme arayüzünü gerçekleştirirken hangi işlemlerde ayrışır?

Cevap: İkisinde de insert/delete temelde $O(n)$'dir. Ayrışma **aramada** olur: sırasız dizide `find(k)` ve find_min/max her zaman $O(n)$ (lineer tarama); sıralı dizide `find(k)` ikili aramayla $O(\log n)$, find_min/max ise $O(1)$ (ilk/son öğe). Karşılığında sıralı diziyi *kurmak* $O(n \log n)$ sıralama gerektirir — tek seferlik bir ön maliyet.

Soru 2: Permutation sort neden $\Omega(n! \cdot n)$ ile ifade edilir, O(…) ile değil?

Cevap: Solomon permütasyonları üreten algoritmayı tanımlamaz, "sihirli" bir fonksiyon varsayar. Ama emin olduğu şey: n öğenin $n!$ permütasyonu vardır ve her birinin sıralılığını kontrol etmek $\Theta(n)$'dir, dolayısıyla iş *en az* $n! \cdot n$ kadardır. "En az" bir **alt sınırdır** → Ω. Kesin çalışma süresi verilmediği için

O (üst sınır) iddia edilmez.

Soru 3: Selection sort için T(n) = T(n−1) + Θ(n) yinelemesi neden Θ(n^2) verir? Substitution ile göster.

Cevap: $T(n) = cn^2$ tahmin et ve yerine koy: $cn^2 \stackrel{?}{=} c(n-1)^2 + \Theta(n)$. Sağ taraf $= c(n^2 - 2n + 1) + \Theta(n) = cn^2 - 2cn + c + \Theta(n)$. İki taraftan cn^2 çıkar: $0 \stackrel{?}{=} -2cn + c + \Theta(n)$. Bu, $\Theta(n) = 2cn - c$ demektir; sol ve sağ aynı mertebede (lineer), dolayısıyla tutarlı. Tahmin doğrulandı: $T(n) = \Theta(n^2)$. Sezgi: $n + (n-1) + \cdots + 1 = n(n+1)/2 = \Theta(n^2)$.

Soru 4: Merge sort neden selection sort'tan asimptotik olarak hızlıdır? Yinelemeleri karşılaştır.

Cevap: Selection sort: $T(n) = T(n-1) + \Theta(n)$ → her adımda problemi *bir* azaltır, n seviye $\times\Theta(n) = \Theta(n^2)$. Merge sort: $T(n) = 2T(n/2) + \Theta(n)$ → her adımda problemi *yarıya* böler, yalnızca $\log n$ seviye × her seviyede toplam $\Theta(n)$ iş $= \Theta(n \log n)$. Farkın kaynağı: lineer azaltma (n seviye) yerine logaritmik bölme ($\log n$ seviye).

11.15 Egzersizler

Egzersiz 1. Sıralı bir dizide `find_prev(k)` ve `find_next(k)` işlemlerini tasarla. İkili aramayla $O(\log n)$'de nasıl yapılır?

Egzersiz 2. `prefix_max` için tümevarımla doğruluğu tam yaz: taban durum ($i = 0$) ve tümevarım adımı (i'deki öğe ile 0..i−1'in en büyüğünün maksimumu).

Egzersiz 3. Insertion sort'u özyinelemeli yaz ve $T(n) = T(n-1) + O(n)$ yinelemesini substitution ile $\Theta(n^2)$'ye bağla. En iyi durum (zaten sıralı girdi) ne olur?

Egzersiz 4. Python'da merge sort ile yerleşik `sorted`'ı farklı n'lerde `time.perf_counter` ile karşılaştır; $n \log n$ eğrisini gözlemle:

```
import time, random

def measure(n):
    A = [random.random() for _ in range(n)]
    t0 = time.perf_counter()
    sorted(A)
    print(n, time.perf_counter() - t0)

for n in (10_000, 100_000, 1_000_000):
    measure(n)
```

Egzersiz 5. İki sıralı listeyi birleştiren two-finger `merge` fonksiyonunu yaz ve neden $\Theta(n)$ olduğunu (her öğeye bir kez dokunulduğunu) açıkla.

11.16 Sonraki Ders İçin Hazırlık

Ders 4 (L4): Hashing

Jason Ku ile, sıralı dizinin $O(\log n)$ aramasını bile geçen bir yapıya — **hash tablosuna** — geçiyoruz: find/insert/delete *beklenen* $O(1)$. Sıralamadan vazgeçip anahtarları doğrudan adreslere "serpiştirmenin" maliyeti ve riski (çakışma) ele alınır. (Not: ders akışında araya **Problem Oturumu 2** girer.)

Ders 4 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle Egzersiz 4'ü (merge sort ölçümü) çöz.
- Sıralama tablosunu (Bölüm 11) ezberden çizebil.
- Ana cümleyi tekrar oku: *"Sıralama, bir kümeyi anahtarla aranabilir kılmanın bedelidir."*

11.17 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Küme (set) arayüzü	Anahtarla erişilen koleksiyon (find, insert, find_min)	Böl. 2
Sırasız dizi	Küme gerçekleştirimi; tüm işlemler $O(n)$	Böl. 4
Sıralı dizi + ikili arama	find $O(\log n)$, find_min/max $O(1)$; kurma $O(n \log n)$	Böl. 5
Yıkıcı / yerinde	Destructive: `A`'yı ezer; in-place: ek $O(1)$ alan	Böl. 6
Permutation sort	Tüm permütasyonlar; $\Omega(n! \cdot n)$	Böl. 7
Selection sort	En büyüğü seç-sona-koy; $\Theta(n^2)$	Böl. 8
Merge sort	Böl-yönet + two-finger merge; $\Theta(n \log n)$	Böl. 10
Substitution yöntemi	Yineleme için form tahmin et, yerine koy, doğrula	Böl. 8, 10
Two-finger merge	İki sıralı listeyi $\Theta(n)$'de birleştirme	Böl. 10

11.18 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, ileri algoritma ve veri mühendisliğinin temel dilini kurar — köprülerin özeti:

1. **Set arayüzü** → Python `dict/set`, veritabanı tabloları: anahtarla erişim her veri katmanının temeli.
2. **Sıralı dizi + ikili arama** → veritabanı indeksleri (B-tree), arama motoru sıralı gönderim listeleri:

$O(\log n)$ sorgu.

3. **Böl ve yönet (merge sort)** → OMSCS CS 6515'in çekirdek paradigması; quicksort, FFT, en yakın çift hep aynı kalıp.
4. **Substitution / yineleme analizi** → her özyinelemeli algoritmanın çalışma süresini kapalı forma bağlama disiplini.
5. **Destructive / in-place** → bellek-kısıtlı sistemler: yerinde sıralama, ek tampon ayıramayan gömülü ortamlarda kritik.
6. n^2 **vs** $n \log n$ → ölçek sezgisi: $n = 10^6$'da n^2 bir milyon kat daha yavaş; "hangi sıralama" kararının pratik bedeli.

! Tek bir şey alıp gideceksen

Bir kümeyi sıralı tutmak aramayı $O(\log n)$'e indirir, ama sıralamanın kendisi $O(n \log n)$'dir. Merge sort bu bedeli "böl ve yönet" ile öder; ve $T(n) = 2T(n/2) + \Theta(n)$ yinelemesi, tüm verimli özyinelemeli algoritmaların habercisidir.

12 Hashing

Karşılaştırma modeli alt sınırı, doğrudan erişim dizisi, hash fonksiyonu, çakışma + zincirleme ve evrensel hashing ile beklenen $O(1)$

ℹ Bölüm bilgisi

- **Ku'nun videosu:** YouTube — Lecture 4: Hashing (≈53 dk)
- **OCW sayfası:** MIT 6.006 Lecture 4: Hashing
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 5 (L4)
- **Hoca:** Jason Ku
- **Okuma süresi:** ≈26 dk

12.1 Bu Derste Ne Var?

Ders 3, kümeyi (set) sıralı diziyle gerçekleştirip aramayı $O(\log n)$'e indirdi. Bu ders tek bir soruyla başlar: *daha hızlı olur mu?* Jason Ku önce **olamaz** der — karşılaştırma modelinde $\log n$ bir alt sınırdır — sonra modeli genişletip **olur** der: **hashing** ile find/insert/delete *beklenen* $O(1)$.

Üç temel kavram bu derste yan yana gelir:

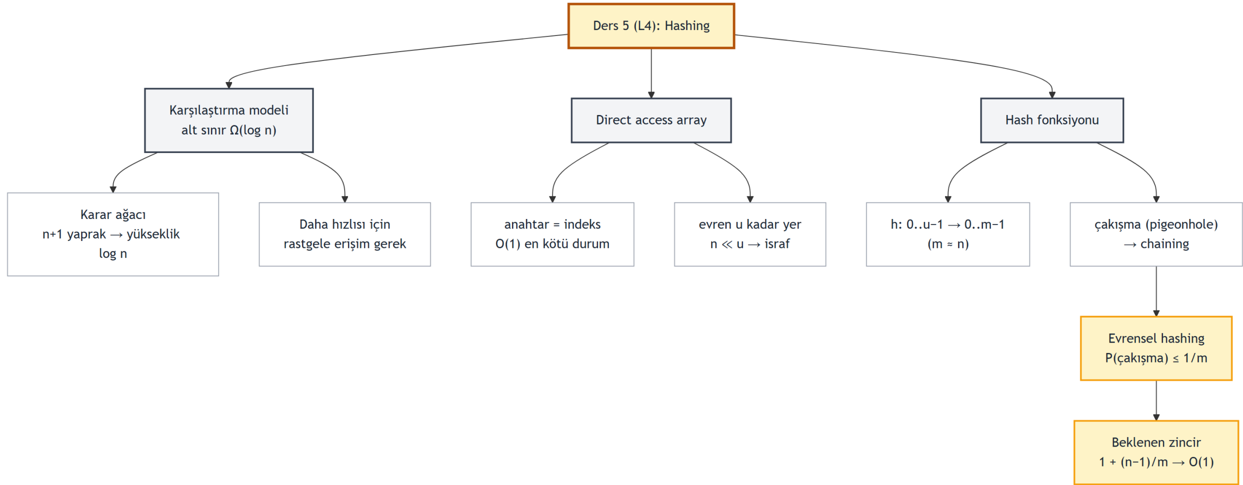
1. **Karşılaştırma modeli alt sınırı** — yalnızca anahtarları karşılaştıran her algoritma için arama $\Omega(\log n)$'dir (karar ağacı argümanı).
2. **Direct access array → hash tablosu** — anahtarı doğrudan adres yap; evren çok büyükse bir hash fonksiyonuyla küçült.
3. **Evrensel hashing** — rastgele seçilen bir hash ailesi, beklenen zincir uzunluğunu sabit yapar.

> *"Today we are going to be talking about hashing."* — Ku, 0:20

Builder Notu — Hash Tablosu = Python dict ve Randomized Algoritma

Bu ders ML değil, ama ML'in ve her ciddi yazılımın altındaki **erişim mekanizmasını** ve **olasılıksal analizi** kurar:

- **Geriye → Python (Phase 1):** Python `dict/set` bir hash tablosudur (open addressing şeması). Bu ders onun iç mantığını ve "neden bazen yavaşladığını" açıklar — bedava görünen `d[key]` çağrısının arkasındaki $O(1)$ buradan gelir.
- **Geriye → Stat 110 (Phase 1):** dersin kalbi olasılıktır — **gösterge rastgele değişkeni** (indicator RV), **beklenen değer**, **beklentinin doğrusallığı** (linearity of expectation). Evrensel hashing,



Şekil 12.1: Ders 5'in (L4) kavram haritası: karşılaştırma modelinin Ω(log n) sınırından, doğrudan erişim dizisine (O(1) ama u kadar yer), oradan hash fonksiyonuna (m ≈ n), çakışmaya ve zincirleme + evrensel hashing yoluyla beklenen O(1)'e.

randomized algoritmanın bu kursta gördüğün **ilk ciddi örneğidir**: garanti "her girdide" değil, "beklenen" anlamda verilir.

- **İleriye → ML ve sistem tasarımı:** feature hashing, embedding indeksleme, cache, veritabanı indeksi, dağıtık sistemlerde consistent hashing — hepsi anahtar → kova $O(1)$ erişimine dayanır.

Tek cümle: *Sıralamayı bırakıp anahtarı doğrudan adrese "serpiştirirsek" arama $O(1)$ olur — yeter ki hash fonksiyonunu rastgele seçip çakışmayı beklenen anlamda küçük tutalım.*

12.2 Set'i $O(\log n)$'den Hızlı Yapabilir miyiz?

Sıralı dizi, `find(k)`'yi ikili aramayla $O(\log n)$ yapar. $\log n$ zaten küçüktür (gerçek problemlerde ~30'u geçmez), ama Ku daha hızlısını ister. Soru iki parçalıdır: (1) belirli bir modelde $O(\log n)$'den hızlı **mümkün mü**, (2) değilse modeli nasıl değiştiririz?

Cevap: **karşılaştırma modelinde** mümkün değildir (Bölüm 12.3). Ama word RAM modelinin **rastgele erişim** gücünü kullanırsak mümkündür (Bölüm 12.4 ve sonrası).

12.3 Karşılaştırma Modeli ve Alt Sınır

Karşılaştırma modeli: sakladığımız nesneleri "kara kutu" gibi düşünürüz; onları ayırt etmenin tek yolu iki anahtarı **karşılaştırmaktır** (eşit mi, büyük mü, küçük mü). Ders 3'teki üç sıralama da bu modeldeydi.

> *"the only way that I can distinguish between them is to say… I can do a comparison on those keys."* — Ku, 5:15

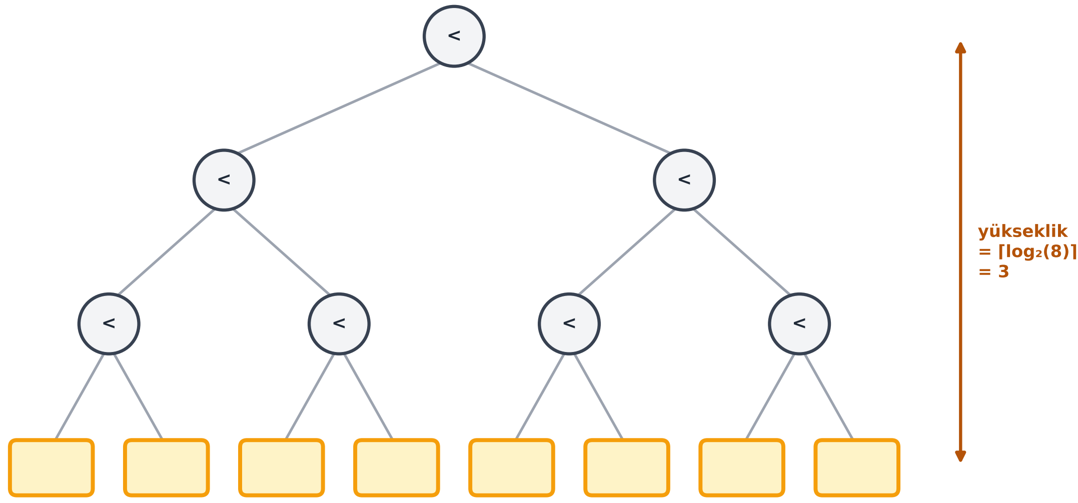
Bir karşılaştırma algoritmasını **karar ağacı (decision tree)** olarak çiz (Şekil 12.2): iç düğümler karşılaştırmalardır (iki dallı: doğru/yanlış), yapraklar çıktılardır. En kötü durumdaki karşılaştırma sayısı = en uzun kök-yaprak yolu = ağacın **yüksekliği**.

Çalışılan Örnek — $\log n$ alt sınırı. Doğru bir arama algoritması, sakladığı n öğeden herhangi birini *veya* "yok"u döndürebilmeli → en az $n+1$ **yaprak** gerekir. İkili bir ağacın $n+1$ yaprağı varsa, minimum yüksekliği $\log(n+1)$'dir (en iyi durum: dengeli ağaç).

> *"the minimum height of any binary tree that has at least n plus 1 leaves... it's log n."* — Ku, 12:44

Demek ki yalnızca karşılaştırmayla, arama $\Omega(\log n)$ sürer. Daha hızlısı için karşılaştırmadan güçlü bir işlem gerekir.

Karşılaştırma modeli: ikili karar ağacı — n+1 yaprak → yükseklik log n → Ω(log n)



Şekil 12.2: Karşılaştırma modeli ve aramanın $\Omega(\log n)$ alt sınırı. Yalnızca **karşılaştırma** yapabilen herhangi bir arama algoritması bir **ikili karar ağacı** olarak çizilebilir: her **iç düğüm** bir anahtar karşılaştırmasıdır (slate daire, iki dallı — örn. $A[mid] < k$), ve algoritmanın verebileceği her olası **çıktı** bir **yaprak**tır (amber kutu). n öğelik bir kümede arama n farklı öğeden birini ya da "yok"u döndürebilmeli → en az $n+1$ yaprak gerekir. Burada $n = 7$, yani 8 yaprak. İkili bir ağacın yüksekliği h ise en fazla 2^h yaprağı olur → $2^h \geq n+1 \Rightarrow h \geq \log_2(n+1)$, yani minimum yükseklik $\lceil \log_2(n+1) \rceil = 3$. En kötü durumdaki karşılaştırma sayısı = en uzun kök-yaprak yolu = ağacın yüksekliği → karşılaştırmaya dayalı arama $\Omega(\log n)$'dir. İkili arama ($O(\log n)$, Şekil 11.3) bu alt sınıra ulaşır; hash tabloları ise karşılaştırma modelinden *çıkıp* anahtarı doğrudan indekse çevirerek beklenen $O(1)$'e iner.

12.4 Direct Access Array (Doğrudan Erişim Dizisi)

Karşılaştırma sabit dallanma (2 yol) verir; bize **sabit-olmayan dallanma** lazım. word RAM'in **rastgele erişimi** tam bunu sağlar: bir sayıyla belleğin herhangi bir yerine sabit zamanda gidebiliriz.

Fikir: anahtarı k olan öğeyi doğrudan k**. indekse** koy. Anahtarı 10 olan öğe, indeks 10'da durur (Şekil 12.3).

> *"This is what I call a direct access array… It takes constant time, worst case."* — Ku, 16:24

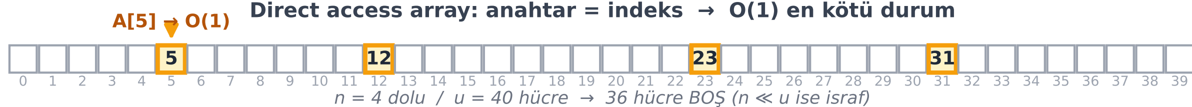
`find`, `insert`, `delete`: hepsi $O(1)$ **en kötü durum**. Mükemmel görünüyor — ama bir sorun var.

12.5 Problem: Anahtar Evreni Çok Büyük

Direct access array, anahtar evreni kadar yer ister. u = saklanabilecek en büyük anahtarın boyutu (anahtar evreninin büyüklüğü). MIT ID'leri 9 haneli → $u \approx 10^9$, oysa sınıfta yalnızca $n \approx 300$ kişi var. u, n'den çok büyükse, çoğu hücre boş kalır — büyük bellek israfı (Şekil 12.3).

> *"u is the maximum size of any number that I'm storing. It's the size of the universe of space of keys"* — Ku, 19:09

İki varsayım gerekir: (1) anahtarlar **tam sayı** olmalı (adres olarak kullanılacak); (2) $u < 2^w$ (w = word boyutu), ki anahtar bir word'e sığıp sabit zamanda erişilebilsin.



Şekil 12.3: **Doğrudan erişim dizisi** (direct access array): bir öğenin anahtarı doğrudan dizi **indeksi** olarak kullanılır — `A[k]`'ye ofset aritmetiğiyle erişilir, dolayısıyla `insert`/`find`/`delete` **en kötü durumda** $O(1)$ (karşılaştırma yok). Bedeli bellektir: dizinin uzunluğu, anahtar **evreni** u kadar olmak zorundadır. Burada temsilî olarak $u = 40$ hücre ayrılmış ama yalnızca $n = 4$ anahtar (5, 12, 23, 31) saklanıyor; **amber** hücreler kendi indekslerinde dolu, kalan 36 hücre (slate) **boş** duruyor. Gerçek ölçekte fark uçurumdur: MIT ID anahtarları için $u \approx 10^9$ iken tipik $n \approx 300$ kayıt tutulur → hücrelerin ~%99.99997'si boş kalır, yani 10^9 word'lük dizi neredeyse tamamen israftır. Bu israf, bir sonraki adımı zorlar: büyük evreni küçük bir aralığa ($m \approx n$) sıkıştıran bir **hash fonksiyonu** (Şekil 12.4).

> Builder Notu — O(1) Rastgele Erişim ve Hash'in ML'de Kullanımı
>
> Doğrudan erişimin "anahtar = indeks → $O(1)$" sezgisi, ML'in en sıcak yollarında (hot path) tekrar tekrar karşına çıkar — ama her zaman tam evreni ayırmadan, hash'le sıkıştırarak:
>
> - **İleriye → feature hashing (hashing trick):** Yüksek-kardinaliteli kategorik öznitelikleri (u devasa:

tüm olası kelimeler/URL'ler) sabit boyutlu (m) bir vektöre $h(k)$ ile indirger. Direct access array'in israfını çözmenin ML'deki birebir karşılığı — çakışmaları *bilerek* tolere eder.

- **İleriye → embedding indeksleme:** Token → indeks → embedding satırı erişimi tam bir direct access array'dir; sözlük boyutu m (u değil) kadar satır ayrılır, h token'ı bu aralığa eşler. Karpathy'nin makemore serisindeki embedding tablosu budur.
- **İleriye → bloom filter sezgisi:** "Bu anahtarı daha önce gördük mü?" sorusunu birkaç hash fonksiyonu + bit dizisiyle çok az bellekte cevaplar — yine u değil m kadar yer, çakışma karşılığında.

12.6 Hash Fonksiyonu

Çözüm: büyük anahtar evrenini ($0 \dots u-1$) küçük bir aralığa ($0 \dots m-1$, $m \approx n$) **bir fonksiyonla** sıkıştır. Bu fonksiyona h (hash fonksiyonu) denir. Artık öğeyi k. indekste değil, $h(k)$**. indekste** sakla.

Boyutu $m \approx n$ olan bu yapıya **hash tablosu** denir. Alan artık $O(n)$'dir — sorun çözüldü gibi. Ama sıkıştırma yeni bir sorun doğurur: **çakışma** (Şekil 12.4).

12.7 Çakışma (Collision) ve Çözümleri

Büyük evreni küçük aralığa eşleyince, **güvercin yuvası ilkesi** (pigeonhole) gereği bazı farklı anahtarlar aynı indekse düşer. Kötü bir h seçilirse, $u > n^2$ iken n anahtarın hepsi tek bir indekse gidebilir → hiçbir kazanç yok.

> *"I really want a hash function that will evenly distribute keys over this space."* — Ku, 27:27

Çakışmayla başa çıkmanın iki yolu:

- **Açık adresleme (open addressing):** çakışan öğeyi dizide başka boş bir yere koy. Python bunu kullanır; analizi zordur (bu derste işlenmez).
- **Zincirleme (chaining):** o indekste tek öğe yerine, bir **zincir** (chain — linked list veya dinamik dizi) tut. Çakışanları zincire ekle; ararken zinciri lineer tara (Şekil 12.4).

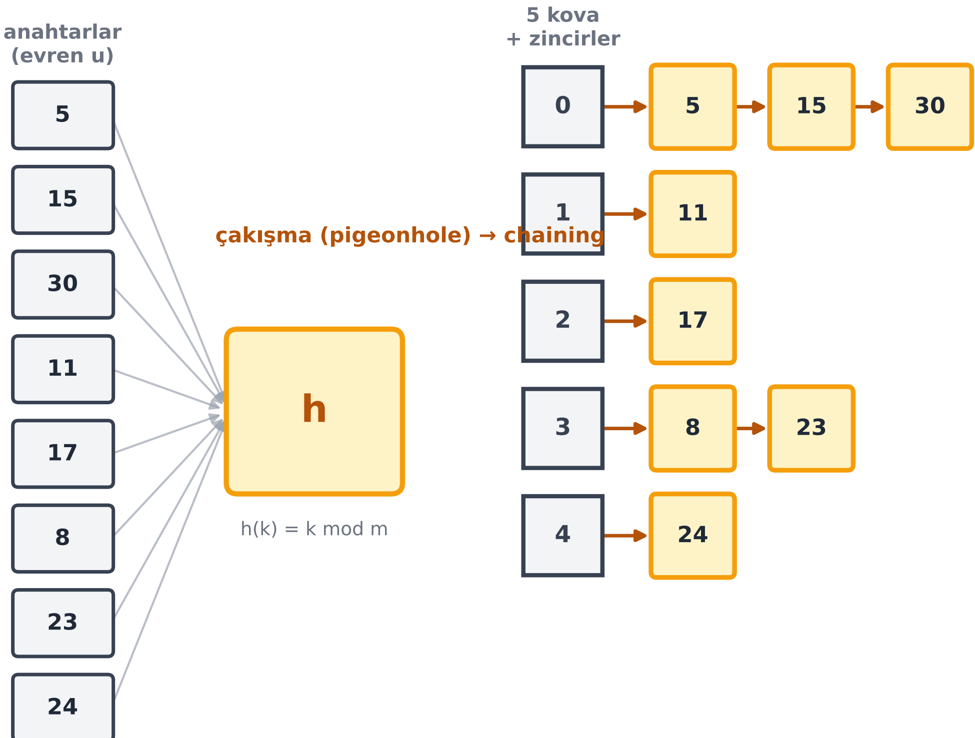
> *"this is an idea called chaining."* — Ku, 32:11

Hedef: zincirleri **kısa** tutmak. Zincirde sabit sayıda öğe varsa, hepsini taramak yine $O(1)$'dir.

12.8 Hash Fonksiyonu Seçimi: Bölme Yöntemi

En basit hash: **bölme yöntemi (division method)**, $h(k) = k \bmod m$. Büyük anahtarı m'ye böler, kalanı indeks yapar.

> *"Modulus– great. This is called the division method."* — Ku, 34:38



Şekil 12.4: **Hash fonksiyonu + zincirleme** (chaining). Sol: anahtarlar büyük bir evrenden $(0 \ldots u-1)$ gelir. Ortadaki **hash fonksiyonu** h bu evreni $m \approx n$ küçük kovaya **sıkıştırır** ($h(k) = k \bmod m$, burada $m = 5$). Sıkıştırma kaçınılmaz olarak **çakışma** doğurur (güvercin yuvası ilkesi: $u > m$ olduğundan en az iki anahtar aynı kovaya düşer). Çözüm **zincirleme**: aynı kovaya düşen anahtarlar o kovanın **zincirine** (bağlı listesine, amber kutular) eklenir. Burada deterministik olarak kova 0 üç anahtarlık bir zincir tutar ($5, 15, 30$ — hepsi $\bmod 5 = 0$) ve kova 3 iki anahtarlık bir zincir tutar $(8, 23)$. Arama, anahtarın düştüğü kovanın zincirini lineer tarar; zincirler kısa tutulursa ($m = \Theta(n)$) işlemler **beklenen** $O(1)$'dir.

Anahtarlar düzgün (uniform) dağılmışsa iyidir — ama bu, girdiye bir dağılım şartı koyar. Python bunu biraz bit karıştırmayla kullanır; **deterministik** olduğu için, performansı bozan kötü girdi dizileri *vardır*. 6.006 ise girdiden **bağımsız** bir garanti ister: hangi anahtarlar saklanırsa saklansın iyi çalışmalı. Sabit (deterministik) bir hash bunu sağlayamaz.

> Builder Notu — Hash-Flooding DoS ve Güvenlik
>
> "Deterministik hash'in kötü girdisi vardır" cümlesi soyut bir uyarı değil; gerçek bir güvenlik açığının (CWE-407 algorithmic complexity attack) tam tanımıdır:
>
> - **Saldırı:** Hash fonksiyonu sabit ve *bilinen* olduğunda (eski PHP, Python <3.3, birçok web framework), saldırgan hepsi aynı kovaya düşen anahtarlar üretir. Zincir $O(n)$ uzar; $O(1)$ sandığın her `insert/find` $O(n)$ olur ve n istek $O(n^2)$ işe patlar → sunucu tek bir HTTP form gönderimiyle çöker (**hash-flooding DoS**).
> - **Çözüm = bu dersin konusu:** hash fonksiyonunu **rastgele tohumla** (random seed) seç. Saldırgan tohumu bilmediği için kötü girdiyi *önceden* tasarlayamaz — bu, Bölüm 12.9'deki evrensel hashing'in pratik gerekçesidir.
> - **Standart oldu:** Python 3.3+ `PYTHONHASHSEED`'i varsayılan olarak rastgeleler; tam da bu saldırıyı engellemek için.

12.9 Evrensel Hashing (Universal Hashing)

Çözüm: hash fonksiyonunu önceden değil, **sonradan rastgele** seç. Kullanıcı girdisini belirler; sonra biz rastgele bir hash seçeriz — hangisini seçtiğimizi bilmediği için kötü girdi vermesi zorlaşır.

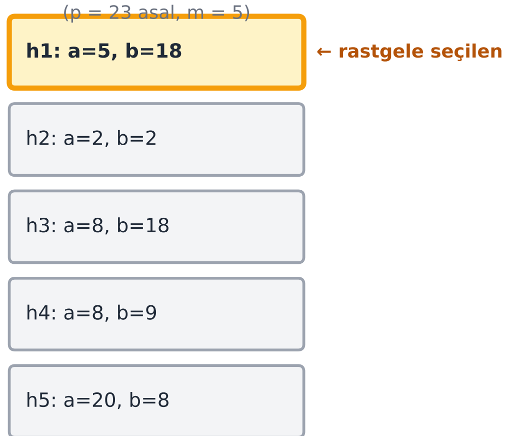
> *"choose one randomly later… I'm going to choose a random hash function."* — Ku, 38:31

Tüm hash fonksiyonları arasından seçmek imkânsızdır (çok fazla). Bunun yerine bir **aile**den seçeriz. **Evrensel hash ailesi** örneği:

$$h(k) = ((a \cdot k + b) \bmod p) \bmod m$$

Burada p, u'dan büyük sabit bir asal; $a \neq 0$ ve b ise hash tablosu kurulurken **rastgele** seçilir. **Evrensellik özelliği:** rastgele seçilen bir h için, herhangi iki farklı anahtarın çakışma olasılığı $\leq 1/m$'dir (Şekil 12.5).

> Builder Notu — Universal Hashing ve Stat 110 (Indicator RV)
>
> Bu, kursta gördüğün **randomized algoritmanın ilk ciddi örneğidir** — ve doğrudan Phase 1 Stat 110'a bağlanır:
>
> - **Gösterge rastgele değişkeni (indicator RV):** "i ve j çakıştı mı?" sorusunu $0/1$ bir değişkenle (X_{ij}) modellersin. Bir indicator RV'nin beklenen değeri, o olayın olasılığına eşittir: $E[X_{ij}] = P(X_{ij} = 1) \leq 1/m$. Doğum günü problemiyle (Ders 1) aynı olasılık aletidir.
> - **Beklentinin doğrusallığı (linearity of expectation):** Zincir uzunluğu birçok indicator RV'nin toplamıdır; toplamın beklentisi, değişkenler bağımlı olsa bile beklentilerin toplamıdır. Bölüm 12.10'deki $E[X_i] = 1 + (n-1)/m$ türetimi bunun üzerine kuruludur.



Şekil 12.5: **Evrensel hashing** (Ku §8): tek bir hash fonksiyonu sabitlenmez — bunun yerine $h(k) = ((a \cdot k + b) \bmod p) \bmod m$ ailesinden, girdi belirlendikten **sonra** a, b **rastgele** seçilir ($1 \leq a \leq p-1$, $0 \leq b \leq p-1$; p anahtarlardan büyük bir asal). Solda aile üyeleri ($h_1, \ldots, h_5$), her biri farklı (a, b) ile; **amber** olan rastgele seçilen h'tir (burada $a = 5, b = 18$). Sağ panel evrensellik garantisini ampirik doğrular: iki sabit, farklı anahtar $k_1 = 7, k_2 = 18$ için 2000 rastgele h denenir; çakışma oranı 0.166, teorik sınır $1/m = 0.200$'ün altında kalır — yani **rastgele bir** h **için herhangi iki farklı anahtarın çakışma olasılığı** $\leq 1/m$'dir. Bu garanti *girdiden bağımsızdır* (saldırgan anahtarları h'ten önce seçemez), $p = 23$, $m = 5$.

- **Köprü:** “garanti her girdide değil, *beklenen* anlamda” disiplini, ML’deki Monte Carlo tahmini, dropout, stokastik gradyan inişi gibi *rastgeleliği bilerek kullanan* yöntemlerin temel düşünme tarzıdır.

12.10 Beklenen Zincir Uzunluğu

Evrensellik, zincirlerin *beklenen* uzunluğunu sabit yapar. İspat olasılıkla yapılır (Stat 110 köprüsü, Şekil 12.6).

Çalışılan Örnek — beklenen zincir uzunluğu. Bir **gösterge rastgele değişkeni** tanımla: $X_{ij} = 1$ eğer anahtar i ve j çakışırsa ($h(k_i) = h(k_j)$), aksi halde 0. i’nin düştüğü indeksteki zincir uzunluğu $X_i = \sum_j X_{ij}$ ($j = 0 \ldots n-1$). **Beklentinin doğrusallığı** ile:

$$E[X_i] = \sum_j E[X_{ij}]$$

> *“Linearity of expectation”* — Ku, 49:05

Toplamı iki parçaya ayır: $j = i$ terimi (anahtar kendisiyle kesin çakışır, $E = 1$) ve $j \neq i$ terimleri (evrensellik gereği her biri $E[X_{ij}] \leq 1/m$). $n-1$ tane $j \neq i$ terimi var:

$$E[X_i] = 1 + (n-1)/m$$

> *“this should equal 1 plus n minus 1 over m… we’re expected to have our chain lengths be constant”* — Ku, 51:26

m’yi n’de lineer seçersek ($m = \Theta(n)$), $(n-1)/m$ sabit olur → beklenen zincir uzunluğu $O(1)$. Dolayısıyla find/insert/delete *beklenen* $O(1)$.

Builder Notu — Beklenen vs En Kötü Durum

Bu dersin en ince fikri, garantinin *tipini* değiştirmektir: en kötü durum $O(n)$ olan bir yapının **beklenen** $O(1)$ olduğunu kanıtlamak.

- **Randomized garanti:** Evrensel hashing’le hâlâ kötü bir durum *vardır* (tüm anahtarlar tek kovaya düşebilir), ama olasılığı küçüktür ve *girdiye değil, bizim rastgele seçimimize* bağlıdır. Saldırgan kötü girdi tasarlayamaz çünkü hangi h’yi seçtiğimizi bilmez.
- **İleriye → ML’de beklenen-değer analizi:** Monte Carlo tahmini, minibatch SGD, dropout — hepsi “her örnekte değil, *beklentide* doğru/iyi” mantığıyla çalışır. Bir tahminin yansızlığını (unbiasedness) göstermek, tam olarak bu $E[\cdot]$ türetimini yapmaktır.
- **Disiplin:** “En kötü durum mu, beklenen mi, amortize mi?” sorusu, bir veri yapısı seçerken sorman gereken ilk sorudur — üçü farklı garantilerdir, üçü farklı risklerdir.

Beklenen zincir uzunluğu E[Xᵢ] = 1 + (n−1)/m · m=Θ(n) → O(1)

m = 20 sabit: E = 1+(n−1)/m (lineer)
m = Θ(n) (m=n): E ≈ 2 (sabit, O(1))
ampirik (m sabit)
ampirik (m = n)

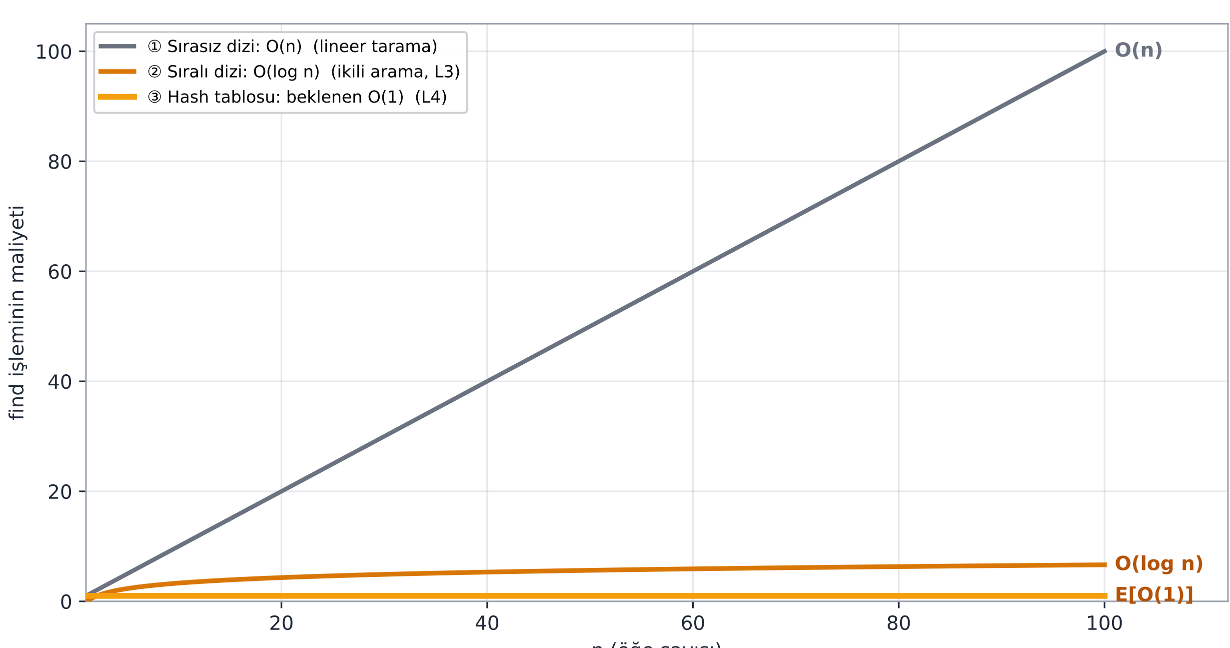
beklenen zincir uzunluğu E[Xᵢ]
n (saklanan anahtar sayısı)

Şekil 12.6: **Beklenen zincir uzunluğu** (İMZA, Stat 110 köprüsü). Bir anahtarın düştüğü kovanın zincir uzunluğu $X_i = 1 + \sum_{j \neq i} X_{ij}$, burada X_{ij} anahtar j'nin i ile çakışıp çakışmadığını gösteren **gösterge rastgele değişkeni** (indicator RV). Evrensel hashing'de $P(X_{ij} = 1) = E[X_{ij}] \leq 1/m$; **beklentinin doğrusallığı** ile $E[X_i] = 1 + \sum_{j \neq i} E[X_{ij}] = 1 + (n-1)/m$ — toplamın beklentisi, çakışmalar bağımsız olmasa bile beklentilerin toplamıdır. Amber çizgi: $m = 20$ **sabit** → $(n-1)/m$ lineer büyür, zincirler uzar. Slate çizgi: $m = \Theta(n)$ (burada $m = n$) → $(n-1)/m \approx 1$ sabit kalır, $E[X_i] \approx 2$ yani **beklenen** $O(1)$. Noktalar deterministik (sabit seed) ampirik simülasyon: n farklı anahtar rastgele evrensel hash ile yerleştirilir, her anahtarın zincir uzunluğu ortalanır — ampirik değerler teorik çizgilerin tam üstünde oturur (ör. $n = 200$, $m = 20$: teori 10.95, ampirik 10.935). Hash tablosunu $m = \Theta(n)$ tuttuğun sürece arama/ekleme/silme beklenen sabit zamandır.

12.11 Dinamik Hashing

n büyüdükçe, sabit m artık n'de lineer kalmayabilir; o zaman zincirler uzar. Çözüm: tıpkı dinamik dizide olduğu gibi, n çok büyüdüğünde **tüm hash tablosunu yeni (daha büyük) m ile yeniden kur**. Doğru oranda yeniden boyutlandırırsan, bu pahalı işlem nadiren olur — amortize edilmiş sabit zaman.

Bu dersin tüm yolculuğu — $\Omega(\log n)$ karşılaştırma sınırından beklenen $O(1)$ hash'e — Şekil 12.7'da tek bir grafikte toplanır.



Şekil 12.7: Arama maliyetinin tüm kurs boyunca evrimi — bu dersin sentezi. **Sırasız dizi:** $O(n)$ (slate) — `find` baştan sona lineer tarar; en kötü durumda n öğenin hepsine bakılır (L1/L2). **Sıralı dizi:** $O(\log n)$ (koyu amber) — ikili arama her adımda aralığı yarıya böler, yalnız ortaları inceler; ama bu **karşılaştırma modelinin alt sınırıdır** ($\Omega(\log n)$, L3) — karşılaştırma yaparak $O(\log n)$'den hızlı arama imkânsızdır. **Hash tablosu: beklenen** $O(1)$ (amber, vurgulu) — anahtarı doğrudan bir kovaya hash'leyerek karşılaştırma modelinin alt sınırını **aşar**; $m = \Theta(n)$ kova ve evrensel hash ile beklenen zincir uzunluğu $1+(n-1)/m = O(1)$ olur (L4). n büyüdükçe $O(n)$ doğrusal patlar, $O(\log n)$ yavaşça tırmanır, ama hash'in düz $E[O(1)]$ çizgisi n'den bağımsız sabit kalır — kursun ulaştığı kazanç budur.

12.12 Bu Dersin Özeti

1. **Karşılaştırma modelinde** arama $\Omega(\log n)$'dir: karar ağacının $n+1$ yaprağı için minimum yükseklik $\log n$.
2. **Direct access array**: anahtar = indeks → $O(1)$, ama anahtar evreni u kadar yer ister.
3. **Hash fonksiyonu** h: $0 \ldots u-1 \to 0 \ldots m-1$, $m \approx n$; alanı $O(n)$'e indirir.
4. **Çakışma** kaçınılmazdır (pigeonhole); çözüm: open addressing veya **chaining**.

5. **Bölme yöntemi** ($k \bmod m$) deterministiktir → kötü girdi vardır.
6. **Evrensel hashing**: rastgele h ailesi, çakışma olasılığı $\leq 1/m$.
7. **Beklenen zincir uzunluğu** = $1 + (n-1)/m$; $m = \Theta(n)$ ile $O(1)$ (gösterge RV + beklentinin doğrusallığı).

! Tek Bir Cümle

Hashing, $O(\log n)$ karşılaştırma sınırını rastgele erişimle aşar; ve hash fonksiyonunu *rastgele* seçmek, en kötü durum garantisini *beklenen* $O(1)$'e dönüştürür.

12.13 Kontrol Soruları

i Soru 1: Karşılaştırma modelinde aramanın neden Ω(log n) olduğunu, karar ağacıyla bir cümlede özetle.

Cevap: Doğru bir arama, n öğeden herhangi birini ya da "yok"u döndürebilmeli → karar ağacında en az $n+1$ yaprak gerekir. İkili bir ağaçta $n+1$ yaprağa ulaşmak için minimum yükseklik $\log(n+1)$'dir, ve en kötü durumdaki karşılaştırma sayısı = en uzun kök-yaprak yolu = yükseklik. Dolayısıyla en az $\log n$ karşılaştırma şarttır. (Hashing bu sınırı aşar çünkü karşılaştırma değil, rastgele erişim — sabit-olmayan dallanma — kullanır.)

i Soru 2: Direct access array O(1) verirken neden her zaman kullanmıyoruz?

Cevap: Direct access array, anahtar evreni u kadar bellek ister. Anahtarlar büyükse (örneğin 9 haneli MIT ID → $u \approx 10^9$) ama öğe sayısı küçükse ($n \approx 300$), dizinin neredeyse tamamı boş kalır — kabul edilemez bellek israfı. Hash fonksiyonu tam bu sorunu çözer: evreni $m \approx n$ boyutuna sıkıştırır (çakışma pahasına).

i Soru 3: Deterministik bir hash fonksiyonu neden en kötü durum garantisi veremez? Universal hashing bunu nasıl aşar?

Cevap: u büyükse, herhangi *sabit* bir h için (pigeonhole), tüm n anahtarı aynı indekse eşleyen bir girdi *vardır* → o girdide zincir uzunluğu n, arama $O(n)$. Saldırgan/kötü şanslı girdi bunu tetikleyebilir. Universal hashing, h'yi girdiden *sonra* rastgele seçer; kullanıcı hangi h'nin seçileceğini bilmediğinden kötü girdi tasarlayamaz. Garanti "her girdide" değil, "beklenen" anlamda sağlanır: çakışma olasılığı $\leq 1/m$.

i Soru 4: E[X_i] = 1 + (n−1)/m formülündeki '1' ve '(n−1)/m' terimleri neyi temsil eder?

Cevap: X_i, anahtar i'nin düştüğü zincirin uzunluğu = $\sum_j X_{ij}$. Beklentinin doğrusallığıyla $E[X_i] = \sum_j E[X_{ij}]$. **"1"**: $j = i$ terimi — anahtar kendisiyle kesin "çakışır" (aynı indeks), yani zincirde her zaman en az kendisi vardır. **"$(n-1)/m$"**: kalan $n-1$ anahtarın her biri, evrensellik gereği $\leq 1/m$ olasılıkla i ile çakışır; toplamları $(n-1) \cdot (1/m)$. $m = \Theta(n)$ seçilirse bu terim sabit kalır → $E[X_i] = O(1)$.

12.14 Egzersizler

Egzersiz 1. $n+1$ yaprağı olan ikili bir ağacın minimum yüksekliğinin $\log(n+1)$ olduğunu bir yineleme (recurrence) kurarak göster. (İpucu: yükseklik h olan bir ağaç en fazla 2^h yaprak tutar.)

Egzersiz 2. Bölme yöntemi $h(k) = k \bmod m$ için, $m = 2^b$ (ikinin kuvveti) seçmek neden kötüdür? (İpucu: yalnızca son b bit kullanılır.)

Egzersiz 3. Universal hash $h(k) = ((a \cdot k + b) \bmod p) \bmod m$'de $a = 0$ neden yasaktır? Anahtar bilgisi nasıl kaybolur?

Egzersiz 4. Python'da deterministik hash'in çakışmasını gözlemle: aynı `hash(k) % m` değerine düşen anahtarlar üret ve zincir uzunluğunu ölç.

```
m = 1024
buckets = [0] * m
for k in range(100_000):
    buckets[hash(k) % m] += 1
print("max zincir:", max(buckets), "ort:", sum(buckets) / m)
```

Egzersiz 5. Beklenen zincir uzunluğu $1 + (n-1)/m$'yi kullanarak, $m = n$ seçildiğinde find işleminin beklenen süresini hesapla. $m = n/2$ olursa ne değişir?

12.15 Sonraki Ders İçin Hazırlık

Ders 5 (L5): Doğrusal Zamanlı Sıralama

Jason Ku ile, karşılaştırma modelinin $\Omega(n \log n)$ sıralama sınırını **aşan** sıralamalara geçiyoruz: counting sort ve radix sort, küçük tam sayı anahtarlarda $O(n)$. Hashing'deki "anahtarı doğrudan indeks yap" fikri burada da işe yarayacak. (Not: ders akışında araya **Problem Oturumu 2** girer.)

Ders 5 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle Egzersiz 4'ü (çakışma ölçümü) çöz.
- $E[X_i] = 1 + (n-1)/m$ türetimini gösterge RV + doğrusallıkla yeniden yaz.
- Ana cümleyi tekrar oku: *"Hash fonksiyonunu rastgele seçmek, en kötü durumu beklenen $O(1)$'e dönüştürür."*

12.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Karşılaştırma modeli	Anahtarları yalnızca kıyaslayan model; arama $\Omega(\log n)$	Böl. 3

Kavram	Tanım	Sayfada
Karar ağacı (decision tree)	Karşılaştırma algoritması; $n+1$ yaprak → yükseklik $\log n$	Böl. 3
Direct access array	Anahtar = indeks; $O(1)$ ama u kadar yer	Böl. 4
Anahtar evreni u	En büyük anahtar boyutu; $u < 2^w$ varsayımı	Böl. 5
Hash fonksiyonu	h: $0 \ldots u-1 \to 0 \ldots m-1$, $m \approx n$	Böl. 6
Çakışma / zincirleme	İki anahtar aynı indekse; chain ile çöz	Böl. 7
Bölme yöntemi	$h(k) = k \bmod m$; deterministik (kötü girdi var)	Böl. 8
Evrensel hashing	Rastgele h ailesi; çakışma olasılığı $\leq 1/m$	Böl. 9
Beklenen zincir	$E = 1 + (n-1)/m$; $m = \Theta(n)$ ile $O(1)$	Böl. 10

12.17 Builder ve OMSCS Bağlantıları

6 köprü

Hashing, ML ve sistem mühendisliğinin altındaki erişim ve olasılık dilini somutlaştırır — köprülerin özeti:

1. **Universal hashing** → Stat 110: gösterge RV, beklentinin doğrusallığı, randomized algoritma analizi — hepsi burada somutlaşır.
2. **Hash tablosu** → Python `dict`/`set`, her dilin map/HashMap'i: $O(1)$ erişimin iç mekanizması.
3. **Consistent hashing** → dağıtık sistemler (cache, sharding, yük dengeleme): anahtarları sunuculara dağıtma.
4. **Hash-flooding (DoS)** → güvenlik: deterministik hash saldırıya açık; rastgele seed bu yüzden standart oldu.
5. **Beklenen vs en kötü durum** → randomized algoritma disiplini: garantiyi "her girdi" yerine "ortalama girdi" üzerinden vermek (ML'de Monte Carlo / beklenen-değer analizi).
6. $u < 2^w$ **varsayımı** → düşük seviye: anahtarın bir word'e sığması, gerçek donanımda sabit-zaman erişimin koşulu.

❗ Tek bir şey alıp gideceksen

Karşılaştırma seni $O(\log n)$'de tutar; rastgele erişim + hash bunu $O(1)$'e indirir. Ama bedava değildir — hash fonksiyonunu *rastgele* seçmezsen, en kötü durum seni $O(n)$ zincire mahkûm eder. Universal hashing, olasılığı senin lehine çevirir.

13 Problem Oturumu 2

Ders 4-5'in uygulaması: master theorem + özyineleme ağacı, sonsuz dizide üstel sıçrama, augmentation ile veri yapısı tasarımı ve two-finger ile O(n)

Oturum bilgisi

- **Solomon'un videosu:** YouTube — Problem Session 2 (≈60 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 2
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 6 (Problem Oturumu 2)
- **Hoca:** Justin Solomon
- **Okuma süresi:** ≈26 dk

13.1 Bu Problem Oturumu Ne Hakkında?

İkinci problem oturumu, son haftanın konularını uygular: **yineleme çözme** (master theorem + özyineleme ağacı), **arama** ve **veri yapısı tasarımı**. Justin Solomon dört problemi birer "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işler. Her problemin kazandırdığı taşınabilir araç Şekil 13.1 içinde özetlenir.

Bir genel beceri tüm oturuma damga vurur: problemler çoğu zaman "saçma" bir hikâyeyle (sonsuz taşlar, tuğla üfleyen kurt) sarmalanır; işin ilk adımı **işe yarayan iki cümleyi çıkarmaktır**. Solomon bunun bir danışmanlık becerisi olduğunu söyler.

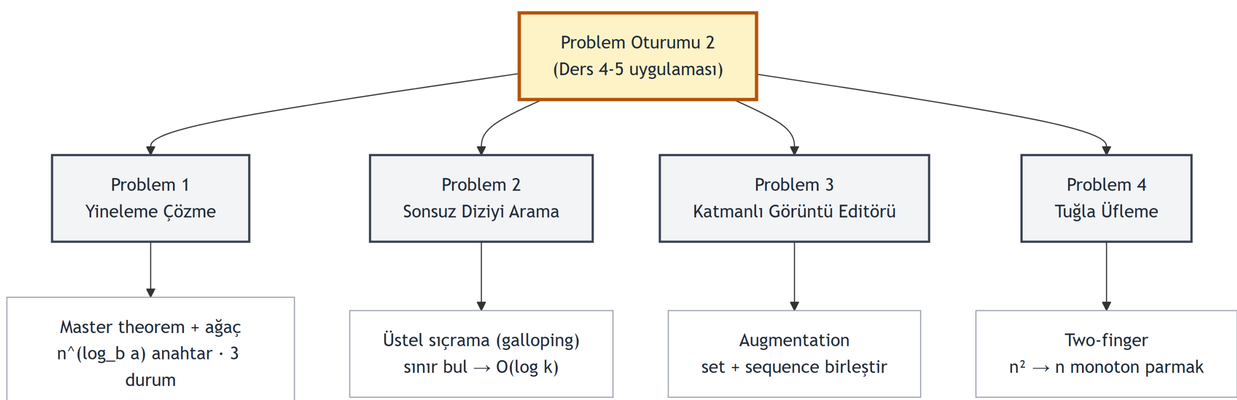
> *"extracting the two words that matter."* — Justin, 1:09:29

İkinci bir uyarı asimptotik gösterim üzerinedir: O, Θ, Ω harflerini özensiz kullanmak en sık yapılan hatadır.

> *"every single time you write one of those letters down, you should step back 50 feet and look at it and think… did I do that right?"* — Justin, 11:54

13.2 Problem 1: Yineleme Çözme — Master Theorem ve Özyineleme Ağacı

İfade. $T(n) = a \cdot T(n/b) + f(n)$ biçimindeki yinelemeleri çöz. Her birini **iki** yolla yap: (1) master theorem (kural seti), (2) özyineleme ağacı çizip seviye seviye işi toplayarak.



Şekil 13.1: Problem Oturumu 2'nin kavram haritası: dört problem (üst sıra) ve her birinin öğrettiği taşınabilir araç (alt sıra). Problem 1 master theorem + özyineleme ağacı (n^(log_b a) anahtar nicelik), Problem 2 üstel sıçrama (galloping) ile O(log k) arama, Problem 3 augmentation (set + sequence birleştirme), Problem 4 two-finger ile $n^2 \to n$ indirgemesi kazandırır. Hepsi Ders 4-5 temeline dayanır.

Yaklaşım — Anahtar Niceliği Hesapla, f ile Kıyasla

Master theorem üç duruma ayrılır; anahtar nicelik $n^{\log_b a}$'dır (a = dallanma sayısı, b = problemin küçülme oranı, f = düğüm başına iş). Bu niceliğin üssünü ($\log_b a$) hesapla, f'in üssüyle kıyasla, duruma karar ver. Özyineleme ağacı aynı sonucu daha yavaş ama **aydınlatıcı** biçimde verir; master theorem ise hızlıdır ama "neden"i göstermez.

> *"the a is kind of like a branching factor… b is the amount that your problem size reduces… f is the amount of work that you do at each node."* — Justin, 6:14

Çözüm. Üç durum:

- **Durum 1:** $f(n) = O(n^{\log_b a - \epsilon})$, $\epsilon > 0 \to T(n) = \Theta(n^{\log_b a})$. ($f$ önemsiz; ağaçta dolaşma baskın.)
- **Durum 2:** $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \to T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$.
- **Durum 3:** $f(n) = \Omega(n^{\log_b a + \epsilon})$ ve düzenlilik ($a \cdot f(n/b) \le c \cdot f(n), c < 1) \to T(n) = \Theta(f(n))$. ($f$ baskın.)

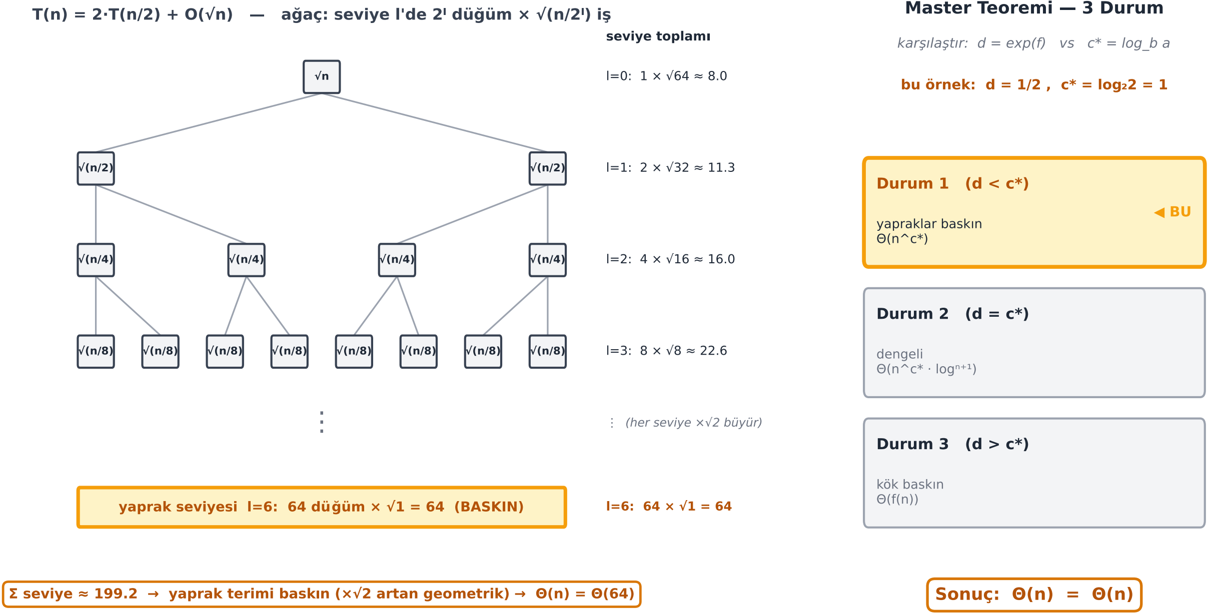
Örnek (a): $T(n) = 2T(n/2) + O(\sqrt{n})$. $a = b = 2$, $n^{\log_2 2} = n$. $f = O(\sqrt{n}) = O(n^{1-1/2})$, yani $\epsilon = 1/2 > 0 \to$ **Durum 1** $\to T(n) = \Theta(n)$.

Örnek (b): $T(n) = 8T(n/4) + O(n^{3/2})$. $\log_4 8 = 3/2$, yani $n^{3/2}$. $f = O(n^{3/2})$, iki taraf eşit $\to$ **Durum 2**, $k = 0 \to T(n) = O(n^{3/2} \log n)$. ($f$ big-O olduğundan sonuç da big-O, big-Θ değil.)

Özyineleme ağacı (ikinci yöntem): Ağaçta l. seviyede 2^l (veya 8^l) düğüm var, her biri $f(n/2^l)$ iş yapar; seviye sayısı $\log_b n$. Tüm seviyeleri topla — ortaya bir **geometrik seri** çıkar; sadeleştirince master theorem'le aynı sonuç gelir. Bu iki okumanın görsel karşılaştırması Şekil 13.2'de gösterilir: sol panel ağaç + seviye toplamları, sağ panel 3-durum kararı.

> *"master theorem… the giant sledgehammer for solving recurrences without understanding why you got the answer."* — Justin, 4:09

Karmaşıklık. Sonuçlar: (a) $\Theta(n)$, (b) $O(n^{3/2} \log n)$.



Şekil 13.2: Master teoremini **özyineleme ağacıyla** okuma: $T(n) = 2T(n/2) + O(\sqrt{n})$ ($n = 64$). **Sol panel:** kök ($l = 0$) tek alt-problem $\sqrt{n}$; her seviyede düğüm sayısı ikiye katlanır (2^l) ve her düğümün işi $\sqrt{n/2^l}$'e iner. **Seviye toplamı** $2^l \cdot \sqrt{n/2^l}$ aşağı indikçe $\sqrt{2}$ çarpanıyla *büyür* — $8 \to 11{,}3 \to 16 \to 22{,}6 \to 32 \to 45{,}3 \to 64$ — yani artan bir **geometrik seri**dir ve **yapraklar** ($l = 6$, **amber/BASKIN**) baskın terimi verir: 64 düğüm $\times\sqrt{1} = 64 = \Theta(n)$. Tüm seviyelerin toplamı $\approx$ 199,2, ama asimptotik olarak yaprak terimi belirleyicidir $\to \Theta(n)$. **Sağ panel:** 3-durum karar kuralı — $f(n)$'in üssü $d = \exp(f) = \frac{1}{2}$ ile kritik üs $c^* = \log_b a = \log_2 2 = 1$ karşılaştırılır. $d < c^*$ olduğundan **Durum 1** seçilir (yapraklar baskın, $\Theta(n^{c^*})$); Durum 2 ($d = c^*$, dengeli) ve Durum 3 ($d > c^*$, kök baskın) burada uygulanmaz. Sonuç: $\Theta(n)$.

13.3 Problem 2: Sonsuz Diziyi Arama

İfade. Sonsuz bir gezegen dizisinde, indeksi k olan gezegeni bul. Bir gezegende durup oracle'a sorabilirsin: "aradığım indeks benden büyük mü küçük mü?" Hedef: $O(\log k)$ zaman (dikkat: k, verinin boyutu değil, aranan indeks).

Yaklaşım — Önce Sağ Sınırı Bul, Sonra İkili Ara

İçgüdü ikili arama; ama ikili arama bir **sağ sınır** ister ve sonsuz dizide sağ sınır yok. Önce bir sağ sınır *bulmalıyız* — sonsuz bir kümenin "ortası" yoktur, bu yüzden indeksi her adımda ikiye katlayarak (üstel sıçrama) hedefi aşan ilk gücü buluruz.

> *"what is the middle of an infinite set of planets? Infinite, exactly. It's a problem."* — Justin, 49:26

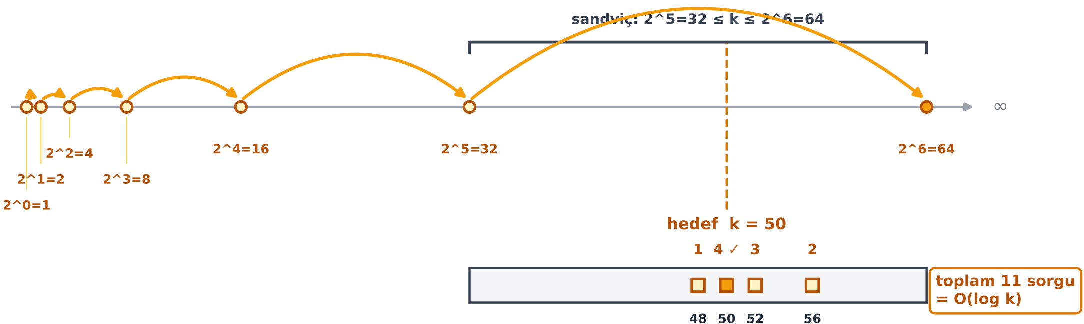
Çözüm. İki aşama:

1. **Sınır bul (üstel sıçrama):** $m = 0, 1, 2, \ldots$ için 2^m indeksli gezegeni ziyaret et, ta ki $k \leq 2^m$ olana dek. Bu, $\log k$ adım sürer ve şu sandviçi verir: $2^{m-1} \leq k \leq 2^m$.
2. **İkili arama:** Artık $[2^{m-1}, 2^m]$ aralığı var (genişliği $\sim k$); bu aralıkta ikili arama, $\log k$ adım.

> *"I'm going to visit planet 2 to the m for each m starting with m equals 0."* — Justin, 50:25

Şekil 13.3 bu iki aşamayı tek bir sayı-doğrusunda izler: önce galloping yaylarıyla sınır, sonra numaralı mid sorgularıyla ikili arama.

Karmaşıklık. $\log k$ (sınır bulma) + $\log k$ (ikili arama) = $O(\log k)$.



Şekil 13.3: Sonsuz diziyi arama — iki aşamalı O(log k) strateji (hedef indeks $k = 50$). **AŞAMA 1 (üstel sıçrama / galloping):** sağ sınır bilinmediği için indeksi her adımda **ikiye katlayarak** $2^0, 2^1, \ldots, 2^m$ sorgulanır (amber yaylar: $1, 2, 4, 8, 16, 32, 64$). İlk $2^6 = 64 \geq k$ bulununca durulur — toplam **7 sorgu** — ve bu, $2^5 = 32 \leq k \leq 2^6 = 64$ **sandviçini** verir (üstteki köşeli parantez). Hedef $k = 50$ amber kesik dikey çizgiyle işaretlidir. **AŞAMA 2 (ikili arama):** artık sonlu $[32, 64]$ aralığında klasik ikili arama yapılır; numaralı mid sorguları sırasıyla $48 \to 56 \to 52 \to 50$ (4 sorgu, ✓ = bulundu). Her iki aşama da $O(\log k)$ olduğundan toplam **11 sorgu** $= O(\log k)$ (rozet sağ altta) — $\log_2 50 \approx 5{,}64$. Anahtar sezgi: sınır galloping ile $O(\log k)$'da bulunur, sonra ikili arama yine $O(\log k)$'da bitirir.

13.4 Problem 3: Katmanlı Görüntü Editörü

İfade. Bir görüntü editörü veri yapısı tasarla: katmanları üst-alt sırada tutar. İşlemler: `make_doc` $O(1)$, `import(x)` (üste ekle) $O(n)$, `display()` (sırayla tüm ID'ler) $O(n)$, ve **`move_below(x, y)`** (x katmanını y'nin altına taşı) $O(\log n)$.

Yaklaşım — İki Yönü Ayır, Sonra Augmentation ile Bağla

Problemde iki yön var: **dizi (sequence)** yönü (display sırası) ve **küme (set)** yönü (bir katmanı hızlı bulmak). İkisini ayrı veri yapısıyla çöz, sonra bir **augmentation** ile bağla — sıralı dizideki her anahtara, bağlı listedeki düğümün işaretçisini iliştir. Böylece “bul” $O(\log n)$, “yeniden bağla” $O(1)$ olur.

Çözüm. İki yapı birlikte çalışır:

- **Çift bağlı liste (doubly linked list):** katmanların ekran sırasını tutar (display için $O(n)$).
- **Sıralı dizi (set):** ID’leri sıralı tutar; ikili aramayla bir ID’yi $O(\log n)$’de bulur.
- **Augmentation:** sıralı dizideki her anahtara, o öğenin **bağlı listedeki düğümüne bir işaretçi** ekle. (Bir düğümü silmek diğer işaretçileri bozmaz — işaretçiler geçerli kalır.)

> *“we can attach data to our keys… I am going to attach a pointer into my linked list.”* — Justin, 1:02:21

`move_below(x, y)` üç adım: (1) x ve y’yi sıralı dizide ikili aramayla bul, bağlı liste işaretçilerini al — $O(\log n)$; (2) x’i bağlı listeden çıkar ve y’nin altına ekle — işaretçiler elimizde olduğundan $O(1)$; (3) küme değişmez (sadece sıra değişti). Önemli sağlama: `move_below` kümeyi değiştirmez, çünkü hangi öğelerin olduğu değil, yalnızca sıraları değişir.

Karmaşıklık. `make_doc` $O(1)$, `import` $O(n)$ (sıralı diziye ekleme), `display` $O(n)$, `move_below` $O(\log n)$ (bulma) + $O(1)$ (yeniden bağlama).

13.5 Problem 4: Tuğla Üfleme

İfade. Bir sıra evin tuğla sayıları verilir. Kurt bir eve üfleyince o ev *ve sağındaki* kendisinden az tuğlalı tüm evler yıkılır. Her ev için, ona üflendiğinde kaç ev yıkıldığını hesapla. (Hikâyenin özü: “her öğe için, sağında kendisinden küçük olanları say + 1”.)

Yaklaşım — Naif Çift Döngüden Monoton Parmaklara

Naif çözüm çift döngü → $O(n^2)$; yasak. Bir yapısal varsayım kullanılır: bir ev hariç hepsi **özel** (sağ komşusu kendisinden büyük-eşit), yani dizi **artan-sonra-bir-düşüş-sonra-artan** iki sıralı parçaya bölünür. Sol yarı (artan) sağa hiçbir şey yıkamaz; her ev yalnızca sağ yarıda kendinden küçükleri yıkar. Sol yarıdaki ev büyüdükçe yıkılan aralık yalnız *genişler* — geri gitmez — ve bu, two-finger’a kapı açar.

Çözüm. Sol yarıdaki ev tuğla sayısı arttıkça, sağ yarıda yıkılan evlerin aralığı yalnızca **büyür** (geri gitmez). Bu, **iki parmak (two-finger)** tekniğine kapı açar: `i` parmağı sol yarıda ilerlerken, `j` parmağı sağ yarıda “ilk yıkılmayan” evi işaret eder; `i` sağa kaydıkça `j` de sağa kayar, asla sola dönmez.

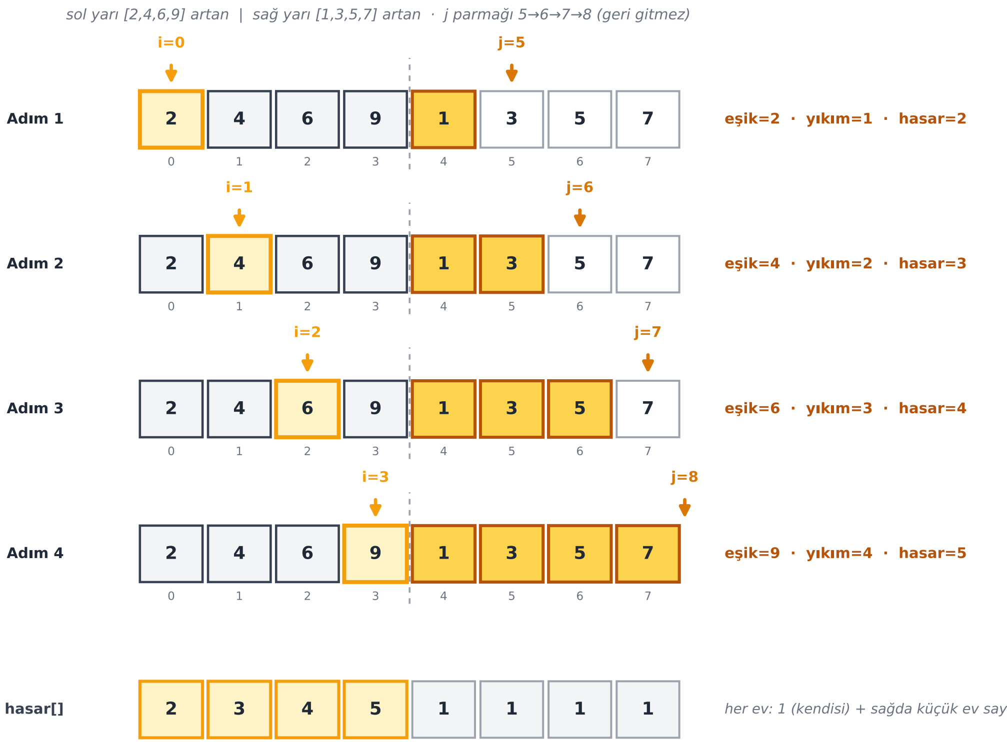
> *“It’s called a two-finger algorithm.”* — Justin, 1:25:02

```
j = right_start                                # sağ yarının ilk evi (global indeks)
for i in range(left_end + 1):
    while j < n and heights[j] < heights[i]:
        j += 1                                 # j asla geri gitmez (5→6→7→8)
```

```
    smaller = j - right_start                # sağda i'den küçük ev sayısı
    damage[i] = 1 + smaller                  # kendisi + sağda küçük ev sayısı
```

Her iki parmak da diziyi yalnızca bir kez kateder → $O(n)$. (Genel hâl, özel varsayım olmadan, aynı fikir + merge sort ile çözülür.) Şekil 13.4 bu izi $[2, 4, 6, 9]$ sol yarısı ile $[1, 3, 5, 7]$ sağ yarısı üzerinde adım adım gösterir; `j` parmağının $5 \to 6 \to 7 \to 8$ ile yalnız sağa kaydığına dikkat.

Karmaşıklık. Two-finger ile $O(n)$ (n^2 naif veya $n \log n$ ikili-arama yerine).



Şekil 13.4: İki parmak (two-finger) tuğla üfleme izi — Problem 4 İMZA figürü. Dizi tek düşüşten (9→1) ikiye ayrılır: **sol yarı** [2,4,6,9] artan, **sağ yarı** [1,3,5,7] artan. Sol parmak `i` sol yarıyı soldan sağa sweep eder (amber, eşik = `heights[i]`); sağ parmak `j` sağ yarıda eşikten küçük olmayan **ilk** evi işaretler. `i` sağa kaydıkça eşik büyür, bu yüzden `j` yalnız **sağa** kayar — asla geri gitmez (5→6→7→8). **Adım 1** (eşik 2): sağda yalnız 1 küçük → hasar 2. **Adım 2** (eşik 4): 1,3 küçük → hasar 3. **Adım 3** (eşik 6): 1,3,5 → hasar 4. **Adım 4** (eşik 9): 1,3,5,7 → hasar 5. Yıkılan sağ-yarı evleri her satırda koyu amber. İki parmak diziyi birer kez katettiği için toplam **O(n)** — naif "her ev için sağı tara" $O(n^2)$ yerine. Sağ yarının kendi içi de artan olduğundan oradaki her ev yalnız kendini yıkar (hasar 1); nihai `hasar[]` = [2,3,4,5,1,1,1,1].

13.6 Ne Öğrendik?

! Altı Taşınabilir Araç

Bu oturum, Ders 4-5'in teorisini dört somut problemde uyguladı ve altı taşınabilir araç kazandırdı:

1. **Master theorem + özyineleme ağacı:** Yinelemeyi iki yolla çöz; $n^{\log_b a}$ anahtar niceliktir, ağaç "neden"i gösterir.
2. **Asimptotik titizlik:** O / Θ / Ω'yu özensiz yazma; f big-O ise sonuç da big-O'dur.
3. **Üstel sıçrama (galloping):** Sağ sınırı olmayan aramada, 2^m ile sınır bul, sonra ikili ara — $O(\log k)$.
4. **Augmentation ile iki yapı birleştirme:** set (ID bul) + sequence (sıra tut); anahtara liste işaretçisi ekle.
5. **Two-finger ile $n^2 \to n$:** Monoton ilerleyen iki parmak, çift döngüyü lineere indirir.
6. **Problem okuma:** Süslü hikâyeden "işe yarayan iki cümleyi" çıkarmak — algoritmik özü görmek.

13.7 Sonraki

Ders 5 (L5) — Doğrusal Zamanlı Sıralama

Sırada **Ders 5 (L5): Doğrusal Zamanlı Sıralama** var — Jason Ku ile, karşılaştırma modelinin $\Omega(n \log n)$ sınırını aşan sıralamalara (counting sort, radix sort) geçiyoruz. Bu oturumdaki **master theorem** ve **two-finger** sezgileri, oradaki doğrusal-zaman analizinde doğrudan işine yarayacak.

14 Doğrusal Zamanlı Sıralama

Karşılaştırma alt sınırı Ω(n log n), direct access array sort, tuple/Excel sort, kararlı sıralama, counting sort ve radix sort O(n)

ℹ Bölüm bilgisi

- **Ku'nun videosu:** YouTube — Lecture 5: Linear Sorting (≈52 dk)
- **OCW sayfası:** MIT 6.006 Lecture 5: Linear Sorting
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 7 (L5)
- **Hoca:** Jason Ku (Erik Demaine, Justin Solomon)
- **Okuma süresi:** ≈25 dk

14.1 Bu Derste Ne Var?

Ders 4'te merge sort, bir diziyi $O(n \log n)$'de sıraladı. Bu ders iki soru sorar: (1) **karşılaştırma** modelinde daha hızlı olabilir miyiz — **hayır**, $n \log n$ optimaldir; (2) modeli genişletirsek — **evet**, anahtarlar küçük tam sayıysa $O(n)$ sıralama mümkündür.

Üç temel kavram bu derste yan yana gelir:

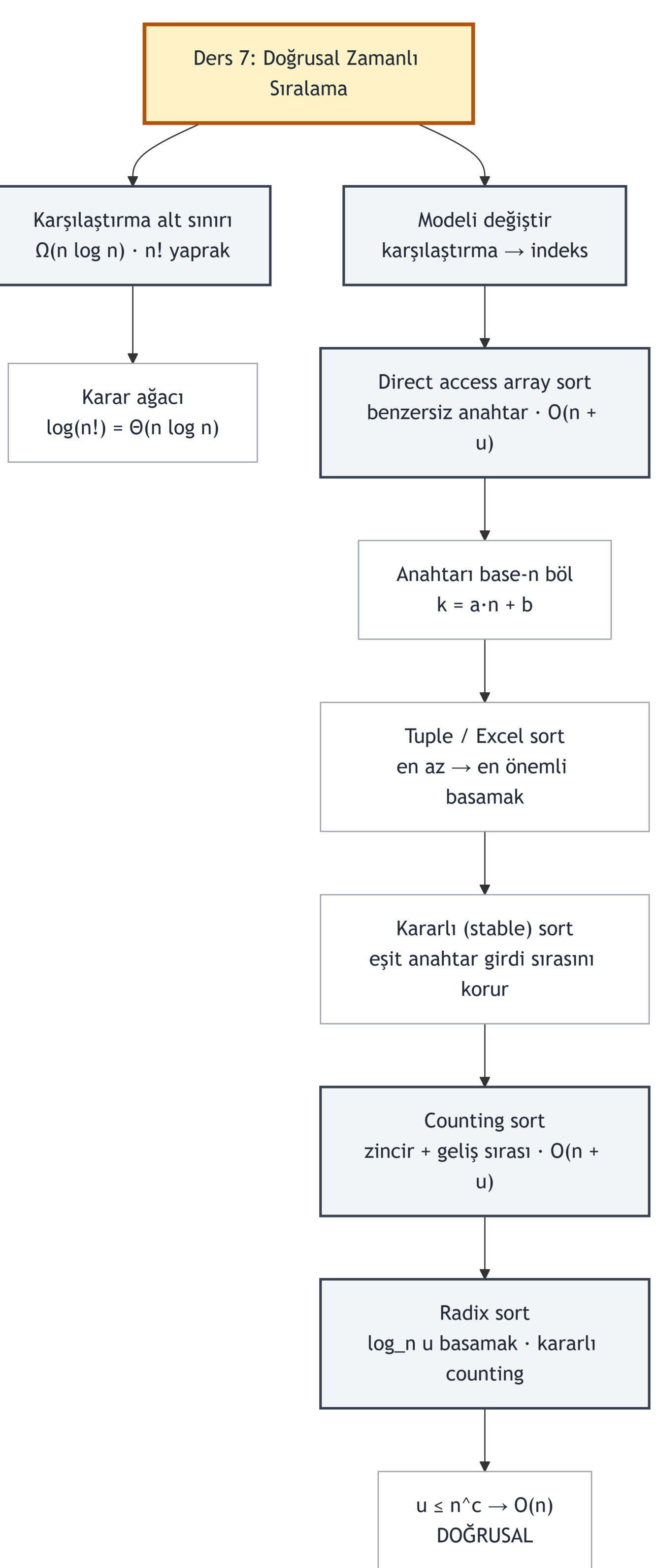
1. **Sıralama alt sınırı** — karşılaştırma modelinde her sıralama $\Omega(n \log n)$'dir ($n!$ yaprak → karar ağacı argümanı).
2. **Direct access array sort → counting sort** — anahtarı doğrudan indeks yap; çakışmaları zincirle, geliş sırasını koru.
3. **Radix sort** — büyük anahtarları base-n basamaklara ayır, kararlı (stable) sıralamayla basamak basamak sırala.

> *"n log n is optimal."* — Ku, 10:19

Builder Notu — Counting Sort = Hash'in Sıra-Koruyan İkizi

Bu dersin çekirdeği, Ders 5'teki (Hashing) "anahtar = indeks" fikrinin **arama** yerine **sıra** için yeniden kullanılmasıdır:

- **Geriye → Ders 5 (Hashing):** Counting sort'un "her indekste bir zincir" yapısı, hash tablosunun çakışma çözümüyle (chaining) birebir aynı veri yapısıdır — fark, amacın *arama* değil *sıra korumak* olmasıdır. Hash'te zincir sırası önemsizdi; burada zincire **geldikleri sırada** ekleme, kararlılığın (stability) ta kendisidir.



Şekil 14.1: Ders 7'nin kavram haritası: karşılaştırma modelinin Ω(n log n) alt sınırından (n! yaprak → karar ağacı) modeli değiştirmeye — direct access array sort (benzersiz anahtar) → counting sort (zincir + geliş sırası, kararlı) → radix sort (base-n basamak + kararlı counting sort) → u ≤ n^c koşulunda O(n) doğrusal sıralama.

- **Geriye → PS1 (Stirling):** $\log(n!) = \Theta(n \log n)$ alt sınırı, problem setindeki Stirling yaklaşımının doğrudan uygulamasıdır; $\log(10^9!) \approx$ milyarlarca karşılaştırma mertebesi.
- **İleriye → büyük veri:** radix sort, tam sayı / sabit-uzunluk anahtarlarda (IP adresleri, timestamp, sabit ID) karşılaştırmalı sıralamayı geçer; veritabanı ve GPU sıralamasında standart.
- **İleriye → çok-kolonlu sıralama:** "Excel spreadsheet sort" — kararlı sıralamayı en az önemliden en önemliye uygulamak, SQL `ORDER BY a, b` ve pandas `sort_values` mantığının ta kendisidir.

Tek cümle: *Karşılaştırma seni $n \log n$'de tutar; ama anahtarlar küçük tam sayıysa, onları doğrudan indeks yapıp basamak basamak kararlı sıralayarak $O(n)$'e inebilirsin.*

14.2 Önceki Ders: Hash Tablosunun Sınırı

Hash tablosu find/insert/delete'i *beklenen* $O(1)$ yapar — ama **en kötü durumda** $O(n)$'dir (kötü hash → tüm anahtarlar tek zincirde). Bu yüzden en-kötü-durum garantisi gerektiğinde (örneğin worst-case isteyen problem setlerinde) hash tablosu uygun değildir; sıralı dizi ($O(\log n)$ garantili) tercih edilir. (Java, zincirleri ağaçla tutarak worst-case $O(\log n)$ elde eder.) Bu ders aramayı değil, **sıralamayı** ele alır.

14.3 Sıralama Alt Sınırı: n log n

Ders 5'teki karar ağacı argümanını sıralamaya uygularız. Bir sıralama algoritmasının çıktısı, girdinin bir **permütasyonudur**: ilk öğe nereye, ikinci nereye, … Her öğe için bağımsız seçim → toplam $n!$ olası çıktı.

> *"I need at least n factorial leaves."* — Ku, 13:55

Çalışılan Örnek — $n \log n$ alt sınırı. Doğru bir sıralama, $n!$ permütasyonun her birini üretebilmeli → karar ağacında en az $n!$ **yaprak**. İkili ağacın minimum yüksekliği $\log(n!)$. Bunu alttan sınırla: $n!$ çarpımındaki n terimden yarısı ($n/2$ terim) en az $n/2$'dir, dolayısıyla $n! \geq (n/2)^{n/2}$. Logaritması:

$$\log(n!) \geq \frac{n}{2} \cdot \log \frac{n}{2} = \Theta(n \log n)$$

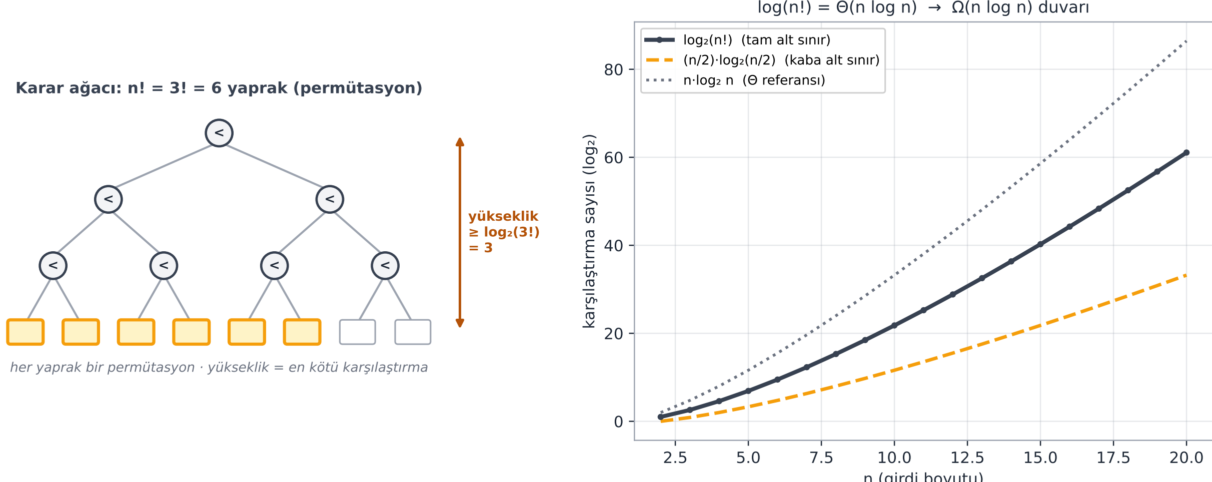
> *"any sorting algorithm here takes at least n log n comparisons, and so a merge sort's the best we can do."* — Ku, 16:17

Demek ki karşılaştırma modelinde $\Omega(n \log n)$ alt sınırı vardır; merge sort optimaldir. (PS1'deki Stirling yaklaşımı da aynı sonucu verir.) Daha hızlısı için karşılaştırmadan çıkıp **rastgele erişim** kullanmak gerekir. Şekil 14.2 bu argümanı küçük bir karar ağacı ve $\log(n!)$ büyüme eğrisiyle birlikte gösterir.

Builder Notu — n log n Duvarı ve Model Değiştirme

$\Omega(n \log n)$ bir **algoritma** sınırı değil, bir **model** sınırıdır — ve bu ayrım bir builder için kritiktir:

- **Sınır neye bağlı:** Alt sınır "yalnız karşılaştırma yapan" algoritmalar için geçerlidir. İki öğeyi karşılaştırmak en çok 1 bit bilgi verir; $n!$ olası çıktıyı ayırt etmek için en az $\log_2(n!) = \Theta(n \log n)$ bit, yani o kadar karşılaştırma gerekir. Merge sort bu sınıra ulaştığı için karşılaştırma dünyasında **optimaldir**.



Şekil 14.2: Karşılaştırma modelinde sıralamanın alt sınırı: doğru bir sıralama, girdideki n öğenin **herhangi** bir sıralanışını düzeltebilmeli — yani $n!$ olası permütasyonun her birini üretebilmelidir. Algoritmayı bir **karar ağacı** olarak düşünün: her iç düğüm bir karşılaştırma ($<$), her yaprak nihai bir permütasyon. **Sol:** $n = 3$ için ağaç — $3! = 6$ yaprak (amber) gerekir; ikili bir ağacın h yükseklikte en fazla 2^h yaprağı olduğundan $2^h \geq n!$, yani yükseklik $\geq \lceil \log_2(3!) \rceil = 3$. Bu yükseklik en kötü durumdaki karşılaştırma sayısıdır. **Sağ:** $\log_2(n!)$ (tam alt sınır, $\sum_{i=1}^{n} \log_2 i$) ile kaba alt sınır $(n/2)\log_2(n/2)$ ve $n \log_2 n$ referansı aynı büyüme bandında ilerler — Stirling ile $\log_2(n!) = n\log_2 n - n\log_2 e + \frac{1}{2}\log_2(2\pi n) = \Theta(n \log n)$. Sonuç: yalnız karşılaştırmaya dayanan **hiçbir** sıralama $\Omega(n \log n)$'den hızlı olamaz — bu, doğrusal zamanın önündeki $n \log n$ **duvarıdır**. Duvarı yıkmanın tek yolu karşılaştırmayı bırakıp anahtarı doğrudan adres olarak kullanmaktır (counting / radix sort).

- **Tek çıkış = modeli değiştirmek:** Daha hızlı olmak istiyorsan karşılaştırmayı bırakıp anahtarı **doğrudan adres** olarak kullanmalısın (rastgele erişim). Counting/radix sort tam da bunu yapar — bu yüzden alt sınırı ihlal etmezler, *dışına çıkarlar*.
- **Genel ders:** Bir alt sınıra çarptığında soru "daha akıllı bir algoritma var mı?" değil, "varsayımlardan hangisini gevşetebilirim?" olur. Burada gevşetilen varsayım "anahtarlar yalnız karşılaştırılabilir"; yerine "anahtarlar küçük tam sayı" konur.

14.4 Karşılaştırmanın Ötesi: Direct Access Array Sort

Anahtarlar **benzersiz (unique)** ve **küçük aralıkta** ise: büyük bir direct access array kur, her öğeyi `x.key` indeksine koy ($O(1)$), sonra diziyi baştan sona gez ve dolu hücreleri sırayla topla.

> *"if u is on the order of n, then we now have linear time sorting algorithm."* — Ku, 21:56

```
def daa_sort(A, u):
    D = [None] * u
    for x in A:
        D[x.key] = x          # benzersiz anahtar -> cakisma yok
    out = []
    for slot in D:            # 0..u-1 sirayla gez
        if slot is not None:
            out.append(slot)
    return out
```

Karmaşıklık: diziyi kurmak + gezmek $O(u)$, n öğe eklemek $O(n)$ → toplam $O(n+u)$. $u = \Theta(n)$ ise **lineer**. Ama iki kısıt var: anahtarlar benzersiz ve aralık küçük olmalı.

14.5 Daha Büyük Aralık: Anahtarı Basamaklara Ayır

$u \leq n^2$ olsun. n^2-boyutlu bir dizi quadratik olur — işe yaramaz. Çözüm: her anahtarı **base-n** iki basamağa ayır. k için:

$$a = k//n, \quad b = k \bmod n, \quad k = a \cdot n + b$$

Burada a **n'ler basamağı**, b **birler basamağı**dır. Her basamak $0..n-1$ aralığındadır. Örneğin $n = 5$ için $17 = (3,2)$ çünkü $3 \cdot 5 + 2 = 17$. Artık tam sayıları değil, bu **ikililer (tuple)** üzerinden sıralayacağız.

Builder Notu — Böl ve Eşle (radix/bucket/hash ortak kök)

Anahtarı base-n basamaklara ayırmak, yalıtık bir hile değil — bilgisayar biliminin tekrar tekrar gördüğü bir kalıptır: **büyük bir problemi sabit sayıda küçük parçaya böl, her parçayı bir kovaya eşle.**

- **Radix:** anahtarı basamaklara böl, her basamağı $0..n-1$ kovasına eşle (bu ders).
- **Bucket sort:** anahtar aralığını eşit kovalara böl, her öğeyi kovasına at, kova içini sırala — aynı

"böl ve eşle" sezgisi, farklı bölme kuralı.
- **Hash bölmeleri (partitioning):** dağıtık sistemlerde (Spark, veritabanı sharding) anahtarı `hash(k) mod p` ile p parçaya böl — her parça bağımsız işlenir.

Üçü de $k = a \cdot n + b$ türünden bir ayrıştırmayla "tek büyük indeks" yerine "sabit sayıda küçük indeks" kullanır. Bir builder bu kalıbı tanırsa, "anahtar evreni çok büyük" probleminde refleks olarak basamaklara/kovaları bölmeyi dener.

14.6 Tuple Sort (Excel Tablo Sıralaması)

İkilileri sıralamak, bir **Excel tablosunu kolonlara göre** sıralamaya benzer: her kolonun bir öncelik sırası var. En önemli basamak (a), en az önemliden (b) daha belirleyicidir — a'yı 1 artırmak, b ne olursa olsun sayıyı büyütür.

> *"tuple sort, but you can also think of it as Excel spreadsheets sort."* — Ku, 29:14

Strateji: basamakları **en az önemliden en önemliye** doğru, art arda sırala. Ama bu yalnızca her geçişte **kararlı (stable)** bir sıralama kullanılırsa işe yarar — sonraki bölümün konusu. Şekil 14.3, 17, 3, 24, 22, 12 örneğinde iki kararlı geçişi yan yana gösterir.

14.7 Kararlı Sıralama (Stable Sort)

Bir sıralama **kararlıdır** eğer eşit anahtarlı öğeler, çıktıda girdideki **göreli sıralarını korur**.

> *"if they are the same thing, then the output maintains their order from the input… that's what we call a stable sorting algorithm."* — Ku, 36:18

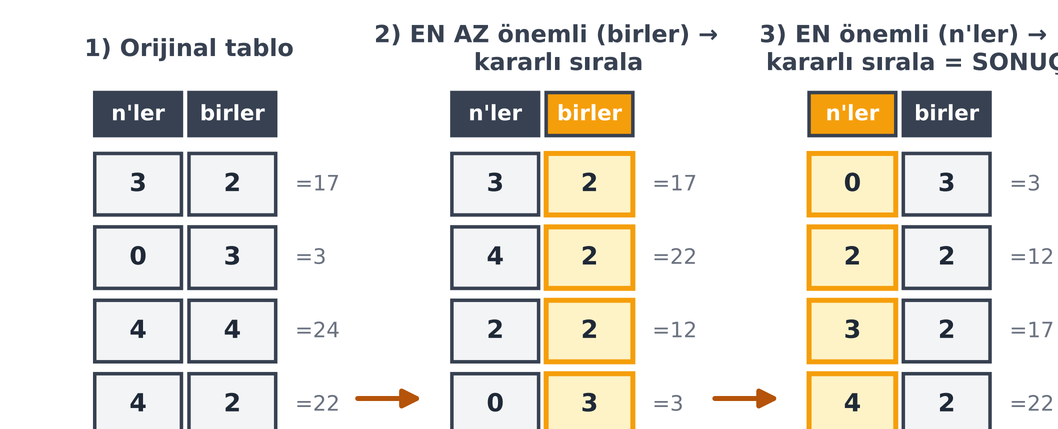
Neden kritik? Tuple sort'ta önce en az önemli basamağı sıralarız; sonra en önemliyi. En önemli basamak eşit olan öğeler için, *önceki* (az önemli) sıralamanın korunması gerekir — bu yalnızca kararlı sıralamayla olur. Şekil 14.4, aynı girdiyi kararlı ve kararsız sıralayarak farkı görselleştirir.

Builder Notu — Kararlılık ve Çok-Kolonlu Sıralama (SQL ORDER BY)

Kararlılık akademik bir incelik değil, her gün kullandığın çok-kolonlu sıralamanın temelidir:

- **SQL `ORDER BY a, b`:** Bunu doğru gerçekleştirmenin bir yolu, önce **ikincil** anahtar `b`'ye göre kararlı sırala, sonra **birincil** anahtar `a`'ya göre kararlı sırala. `a` eşit olan satırlar `b` sırasını korur — tuple sort'un birebir aynısı. Anahtar her zaman *az önemliden çok önemliye* uygulanır.
- **pandas `df.sort_values([...])` ve Excel'in çok-seviyeli sıralaması** aynı mantığı kullanır; altta yatan sıralama kararlı olmasaydı ikincil sıra rastgele bozulurdu.
- **Programlama dili garantisi:** Python'un `sorted`/`list.sort` ve Java'nın nesne sıralaması **kararlı** olmayı garanti eder (Timsort); C++ `std::sort` etmez (kararlı isterse `std::stable_sort`). Bir builder bu garantiyi bilmeden çok-kolonlu sıralama yazarsa, dilden dile sessizce farklı sonuçlar alır.

Tek cümle: *Kararlılık, "önce ikincil sonra birincil anahtar" hilesiyle çok-kolonlu sıralamayı tek-kolonlu*



Şekil 14.3: Tuple sort = Excel kolon sıralaması (Ku §5): bir sayıyı base-n basamaklarından oluşan bir **ikili** (a, b) gibi düşün (a = n'ler, b = birler). $17, 3, 24, 22, 12$ ($n = 5$, base-5) örneği iki **kararlı** geçişle sıralanır: önce **EN AZ önemli** basamağa (birler, amber kolon) göre kararlı sırala, sonra **EN önemli** basamağa (n'ler, amber kolon) göre kararlı sırala. Sonuç $3, 12, 17, 22, 24$. Kararlılık şart: 3. geçişte n'ler basamağı eşit olan $24 = (4, 4)$ ve $22 = (4, 2)$ ikilileri, 1. geçişte kurulan birler sırasını korur (22 önce, 24 sonra). Bu, radix sort'un counting sort'u neden **kararlı** yardımcı olarak kullandığının çekirdeğidir.

Şekil 14.4: **Kararlı (stable) sıralama**: eşit anahtarlı ama farklı yüklü (payload) öğeler için, kararlı bir sıralama bunların **girdideki göreli sırasını korur**; kararsız bir sıralama onları yeniden dizebilir. Üstte **girdi**: anahtarlar $3, 1, 3, 2, 1, 3$ (altlarındaki $a..f$ etiketleri girdi sırasını işaretler — eşit anahtarlı öğeleri birbirinden ayıran yük). Ortada **kararlı** çıktı (**amber** ✓): anahtar $= 3$ grubu $a \to c \to f$ sırasını AYNEN korur, anahtar $= 1$ grubu $b \to e$ kalır. Altta **kararsız** çıktı ($\times$): anahtarlar yine artan $(1, 1, 2, 3, 3, 3)$ ama eşit anahtar içinde sıra bozulur (3 grubu $f \to c \to a$). Anahtarlar sıralı olsa bile kararsız sonuç farklıdır — bu yüzden çok-kolonlu sıralama (önce ikincil, sonra birincil anahtar) ve radix sort yalnız **kararlı** bir alt-sıralama ile doğru çalışır.

sıralamalardan kurmanı sağlar.

14.8 Counting Sort

Direct access array sort benzersiz anahtar isterdi; ama tuple sort'ta basamaklar **tekrar eder**. Çözüm: her indekste tek öğe yerine bir **zincir** (sequence veri yapısı — dinamik dizi/bağlı liste) tut ve öğeleri **geldikleri sırada** sona ekle. Bu **counting sort**'tur — hashing'e benzer ama amaç sıra korumak.

> *"This is called counting sort."* — Ku, 39:41

Okurken, dolu zincirleri 0'dan $u-1$'e gez ve içlerini sırayla oku. Geliş sırası korunduğu için counting sort **kararlıdır**. Zincirlerin toplam uzunluğu n, dizi boyutu $u \to O(n+u)$. Şekil 14.5 bu zincir/histogram yapısını ve kararlılığı adım adım gösterir.

14.9 Radix Sort

Counting sort'u tuple sort'un yardımcı (kararlı) sıralaması olarak kullanırsak, büyük anahtarları sıralayabiliriz. **Radix sort**: en büyük anahtar u olan tam sayıları base-n basamaklara ayır ($\log_n u$ basamak), sonra counting sort ile en az önemliden en önemliye tuple sort yap.

> *"this is what we call radix sort."* — Ku, 47:39

Her basamağı sıralamak $O(n)$ (counting sort, $u = n$). Basamak sayısı $\log_n u$. Ayrıca her sayıyı ayırmak da $O(n \cdot \log_n u)$. Toplam:

$$T(n) = O(n + n \cdot \log_n u)$$

Kritik sonuç: $u \leq n^c$ (sabit c) ise, $\log_n u = \log_n n^c = c$ (sabit) → $T(n) = $ $O(n)$.

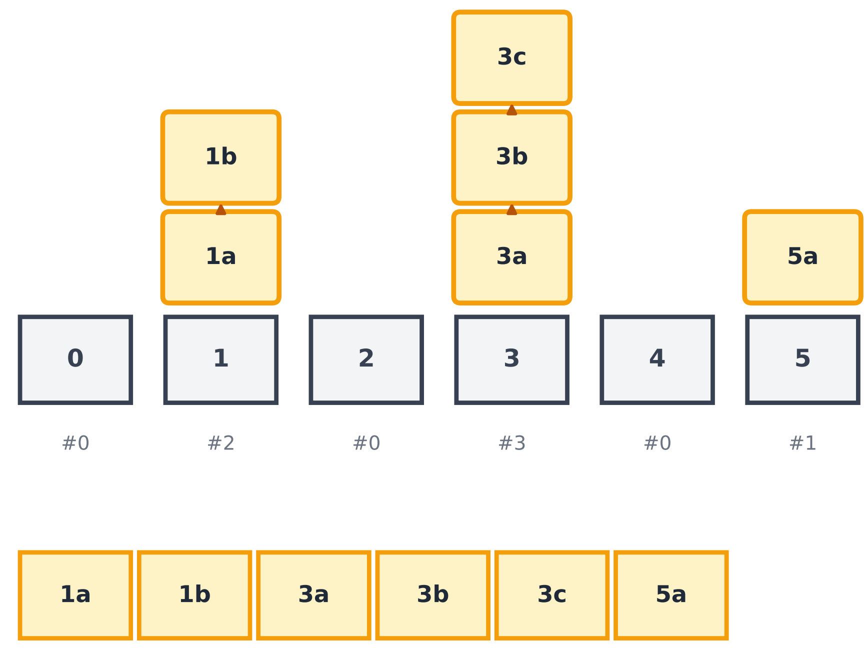
> *"if u is less than n to the c for some constant c… we get a linear time algorithm."* — Ku, 50:35

Yani anahtarlar n'in bir polinomuyla sınırlıysa, radix sort **lineer** zamanda sıralar. Şekil 14.6, $17, 3, 24, 22, 12$ üzerinde iki kararlı counting sort geçişini (LSD → MSD) gösterir.

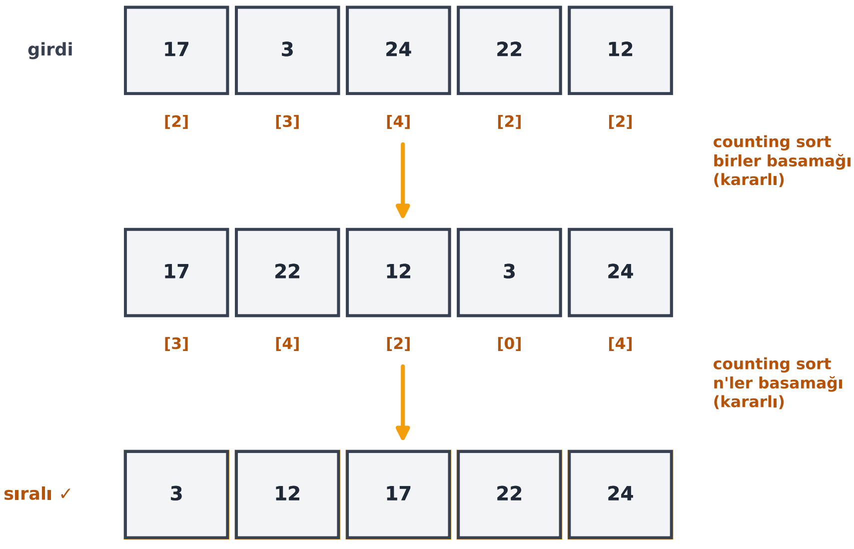
Builder Notu — Radix Sort, GPU ve Büyük Veri

Radix sort akademik bir oyuncak değil; **sabit-uzunluk tam sayı anahtarlar** söz konusu olduğunda gerçek dünyada karşılaştırmalı sıralamayı geçer:

- **Sabit-uzunluk anahtarlar her yerde:** 32-bit IP adresleri, 64-bit timestamp'ler, sabit ID'ler — hepsi $u \leq n^c$ koşulunu pratikte sağlar (örn. $u = 2^{32}$, $n = 10^6$ için $\log_n u \approx \log_{10^6} 2^{32} \approx 1.6$, yani 2 basamak yeter). Bu yüzden ağ/log işleme boru hatlarında radix sort doğal tercihtir.
- **GPU sıralama:** Radix sort, karşılaştırma yerine basamak sayımına (histogram) dayandığından **paralelleştirmesi kolaydır** — her thread bir öğenin basamağını sayar, prefix-sum ile yerini bulur. CUDA Thrust ve cuDF gibi kütüphanelerin varsayılan tam sayı sıralaması radix tabanlıdır; merge sort'un dallanması GPU'da pahalıdır, radix'in düz histogramı değildir.



Şekil 14.5: Counting sort (Ku §7, **İMZA** figür): doğrudan erişim dizisinin sıra-koruyan ikizi. u kova (burada $u = 6$, indeks $0..5$); her öğenin anahtarı k, onu k. kovaya yollar. Doğrudan erişim sortundan farkı: her kovada tek öğe değil bir **ZİNCİR** (chain) tutulur ve öğeler **geliş sırasıyla** sona eklenir (amber yukarı-oklar geliş yönünü gösterir). Girdi $(3a, 1a, 3b, 5a, 1b, 3c)$ — harfler eşit anahtarların girdi sırasını işaretler. Dağıtımdan sonra zincirler sayma histogramıdır: kova $1 = [1a, 1b]$ (#2), kova $3 = [3a, 3b, 3c]$ (#3), kova $5 = [5a]$ (#1), gerisi boş. Çıktıyı üretmek için zincirleri 0'dan $u-1$'e **sırayla oku**: $[1a, 1b, 3a, 3b, 3c, 5a]$ — eşit anahtarlı öğeler girdi sırasını korur, yani sıralama **kararlıdır** (stable). Toplam iş = zincirlere n öğe ekle + u kovayı gez $= O(n + u)$; $u = \Theta(n)$ ise **lineer**. Bu kararlılık, radix sort'un (Şekil 14.6) basamak-basamak doğru çalışmasının temelidir.



Şekil 14.6: Radix sort = **counting sort + base-n basamaklar** (Ku §8, İMZA örneği $17, 3, 24, 22, 12$; $n = 5$, taban $5 \to$ her sayı iki basamak). $u = \max + 1 = 25 \leq n^2$ olduğu için $\lceil \log_5 25 \rceil = 2$ basamak yeter. **1. geçiş (LSD, birler basamağı):** girdideki her sayının **en az** önemli basamağı (amber rozet $[d_0]$: 17=[2], 3=[3], 24=[4], 22=[2], 12=[2]) ile **kararlı** counting sort $\to [17, 22, 12, 3, 24]$. **2. geçiş (MSD, n'ler basamağı):** bu kez **en** önemli basamak ($[d_1]$: 17=[3], 22=[4], 12=[2], 3=[0], 24=[4]) ile yine kararlı counting sort $\to [3, 12, 17, 22, 24]$ (amber, **sıralı** ✓). Kararlılık şarttır: 2. geçişte n'ler basamağı eşit olan (22 ve 24, ikisi de 4) öğeler 1. geçişten gelen birler sırasını korur $\to$ tuple sort doğru çalışır. Her geçiş $O(n + \text{base})$; sabit sayıda geçiş ($u \leq n^c \to c$ basamak), base $= n$ alınırsa toplam $O(n)$ — **doğrusal zaman**, karşılaştırma sınırı $\Omega(n \log n)$ aşılır.

- **Veritabanı / büyük veri:** Spark, ClickHouse gibi sistemler tam sayı/sabit anahtar kolonlarında radix veya hibrit (radix + merge) sıralama kullanır; "anahtar küçük tam sayı mı?" sorusu sıralama planlayıcısının ilk kontrolüdür.

Tek cümle: *Anahtarların küçük tam sayı olduğunu bildiğin an, radix sort hem asimptotik ($O(n)$) hem donanım (GPU paralelliği) açısından merge sort'u geçer.*

14.10 Bu Dersin Özeti

1. **Karşılaştırma modelinde** sıralama $\Omega(n \log n)$'dir: $n!$ permütasyon → $n!$ yaprak → $\log(n!) = \Theta(n \log n)$.
2. **Direct access array sort**: benzersiz anahtar + küçük aralık → $O(n + u)$; $u = \Theta(n)$ ile lineer.
3. **Anahtarı base-n basamaklara ayır** ($a = k//n, b = k \bmod n$) → $u \leq n^2$ için ikililer.
4. **Tuple sort**: basamakları en az önemliden en önemliye, kararlı sıralamayla sırala.
5. **Kararlı (stable) sıralama**: eşit anahtarlar girdi sırasını korur — tuple sort için şart.
6. **Counting sort**: direct access + zincir (sıra korur); kararlı; $O(n + u)$.
7. **Radix sort**: $\log_n u$ basamak, counting sort ile; $u \leq n^c$ ise $O(n)$.

! Tek Bir Cümle

Karşılaştırma $n \log n$'de tıkanır; ama anahtarlar küçük tam sayıysa, onları indeks yapıp (counting sort) basamak basamak kararlı sıralayarak (radix sort) $O(n)$'e inilir.

Şekil 14.7 bu sıçramayı tek grafikte özetler: Ders 4'ün $n \log n$ merge sort'undan, bu dersin $O(n)$ radix sort'una.

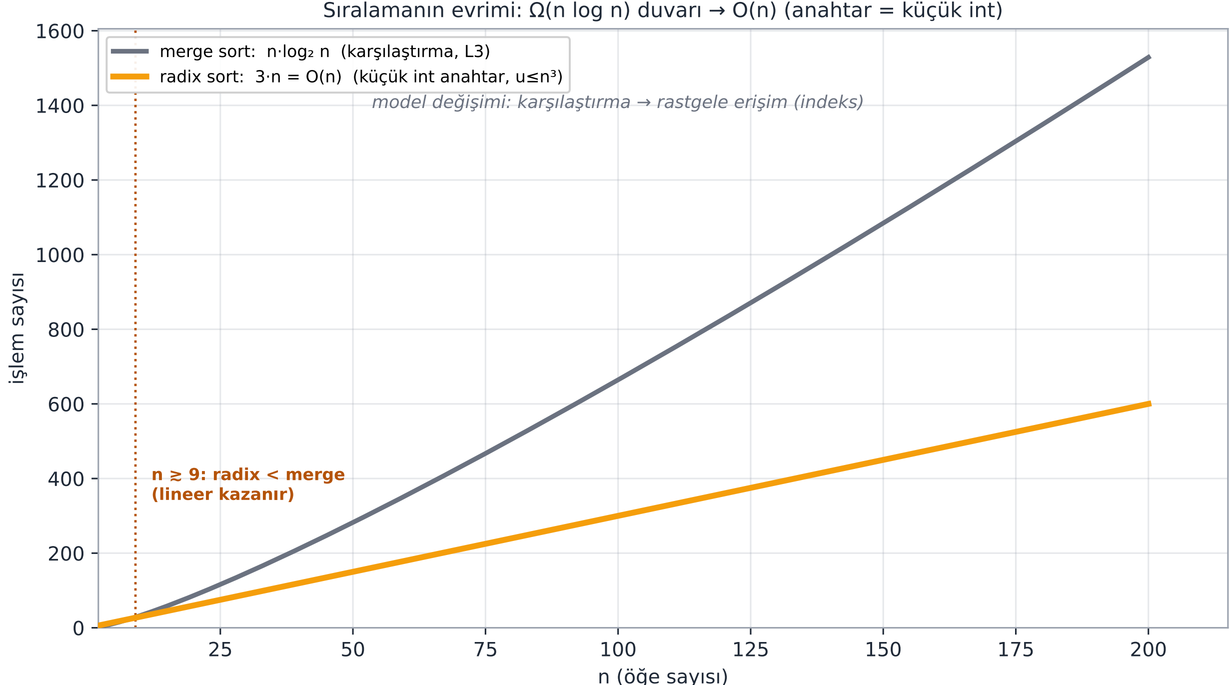
14.11 Kontrol Soruları

i Soru 1: Sıralama alt sınırının neden Ω(n log n) olduğunu, karar ağacı + n! ile özetle.

Cevap: Bir sıralama, girdinin $n!$ olası permütasyonundan herhangi birini üretebilmeli → karar ağacında en az $n!$ yaprak. İkili ağacın minimum yüksekliği $\log(n!)$, ve en kötü durum karşılaştırma sayısı = yükseklik. $\log(n!) \geq \log((n/2)^{n/2}) = (n/2)\log(n/2) = \Theta(n \log n)$. Dolayısıyla her karşılaştırmalı sıralama $\Omega(n \log n)$'dir; merge sort bu sınıra ulaştığı için optimaldir.

i Soru 2: Counting sort neden direct access array sort'tan daha geneldir?

Cevap: Direct access array sort, her indekste tek öğe sakladığı için **benzersiz anahtar** ister. Counting sort, her indekste bir **zincir** (sıra koruyan sequence yapısı) tutar; böylece tekrar eden anahtarları geldikleri sırada saklar. Bu hem tekrarlı anahtarları destekler hem de **kararlılık** sağlar — ki radix sort'un çalışması için bu şarttır.



Şekil 14.7: Sıralamanın evrimi — maliyet duvarının yıkılışı. **Slate eğri:** merge sort, $n \cdot \log_2 n$ (böl-yönet + two-finger merge, Ders 4/L3) — karşılaştırma modelindeki $\Omega(n \log n)$ **duvarı**, selection sort'un $\Theta(n^2)$'sinden çok daha iyi olsa da bu duvarı aşamaz. **Amber çizgi:** radix sort, $3 \cdot n = O(n)$ — anahtarlar **küçük tamsayı** olduğunda ($u \leq n^3$, yani $c = 3$ sabit basamak) sıralama **doğrusal**. İki eğri $n \approx 9$'da kesişir (noktalı amber dikey çizgi); ötesinde radix sort merge sort'un altında kalır, yani **doğrusal kazanır**. Sıçramanın sebebi modeli değiştirmektir: karşılaştırma yerine anahtarı doğrudan **indeks** olarak kullanmak (counting/radix sort), karşılaştırma alt sınırının dışına çıkmanın tek yoludur. Bu figür Ders 4'ün $n \log n$ merge sort'unu (bkz. Şekil 11.5) Ders 7'nin $O(n)$ radix sort'una (bkz. Şekil 14.6) bağlar.

> **i** Soru 3: Tuple sort'ta basamakları neden en az önemliden en önemliye sıralarız, tersi değil?
>
> **Cevap:** En önemli basamak, son geçişte sıralanmalı ki nihai sırada baskın olsun. Eğer önce en önemliyi sıralarsak, sonra en az önemliyi sıralamak önceki işi **bozar** (eşitlikler yeniden dağılır). En az önemliden başlayıp en önemliyle bitirince, kararlı sıralama sayesinde en önemli basamakta eşit olanlar önceki (az önemli) sıralamayı korur. Bu yüzden hem sıra (en az → en önemli) hem de **kararlılık** zorunludur.

> **i** Soru 4: Radix sort hangi koşulda O(n) olur? n^3'e kadar anahtarlar için kaç basamak gerekir?
>
> **Cevap:** Radix sort $O(n + n \cdot \log_n u)$'dur. $u \leq n^c$ (sabit c) ise $\log_n u = c$ sabittir → $O(n)$ lineer. n^3'e kadar anahtarlar için ($u = n^3$): her anahtarı base-n üç basamağa ayırırız ($\log_n n^3 = 3$, sabit). Üç kararlı counting sort geçişi, her biri $O(n)$ → toplam $O(3n) = O(n)$. Anahtarlar n'in bir polinomuyla sınırlı kaldıkça radix sort lineerdir.

14.12 Egzersizler

Egzersiz 1. Stirling yaklaşımını ($n! \approx \sqrt{2\pi n}(n/e)^n$) kullanarak $\log(n!) = \Theta(n \log n)$'i göster ve Bölüm 2'deki $(n/2)^{n/2}$ alt sınırıyla karşılaştır.

Egzersiz 2. $u = n^2$ için, radix sort'un iki basamaklı (base-n) çalışmasını $17, 3, 24, 22, 12$ üzerinde elle yürüt; her counting sort geçişinin çıktısını yaz.

Egzersiz 3. Bir sıralamanın kararlı olup olmadığını test eden bir örnek tasarla: aynı anahtarlı ama farklı "yük" taşıyan öğelerle. Selection sort kararlı mıdır?

Egzersiz 4. Python'da counting sort'u tek basamak için yaz ve `sorted(..., key=...)`'in kararlılığıyla karşılaştır:

```
def counting_sort(A, key, base):
    chains = [[] for _ in range(base)]
    for x in A:
        chains[key(x)].append(x)      # geldigi sirada ekle -> kararli
    out = []
    for chain in chains:
        out.extend(chain)
    return out
```

Egzersiz 5. Radix sort'un çalışma süresi $O(n \cdot \log_n u)$ için, $u = 2^{32}$ (32-bit tam sayılar) ve $n = 1000$ olduğunda kaç basamak gerekir? Bu $n \log n$'den iyi mi?

14.13 Sonraki Ders İçin Hazırlık

Ders 6 (L6): İkili Ağaçlar — Bölüm 1

Erik Demaine ile, sıralı diziyi ($O(n)$ ekleme) ve hash tablosunu (worst-case $O(n)$) aşan dengeli bir yapıya — **ikili ağaçlara** — geçiyoruz: tüm işlemler worst-case $O(\log n)$. (Not: ders akışında araya **Problem Oturumu 3** girer.)

> **Ders 6 Öncesi Yapılacak**
>
> - Bu dersin egzersizlerini, özellikle Egzersiz 4'ü (counting sort) çöz.
> - Üç sıralama yöntemi (direct access → counting → radix) arasındaki ilişkiyi ezberden anlat.
> - Ana cümleyi tekrar oku: *"Anahtarlar küçük tam sayıysa, basamak basamak kararlı sıralayarak $O(n)$'e inilir."*

14.14 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Sıralama alt sınırı	Karşılaştırma modelinde $\Omega(n \log n)$; $n!$ yaprak	Böl. 2
Direct access array sort	Benzersiz anahtar, küçük aralık; $O(n+u)$	Böl. 3
Base-n ayırma	$k = a \cdot n + b$; $a = k//n$, $b = k \bmod n$	Böl. 4
Tuple sort	Basamakları en az → en önemli, kararlı sırala	Böl. 5
Kararlı (stable) sıralama	Eşit anahtarlar girdi sırasını korur	Böl. 6
Counting sort	Direct access + zincir; kararlı; $O(n+u)$	Böl. 7
Radix sort	$\log_n u$ basamak + counting sort; $u \leq n^c \to O(n)$	Böl. 8

14.15 Builder ve OMSCS Bağlantıları

> 6 köprü
>
> Bu ders, "ne zaman karşılaştırmadan vazgeçilir" sezgisini kurar — köprülerin özeti:
>
> 1. $n \log n$ **alt sınırı** → teorik temel: hiçbir karşılaştırmalı sıralama bunu geçemez; modeli değiştirmek (rastgele erişim) tek çıkış.
> 2. **Counting/radix sort** → büyük veri: sabit-uzunluk tam sayı/string anahtarlarda (IP, timestamp, sabit ID) karşılaştırmalı sıralamayı geçer; GPU ve veritabanı sıralaması.
> 3. **Kararlı sıralama** → çok-kolonlu sıralama: SQL `ORDER BY a, b`, pandas `sort_values` — hep kararlı, en az önemliden uygulama mantığı.
> 4. **Base-n ayırma** → genel teknik: büyük problemi sabit sayıda küçük parçaya bölmek (radix, bucket, hash bölmeleri).
> 5. **Direct access fikri** → counting sort ve hashing ortak kökü: "anahtar = indeks" — arama ve

sıralamanın aynı sezgisi.
6. **Polinom-sınırlı anahtar** → pratik kural: anahtarlar n^c ile sınırlıysa lineer sırala; değilse merge sort'a dön.

❗ Tek bir şey alıp gideceksen

Karşılaştırma modelinde $n \log n$ bir duvardır — $n!$ permütasyon onu zorunlu kılar. Ama anahtarların küçük tam sayı olduğunu *bilirsen*, onları doğrudan indekse çevirip (counting sort) basamak basamak kararlı sıralayarak (radix sort) duvarı aşar, $O(n)$'e inersin.

15 Problem Oturumu 3

Ders 4-5'in uygulaması: hashing ve doğrusal sıralamanın dört uygulaması — indirgeme, değişmez, radix dönüşümleri, two-finger ve kayan pencere

ℹ Oturum bilgisi

- **Demaine'in videosu:** YouTube — Problem Session 3 (≈60 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 3
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 8 (Problem Oturumu 3)
- **Hoca:** Erik Demaine
- **Okuma süresi:** ≈26 dk

15.1 Bu Problem Oturumu Ne Hakkında?

Üçüncü problem oturumu (Erik Demaine), son iki haftanın konularını uygular: **hashing** (Ders 5) ve **doğrusal-zamanlı sıralama** (counting/radix sort, Ders 4-5). Demaine, her problem için izlediği yöntemi açıkça gösterir: kelime problemini biçimsel bir algoritma hedefine çevir, fikirler üret, sonra detayları kontrol et. Bu üç adımlı disiplin, dört problemin de ortak iskeletidir.

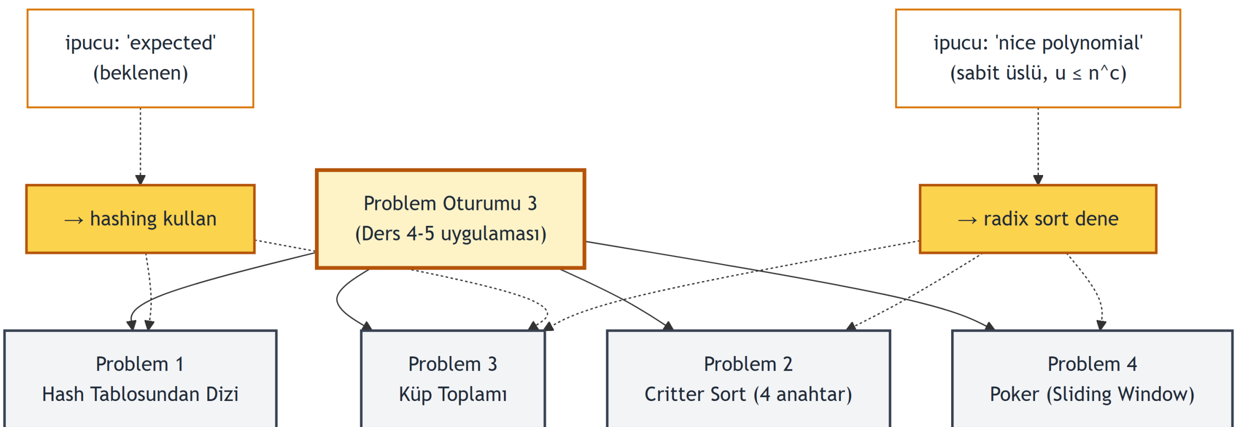
> *"convert the word problem into a concise formal algorithms thing… come up with ideas… check the details."* — Demaine, 0:46

İki anahtar ipucu tüm oturuma damga vurur (Şekil 39.1): bir problem ifadesinde "**expected**" (beklenen) görürsen rastgelelik söz konusudur ve elindeki tek rastgelelik aracı **hashing**'dir; "**nice polynomial**" (sabit üslü polinom, $u \leq n^c$) görürsen **radix sort** dene. Dört problem birer "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir.

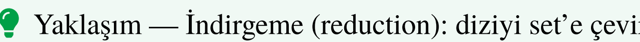
> *" 'expected' is always a good keyword, because it means randomization is involved… the only form of randomization you will use is essentially hashing."* — Demaine, 6:10

15.2 Problem 1: Hash Tablosundan Dizi

İfade. Bir hash tablosu (set: build $O(n)$ beklenen, find $O(1)$ beklenen, insert/delete $O(1)$ beklenen amortize) **black box** olarak verilir. Bununla bir **dizi (sequence)** kur: get/set_at $O(1)$ beklenen, insert/delete_at $O(n)$ beklenen, insert/delete first/last $O(1)$ beklenen amortize.



Şekil 15.1: Problem Oturumu 3'ün kavram haritası: kök (PS3) dört probleme dallanır ve iki anahtar ipucu düğümü tüm araç seçimini yönlendirir. Problem 1 hash tablosundan dizi kurar (indirgeme + değişmez); Problem 2 dört farklı anahtarı doğrusal sıralar (radix dönüşümleri); Problem 3 küp toplamını çözer (hash vs two-finger); Problem 4 poker ellerini sayar (kayan pencere + frekans tablosu). Soldaki iki ipucu — 'expected' bekleneni görünce hashing, 'nice polynomial' sabit üslü polinom görünce radix sort — Demaine'in problem ifadesinden araca giden tercüme kuralıdır.


Yaklaşım — İndirgeme (reduction): diziyi set'e çevir

Bu bir **indirgemedir (reduction)**: sequence problemini, çözümünü zaten bildiğimiz set problemine çeviriyoruz. İlk hamle, zorlukları ayırmaktır. `insert_at/delete_at` için $O(n)$ bütçesi var → her seferinde tüm yapıyı **yeniden kurmak** serbest. Asıl ustalık gerektiren kısım, uçlardaki ($O(1)$ amortize) işlemlerdir; orada bir indeks hilesi gerekir.

> *"this is what we call a reduction… we're reducing the sequence problem to the set problem."* — Demaine, 3:17

Çözüm. Anahtar fikir: her öğeye **anahtar = sıradaki indeksi** ver, böylece set'te saklanabilir.

- **get_at(i):** `find(first + i).value`. **set_at(i, x):** `find(first + i)` ile nesneyi bul, value'sunu x yap. $O(1)$ beklenen.
- **insert/delete_at(i):** iter ile öğeleri bir diziye çıkar, kaydır, `build` ile yeniden kur. $O(n)$ — bütçe içinde.
- **insert/delete_last:** anahtar = `first + length` (veya `first + length - 1`). $O(1)$.
- **insert/delete_first:** sorun — indeks 0'a eklemek tüm anahtarları kaydırır ($O(n)$). Çözüm: bir `first` değişkeni tut (en küçük anahtar); başa eklerken `first`'ü azalt (negatif anahtarlara izin ver). İndeks i artık `first + i` anahtarına eşlenir.

Aşağıdaki özet, motorun (`HashSequence`) çekirdek mantığını gösterir: `first` değişkeni ve `insert_first`'ün `first`'ü azaltışı, `insert_at`'in iter + build ile yeniden kurması.

```
class HashSequence:                          # diziyi black-box set'e indirger
    def __init__(self, iterable=()):
        self._d = {}                         # set: anahtar -> değer
        self._first = 0                      # en küçük anahtar (negatif olabilir)
        self.build(iterable)
```

```
    def get_at(self, i):
        return self._d[self._first + i]         # find(first+i) — O(1) beklenen

    def insert_first(self, x):
        self._first -= 1                        # başa ekle: first AZALIR (negatif anahtar)
        self._d[self._first] = x                # O(1) — hiçbir şey kaymaz

    def insert_last(self, x):
        self._d[self._first + len(self._d)] = x  # anahtar = first + length — O(1)

    def insert_at(self, i, x):
        items = self.to_list()                  # iter ile çıkar — O(n)
        items.insert(i, x)
        self.build(items)                       # yeniden kur — O(n), bütçe içinde
```

Bu, bir **değişmez (invariant)** doğurur: anahtarlar her zaman `first` ile `first + length - 1` arasındadır. Demaine, değişmez yazmanın veri yapısının *neden doğru* olduğunu gösterdiğini vurgular; doğruluk her işlemin değişmezi koruduğunu göstererek tümevarımla ispatlanır. Şekil 15.2 bu değişmezin `build → insert_first → insert_last → delete_first` boyunca nasıl korunduğunu adım adım izler.

> *"This is what we call an invariant. Useful to write these things down so you can understand… why is your data structure correct?"* — Demaine, 26:23

Karmaşıklık. get/set_at $O(1)$ beklenen, insert/delete_at $O(n)$ beklenen, uç işlemleri $O(1)$ beklenen amortize. (Negatif anahtarlar, çift-katlama "folding" hilesiyle non-negatife eşlenebilir.)

15.3 Problem 2: Critter Sort

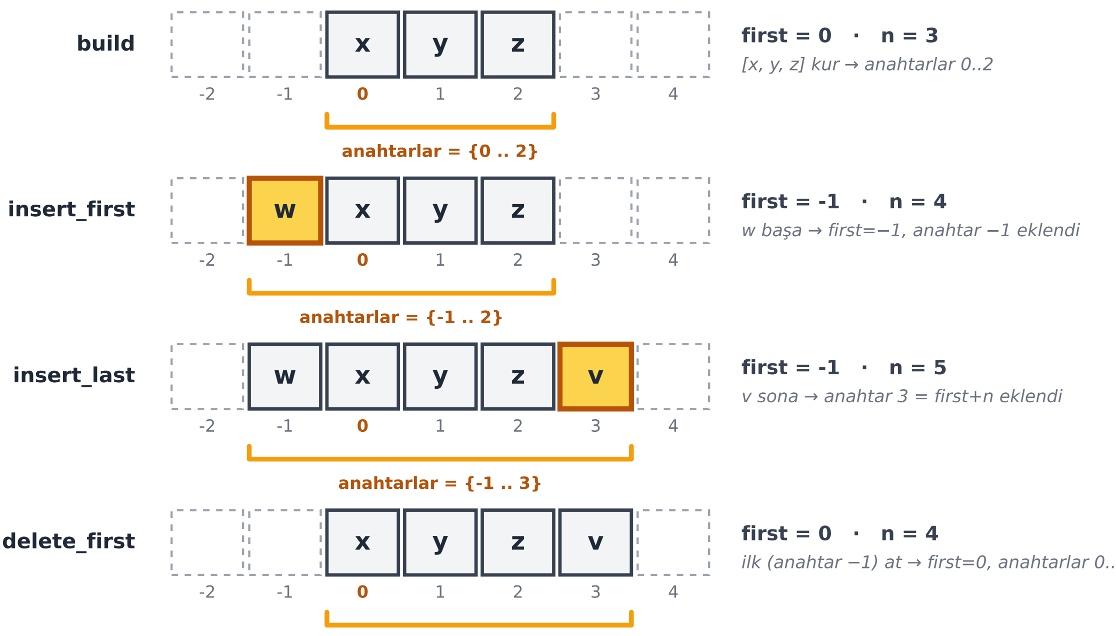
İfade. n nesneyi dört farklı anahtar türüne göre sırala; her biri için en hızlı doğru algoritmayı bul. (a) tam sayılar $-n..n$; (b) 26-harfli, uzunluğu $\leq 10 \cdot \log n$ olan string'ler; (c) tam sayılar $0..n^2$; (d) rasyoneller w/f, $0 < w < f < n^2$.

Yaklaşım — Her Anahtarı radix sort'un Kabul Ettiği Tamsayıya Dönüştür

Hepsinde radix sort denenir, ama her biri bir **dönüşüm** ister: radix sort yalnızca $0..u-1$ aralığında tamsayı anahtar alır ve doğrusal kalması için $u \leq n^c$ (sabit üslü) olmalıdır. İş, her anahtar türünü bu kalıba sokan doğru dönüşümü bulmaktır — negatifi kaydır, string'i tek sayıya çevir, rasyoneli çapraz çarp veya ölçekle.

Çözüm. Dört şık:

- **(a)** $-n..n$**:** Negatifleri "folding" ile araya serpmek sırayı bozar (Demaine'in özellikle uyardığı tuzak). Doğru hamle: **her anahtara n ekle** → $0..2n$. Radix sort, $u = 2n+1$ → **doğrusal**.



Şekil 15.2: Hash tablosundan Sequence İNDİRGEMESİ + değişmez (invariant) izi — Problem 1. Anahtar ekseni −2..4 (7 hücre); indeks i → anahtar (first + i) eşlemesi. **Satır 1 build** [x,y,z]: anahtarlar 0..2, first=0. **Satır 2 insert_first(w):** w başa eklenir, first −1'e iner ve −1 anahtarı doğar (amber) — hiçbir şey kaymadan O(1). **Satır 3 insert_last(v):** v sona eklenir, anahtar 3 = first+length doğar (amber). **Satır 4 delete_first:** en küçük anahtar (−1) atılır, first 0'a döner. Her satırın altındaki amber köşeli parantez **değişmezi** gösterir: anahtarlar her işlemden sonra TAM OLARAK first..first+n−1 aralığını doldurur — boşluk yok, taşma yok. Bu, veri yapısının doğruluğunu tümevarımla garanti eder (Demaine 26:23).

- **(b) String'ler:** Her harfi bir basamağa eşle, sonra tüm string'i **base-n tek sayıya** çevir (en önemli harf en yüksek basamak; kısa string'leri sonda doldur). String uzunluğu $10 \cdot \log n$ olsa da, base-n radix sort her counting-sort geçişinde $\log n$ harfi birden işler → yalnızca 10 geçiş ($u = n^{10}$) → **doğrusal**. (Harf-harf base-26 tuple sort olsaydı $\log n$ geçiş gerekir = $n \log n$, yavaş.)

  "Tuple sort is the thing… sort by the last thing, then sort by the previous thing… You can also think of this as radix sorting on a number written in base 26." — Demaine, 33:53

- **(c)** $0..n^2$**:** Doğrudan radix sort (iki counting-sort geçişi, $u = n^2$) → **doğrusal**.
- **(d) Rasyoneller** w/f**:** İki yol. (1) **Merge sort + çapraz çarpım:** w_i/f_i ile w_j/f_j'yi kıyaslamak için bölme yerine $w_i \cdot f_j$ ile $w_j \cdot f_i$'yi karşılaştır (paydalar pozitif olduğundan işaret korunur) → $O(n \log n)$. (2) **Doğrusal:** İki farklı oran en az $1/n^4$ uzaktadır ($|w_i f_j - w_j f_i| \geq 1$, payda $< n^4$); her oranı n^4 ile ölçekleyip tabana yuvarla ($w_i \cdot n^4 \;//\; f_i$) → farklı oranlar farklı tamsayılara gider; $w_i < f_i$ olduğundan anahtarlar $\lfloor w_i \cdot n^4 / f_i \rfloor < n^4$, yani $u \leq n^4$ (bölmeden önceki ara çarpım $w_i \cdot n^4$ bile $< n^6$ — her iki sınır da sabit-üslü), radix sort → **doğrusal**.

  "Merge sort is always a good answer. It's not the best answer." — Demaine, 40:43

> Builder Notu — pad değeri harf kodlarından KÜÇÜK olmalı
>
> Şık (b)'de Notion harfleri $0..25$'e eşlemeyi önerir, ama kısa string'leri sonda **pad** ile doldururken dikkat gerekir: pad = 0 ve `'a'` = 0 olsaydı `ab` ile `aba` aynı sayıya giderdi (`ab` + pad = `ab` + `a`), leksikografik sıra bozulurdu. Motor bu yüzden harfleri $1..26$ kodlar ve **pad = 0** kullanır (taban 27) — pad tüm harflerden küçük kaldığından `ab` < `aba` korunur. İfade düzeyinde bağlayıcı olan, leksikografik doğru sıralamadır; kodlama detayı ona hizmet eder.

Karmaşıklık. (a), (b), (c) doğrusal; (d) merge sort ile $O(n \log n)$ veya ölçekleme + radix sort ile **doğrusal**.

15.4 Problem 3: Küp Toplamı

İfade. n tam sayı (küp kenar uzunlukları) verilir; toplamı h olan iki sayı bul. (a) tam çözüm, **doğrusal beklenen** zaman. (b) tam çift yoksa h'ye **en yakın küçük(-eşit)** çifti, **doğrusal en-kötü-durum** zamanında bul.

> Yaklaşım — 'expected' → hash, 'en-kötü-durum' → radix + two-finger
>
> İki şık iki ipucuyla ayrışır. (a) "**beklenen**" geçer → randomizasyon → **hashing**. (b) "**en-kötü-durum**" geçer → hash yasak (worst-case garantisi yok); ama $h = 600 \cdot n^6$ sabit bir polinomdur → **radix sort** ipucu, ardından sıralı dizide **two-finger** ile çift arama.

Çözüm.

(a) Hash ile: Tüm sayıları bir hash tablosuna koy (build $O(n)$ beklenen). Sonra her s_i için `find(h - s_i)` çağır — n çağrı, her biri $O(1)$ beklenen.

> *"Subtract h from si and see whether that exists."* — Demaine, 1:00:38

(b) Two-finger ile: Önce h'den büyük tüm sayıları **at** (asla çözümde olamazlar, sayılar negatif değil). Kalanlar $\leq h$ olduğundan radix sort'la **sırala** (doğrusal). Sonra **iki parmak**: i = 0 (en küçük), j = son (en büyük). $s_i + s_j > h$ ise $j{-}{-}$ (toplam çok büyük); $\leq h$ ise aday listesine kaydet ve $i{+}{+}$ (daha büyük toplam dene). Her parmak diziyi yalnızca bir kez kateder.

> *"The two-finger algorithm… This is the big idea. You're doing it all the time in this class."* — Demaine, 1:10:34

Şekil 15.3 bu izi $[21, 5, 13, 8, 2, 34]$ girdisi ve $h = 30$ üzerinde gösterir: 34 > 30 olduğundan **elenir**, kalanlar sıralı $[2, 5, 8, 13, 21]$ olur. İki parmak **4 adımda** ilerler; en iyi aday $(29, 8, 21)$, yani $8 + 21 = 29 \leq 30$ — h'yi geçmeyen en büyük çift. Doğruluk bir **değişmezle** ispatlanır: her adımda, j'nin sağındaki ve i'nin solundaki tüm çiftler ya çok büyüktür ya da görülen en iyi adaydan küçük-eşittir.

Karmaşıklık. (a) doğrusal beklenen (hash); (b) doğrusal en-kötü-durum (radix sort + two-finger). İkili-arama yaklaşımı $O(n \log n)$ verirdi.

15.5 Problem 4: Poker — Kayan Pencere ve Frekans Tablosu

İfade. n kartlık bir deste (her kartta 26 harften biri). Bir "el" = i. konumdan başlayıp **döngüsel** k kart al, sonra **sırala**. (a) İki indeks i, j için ellerin eşit olup olmadığını **sabit zamanda** söyleyen bir yapı kur. (b) En sık görülen eli bul.

Yaklaşım — Sıralı el = frekans tablosu; pencereyi O(1)'de kaydır

El sıralandığından, hangi harfin nerede olduğu değil, her harften **kaç tane** olduğu önemlidir. 26 harf olduğundan her el bir **frekans tablosuyla** (26 sayı) temsil edilir. Bu tablo, base-$(n + 1)$ yazılmış 26 basamaklı bir sayıdır; $u = (n + 1)^{26}$ sabit üslü bir polinomdur → radix sort uygulanabilir. Ardışık eller bir kart farkıyla aktığından, tablo her seferinde sıfırdan değil **kayan pencere** ile $O(1)$'de güncellenir.

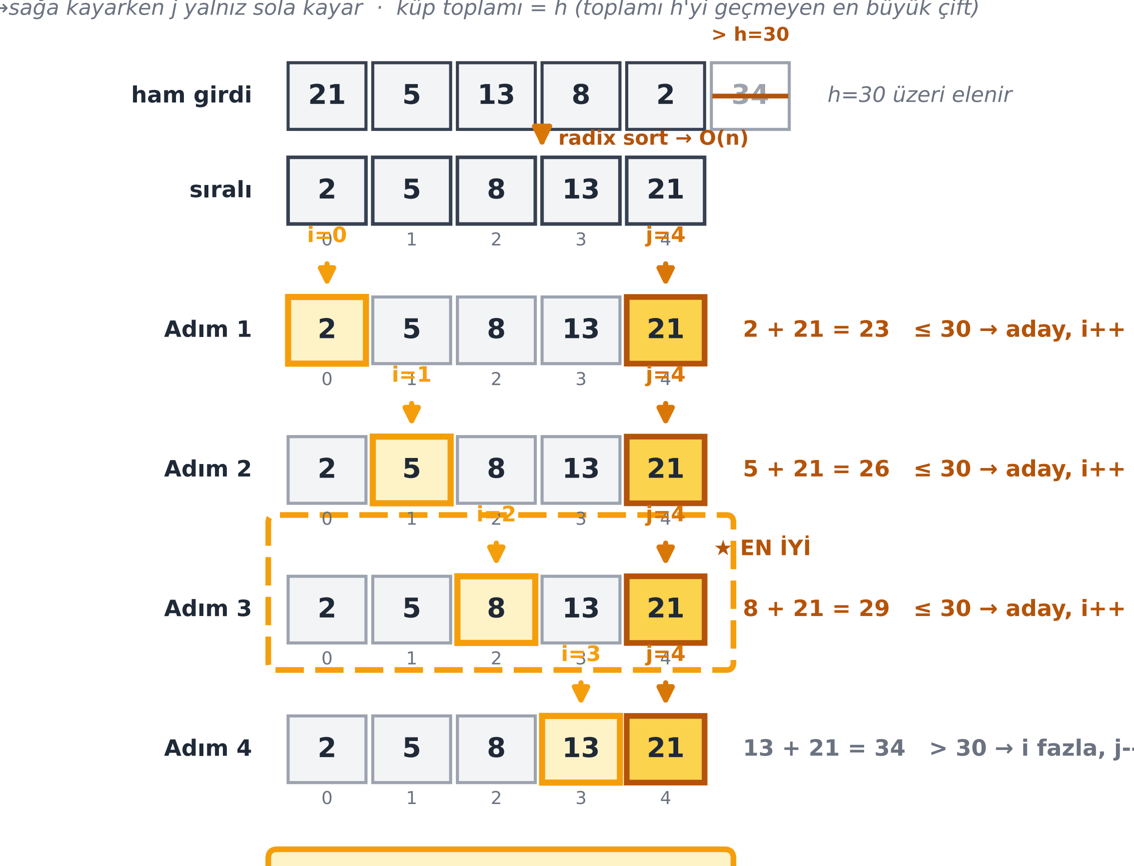
Çözüm.

(a) Kayan pencere (sliding window): İlk k kartın frekans tablosunu hesapla (26 sayım). Pencereyi bir kaydır: çıkan kartın sayacını azalt, giren kartın sayacını artır → $O(1)$'de bir sonraki elin tablosu. Her i için tabloyu kopyala (26 sayı, sabit). İki eli karşılaştırmak = 26 sayıyı karşılaştırmak → $O(1)$.

> *"It's called a sliding window technique, where you compute it for the first k guys. And then you remove this item and add this item."* — Demaine, 1:24:21

Aşağıdaki özet, motorun (`hand_tables`) kayan pencere çekirdeğini gösterir: ilk el $O(k)$, her sonraki el çıkan/giren tek kartla $O(1)$.

```
def hand_tables(deck, k):                         # her i için döngüsel k-kartlık el
    n = len(deck)
    freq = [0] * 26                               # 26 harf → frekans tablosu
    for t in range(k):
        freq[deck[t]] += 1                        # ilk el — O(k)
    tables = [tuple(freq)]
    for i in range(1, n):
```



Şekil 15.3: İki parmak (two-finger) küp toplamı izi — Problem 3b İMZA figürü. Ham girdi $[21, 5, 13, 8, 2, 34]$, hedef $h = 30$. **34 > h** olduğundan elenir (üzeri çizili); kalanlar radix sort ile sıralı $[2, 5, 8, 13, 21]$ olur. Sol parmak `i` (amber) en küçükten, sağ parmak `j` (koyu amber) en büyükten başlar. **Adım 1** ($2 + 21 = 23 \leq 30$): aday, i++. **Adım 2** ($5 + 21 = 26 \leq 30$): aday, i++. **Adım 3** ($8 + 21 = 29 \leq 30$): EN İYİ aday, i++. **Adım 4** ($13 + 21 = 34 > 30$): toplam fazla, j–. Sonuç **best = (29, 8, 21)** — h'yi geçmeyen en büyük çift. Her parmak diziyi yalnız bir kez katettiği için toplam O(n) (ikili-arama O(n log n) yerine).

```
        freq[deck[i - 1]] -= 1                      # çıkan kart: −1  (kayan pencere)
        freq[deck[(i + k - 1) % n]] += 1            # giren kart: +1  (döngüsel)
        tables.append(tuple(freq))                  # bir sonraki el — O(1)
    return tables
```

(b) En sık el: Tüm frekans tablolarını (her biri $(n+1)^{26}$ aralığında bir sayı) **radix sort** ile sırala (doğrusal, sabit üslü polinom). Tek bir taramayla eşit ardışık grupları say → en sık (ve istenirse leksikografik en iyi) eli bul.

Şekil 15.4 bu izi deste $a\,b\,a\,c\,b\,a$ ($n=6$, $k=3$, döngüsel) üzerinde gösterir: el0 → el1 geçişinde çıkan `a` (−1), giren `c` (+1). Eller arasında el0, el4 ve el5 aynı tabloya (a:2, b:1) sahiptir; bu üç tekrar onu **en sık el** (3 kez) yapar.

Karmaşıklık. (a) yapı kurma $O(n)$ (kayan pencere), karşılaştırma $O(1)$; (b) radix sort + tarama → $O(n)$.

15.6 Ne Öğrendik?

! Altı Taşınabilir Araç

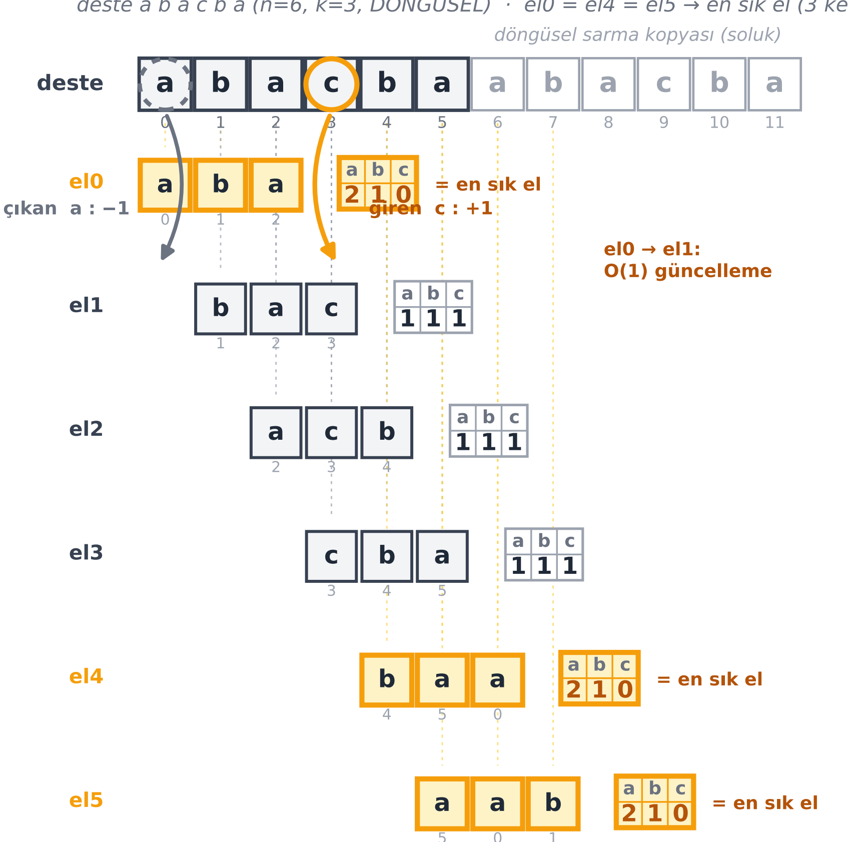
Bu oturum, Ders 4-5'in teorisini dört somut problemde uyguladı ve altı taşınabilir araç kazandırdı:

1. **Reduction (indirgeme):** bir problemi (sequence), çözümünü zaten bildiğimiz başka bir probleme (set/hash) çevirmek.
2. **Invariant (değişmez):** veri yapısının doğruluğunu tümevarımla garanti eden, her işlemde korunan koşul (`first..first+length-1`).
3. **"Expected → hashing, polynomial → radix sort":** problem ifadesindeki kelime ipuçlarını araca çevirmek.
4. **Radix sort dönüşümleri:** negatife n ekle, string'i base-n sayıya çevir, rasyoneli çapraz çarp veya ölçekle-yuvarla.
5. **Two-finger:** sıralı dizide, monoton ilerleyen iki parmakla $O(n)$ çift arama (ikili-aramanın $n \log n$'ini geçer).
6. **Sliding window + frequency table:** sabit alfabe (26) → her pencereyi $O(1)$'de güncelle; sıralı eli 26 sayıyla temsil et.

15.7 Sonraki

Ders 6 (L6) — İkili Ağaçlar, Bölüm 1

Sırada **Ders 6 (L6): İkili Ağaçlar — Bölüm 1** var — Erik Demaine ile, sıralı diziyi (O(n) ekleme) ve hash tablosunu (worst-case O(n)) aşan dengeli bir yapıya — **ikili ağaçlara** — geçiyoruz: tüm işlemler worst-case O(log n). Bu oturumdaki **değişmez (invariant)** ve **two-finger** sezgileri, ağaç işlemlerinin doğruluğunu kurarken doğrudan işine yarayacak.



Şekil 15.4: Kayan pencere (sliding window) + frekans tablosu izi — Problem 4. Deste $a\,b\,a\,c\,b\,a$ ($n = 6$, $k = 3$, DÖNGÜSEL); soluk şerit sağda döngüsel sarma kopyasıdır. Her el için yalnız a/b/c sütunları gösterilen mini frekans tablosu (26 sayıdan ilgili üçü) sağda yer alır. el0 → el1 geçişinde **çıkan** kart `a` (−1, soluk daire) ve **giren** kart `c` (+1, amber daire) tabloyu O(1)'de günceller — sıfırdan saymaya gerek yok. el0, el4 ve el5 aynı tabloya (a:2, b:1) sahiptir (amber çerçeveli) → **en sık el (3 kez)**. İki eli karşılaştırmak = 26 sayıyı karşılaştırmak = O(1) (sabit alfabe).

Builder Notu — reduction = adapter pattern; sliding window = stream processing

İki araç, yazılım mühendisliğinde doğrudan karşılık bulur. **Reduction**, bir API'yi başka bir API üzerine kurmaktır: tıpkı bir **adapter pattern**'in mevcut bir arayüzü (set) beklenen bir arayüze (sequence) sarması gibi — yeni mantık değil, mevcut yeteneğin yeniden paketlenmesi. **Sliding window** ise **stream processing**'in kalbidir: sonsuz/uzun bir akışta sabit boyutlu pencerenin özetini her adımda sıfırdan değil artımlı (çıkan −, giren +) güncellemek. CNN konvolüsyon kernel'i de sabit bir pencereyi görüntü üzerinde gezdirir — ama paylaşılan yalnızca pencere *geometrisidir*: konvolüsyon her konumda dot-product'ı **sıfırdan** hesaplar; bu dersin yük taşıyan özelliği olan **O(1) artımlı güncelleme** ise stream/windowed-aggregation tarafına özgüdür.

16 İkili Ağaçlar — Bölüm 1

Düğüm + parent/left/right işaretçileri, kök/yaprak, derinlik vs yükseklik, örtük traversal sırası, O(h) işlemler (subtree_first/successor/insert_after/delete) ve BST özelliği (sequence vs set)

Bölüm bilgisi

- **Demaine'in videosu:** YouTube — Lecture 6: Binary Trees, Part 1 (≈51 dk)
- **OCW sayfası:** MIT 6.006 Lecture 6: Binary Trees, Part 1
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 9 (L6)
- **Hoca:** Erik Demaine
- **Okuma süresi:** ≈25 dk

16.1 Bu Derste Ne Var?

Bugüne dek gördüğümüz yapıların hiçbiri *her şeyi* iyi yapmıyor: bağlı liste ortaya ekler ama ortaya $O(n)$'de ulaşır; dizi ortaya ulaşır ama eklerken kaydırır; hash tablosu find yapar ama find_prev/find_next'te kötüdür. Erik Demaine'in deyişiyle, **ikili ağaçlar** neredeyse her işlemi $O(\log n)$'de yapan "gördüğümüz en güçlü veri yapısıdır".

Bu ders (Bölüm 1) tüm işlemleri $O(h)$ (h = ağacın yüksekliği) yapar; bir sonraki ders $h = O(\log n)$ garantisini ekler.

Üç temel kavram bu derste yan yana gelir:

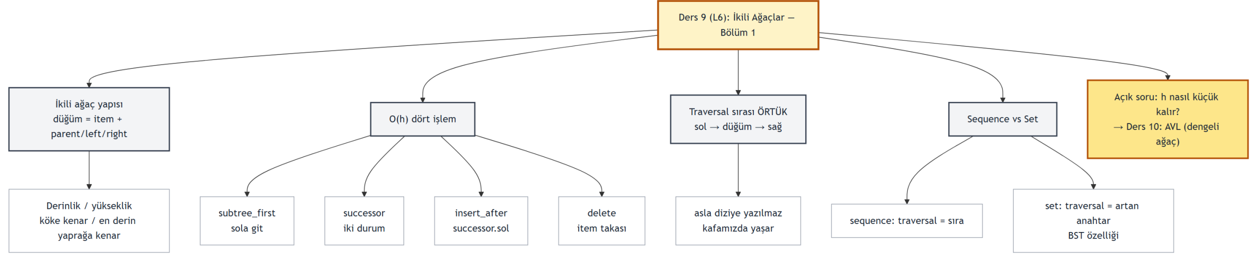
1. **İkili ağaç yapısı** — düğüm (item + parent/left/right), kök, yaprak, traversal (geziş) sırası.
2. $O(h)$ **işlemler** — subtree_first, successor, insert_after, delete; hepsi yükseklikle sınırlı.
3. **BST özelliği** — sıralı anahtarlarla, ağaç ikili aramayı dinamik olarak destekler (find, find_prev/find_next).

> *"today we are doing some of the coolest data structures we will see in this class, maybe some of the coolest data structures ever — binary trees."* — Demaine, 0:18

Builder Notu — İki Dünyanın Köprüsü

İkili ağaç, şimdiye dek gördüğümüz iki dünyanın güçlü yanlarını *dinamik* olarak birleştirir: bağlı listenin esnek değiştirmesi + sıralı dizinin hızlı araması.

- **Geriye → Ders 2-5:** ağaç, bağlı listenin (esnek ama yavaş erişim) ve sıralı dizinin (hızlı arama ama statik) güçlü yanlarını birleştirir — birinin zaafı ötekinin gücüdür.



Şekil 16.1: Ders 9'un (L6) kavram haritası: ikili ağaç yapısından (düğüm = item + parent/left/right) O(h) dört temel işleme (subtree_first → successor → insert_after → delete), in-order traversal sırasının ÖRTÜK temsiline (asla diziye yazılmaz) ve aynı ağacın iki arayüzüne — sequence (traversal = sıra) ve set (traversal = artan anahtar, BST özelliği). Geriye kalan tek soru, yüksekliği h'yi nasıl küçük (log n) tutarız — bu açık düğüm Ders 10'da AVL ile kapanır.

- **Geriye → Ders 4 (ikili arama):** BST'de find, sıralı dizideki ikili aramanın ağaç üzerindeki halidir — ama ekleme/silme artık kaydırma değil, $O(h)$ işaretçi işi.
- **İleriye → veritabanı:** B-tree / B+-tree indeksleri tam bu fikrin disk-dostu genellemesidir; çoğu SQL motorunun varsayılan indeksi bir dengeli ağaçtır (B-tree).
- **İleriye → Ders 10 (denge):** bugün h herhangi bir şey olabilir (kötü ağaçta $O(n)$); sonraki ders AVL ile $h = O(\log n)$ garanti eder.

Tek cümle: *İkili ağaç, sıralı bir düzeni "kafadaki" traversal sırasıyla temsil edip, tüm işlemleri ağaç yüksekliği h ile sınırlar — yeter ki h küçük* ($\log n$) *kalsın.*

16.2 Hedef: Tüm İşlemler O(log n)

İki arayüzü hatırla: **dizi (sequence)** — ortaya ekle/sil, i. öğeye eriş; **küme (set)** — anahtarla ara, find_prev/find_next. Şimdiye dek hiçbir yapı bunların *hepsini* verimli yapmadı. İkili ağaçların hedefi: build/iterate dışındaki tüm işlemleri $O(\log n)$ (uçlarda sabit-zamanlı bağlı liste/dinamik diziden bir log faktör geride, ama her yerde hızlı).

Bugün bu işlemleri $O(h)$ (h = yükseklik) yapacağız; h'yi $\log n$'e bağlamak sonraki dersin işi. Şekil 16.2 bu "$O(h)$ iyi mi kötü mü?" sorusunu somutlaştırır: aynı 7 öğe, zincir ağaçta $h = 6$, dengeli ağaçta $h = 2$.

16.3 İkili Ağaç Nedir?

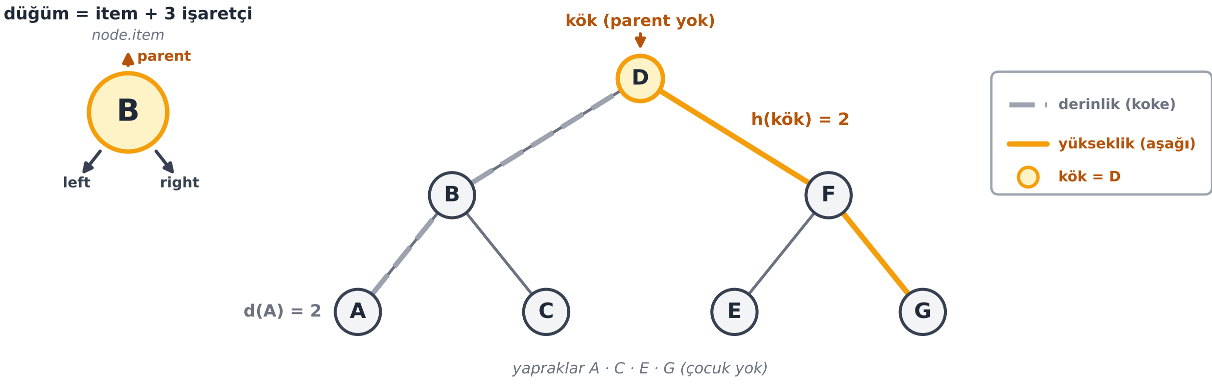
Bir **ikili ağaç**, dairelerle çizdiğimiz **düğümlerden (node)** oluşur. Her düğüm bir **item** ve üç işaretçi tutar: `node.parent`, `node.left`, `node.right`. Çizimde çift yönlü (parent ↔ child) bağlar olduğundan oklar yerine düz çizgiler kullanılır.

- **Kök (root):** parent'ı olmayan tek düğüm.
- **Yaprak (leaf):** çocuğu olmayan düğüm.
- **Değişmez:** `node.left.parent == node` (ve right için de) — parent, left/right işleminin tersidir.

Şekil 16.2: $O(h)$ iyi mi kötü mü? → tamamen h'ye bağlı. İki panel AYNI 7 öğeyle $(0..6)$ çizilir. **Sol — zincir ağaç** (yalnız sağ çocuklar, `build_chain_tree(7)`): bağlı liste gibi düz iner, $h = 6 = n - 1$ (motor hesaplı, EXACT); tüm $O(h)$ işlemler $O(n)$'e düşer. **Sağ — dengeli ağaç** (`build_balanced_tree([0..6])`, sıralı listeyi ortadan böl): yalnızca 3 seviye, $h = 2 = \lfloor \log_2 7 \rfloor$ (motor hesaplı, EXACT). İki ağacın in-order traversal'ı AYNI $[0, 1, 2, 3, 4, 5, 6]$ — yalnız ŞEKİL farklı. Açık soru: h'yi kim küçük tutar? → Ders 10: AVL (kendini dengeleyen ağaç) bu düğümü kapatır.

> *"with binary trees, because we use two types of next pointers — left and right — we can build a tree, and trees in general can have logarithmic height."* — Demaine, 8:37

Şekil 16.3 bu anatomiyi tek bir örnek ağaçta toplar: düğüm yapısı, kök/yaprak ve derinlik ile yüksekliğin görsel ayrımı.



Şekil 16.3: İkili ağaç anatomisi (L6 §2/§4): deterministik örnek ağaç (D kök; in-order traversal A, B, C, D, E, F, G; $h = 2$). **Orta:** 7 daire düğüm + düz çizgi kenarlar (ok yok — çift yönlü parent↔child bağı); kök D amber çerçeveli (parent yok), yapraklar A, C, E, G (çocuk yok). **Sol yakın çekim:** B düğümü = item + 3 işaretçi (parent yukarı, left/right aşağı). **Sağ:** iki yol kıyaslanır — A için **derinlik** (slate kesikli, $A \to B \to D$, köke 2 kenar, $d(A) = 2$) ve kök D için **yükseklik** (amber, $D \to F \to G$, en derin yaprağa 2 kenar, $h = 2$). Ağacın yüksekliği = kök yüksekliği = h; bugünkü tüm işlemler $O(h)$.

16.4 Neden İki İşaretçi?

Bağlı liste düğümünün tek "sonraki" işaretçisi vardır → yalnızca bir liste kurabilir, ortadaki düğümün **derinliği lineerdir** (oraya ulaşmak $O(n)$). İkili ağaç **iki** tür sonraki işaretçi (left, right) kullandığından bir *ağaç* kurabilir; ağaçlar **logaritmik yükseklikte** olabilir, böylece kökten herhangi bir düğüme yalnızca $\log n$ adımda ulaşılır. İki dünyanın en iyisinin sırrı budur.

16.5 Tanımlar: Alt Ağaç, Derinlik, Yükseklik

- **Alt ağaç (subtree of x):** x ve tüm torunları (descendants); x o alt ağacın köküdür.
- **Derinlik (depth):** x'ten köke giden yoldaki **kenar sayısı** (x'in atalarının sayısı). Kökün derinliği 0.

- **Yükseklik (height):** x'in alt ağacındaki **en uzun aşağı yoldaki kenar sayısı** (= alt ağaçtaki maksimum derinlik). Yaprakların yüksekliği 0.

> *"height of a node is the number of edges in the longest downward path."* — Demaine, 13:03

Ağacın yüksekliği = **kökün yüksekliği =** h. Bugünkü tüm işlemler $O(h)$ olacak; kötü bir ağaç (sadece sağ işaretçiler kullanılan, bağlı liste gibi) $O(n)$ yüksekliğe sahip olabilir — bundan kaçınmak sonraki dersin konusu (bkz. Şekil 16.2).

16.6 Traversal (Geziş) Sırası

Ağaçta doğal bir düzen vardır: **traversal sırası** (in-order). Özyinelemeli tanım: her düğüm için, `node.left` alt ağacındaki düğümler x'ten **önce**, `node.right` alt ağacındakiler **sonra** gelir.

> *"the nodes in x.left are before x, and the nodes in x.right come after x."* — Demaine, 18:24

Geziş algoritması: sol alt ağacı gez → x'i çıkar → sağ alt ağacı gez (tüm ağaç için $O(n)$). **Kritik:** bu sıra **asla açıkça hesaplanmaz** — diziye yazılamaz (ortaya ekleme $O(n)$ olurdu). Sıra her zaman *örtüktür*, "kafamızdadır"; ağaç yapısı onu örtük olarak kodlar.

> *"the traversal order is never explicitly computed... it's always implicit, in our heads."* — Demaine, 30:37

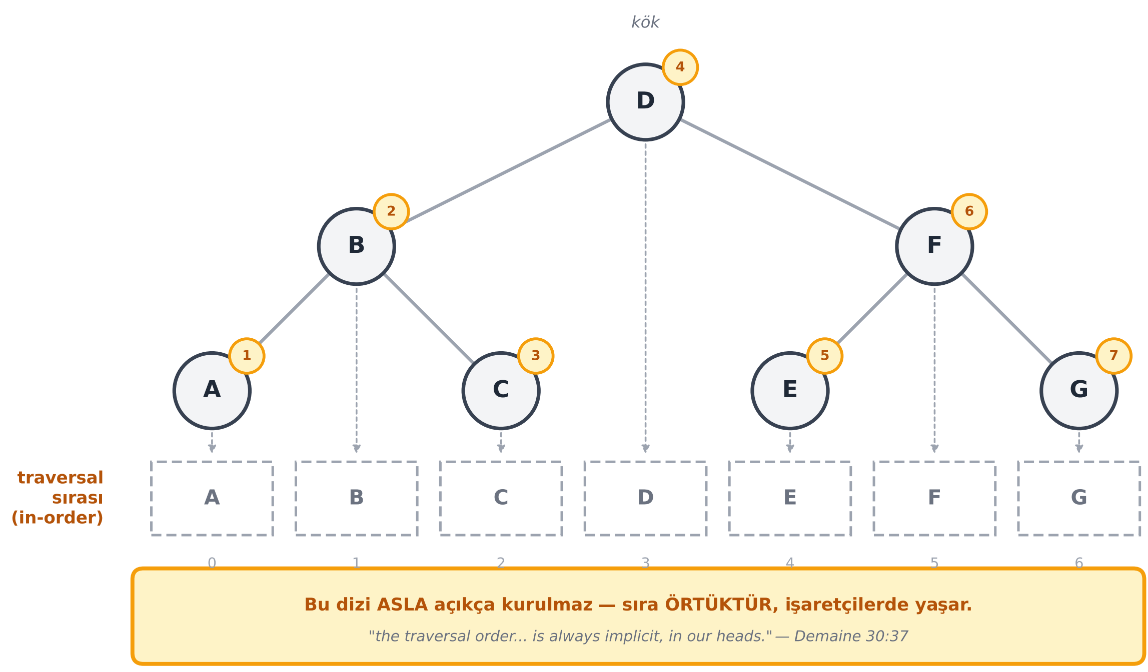
Şekil 16.4, örnek ağacın bu örtük sırasını (altta hayalet bir dizi olarak) ve düğümlerden o diziye inen kesikli okları gösterir.

Builder Notu — Örtük Düzen, Range Scan ve Lazy Temsil

Traversal sırasının asla diziye yazılmaması, bir veri yapısı tasarım ilkesinin saf hâlidir: *pahalı bir gösterimi açıkça tutmak yerine, ucuz güncellenebilir bir yapıyla örtük temsil et.*

- **DB indeks `BETWEEN` / range scan:** Bir B-tree indeksinde `WHERE x BETWEEN a AND b` sorgusu, tam olarak in-order traversal'ın bir parçasını yürür — `a`'yı bul, successor zinciriyle `b`'ye kadar git. Sıralı sonuç "kafadaki" düzenden okunur, ayrıca tutulan bir sıralı dizi yoktur.
- **Sıralı yineleme (ordered iteration):** Bir set'i artan sırada gezmek (`SELECT ... ORDER BY` indeks üzerinden) ağaçta doğal $O(n)$ traversal'dır; ekstra sıralama maliyeti yoktur çünkü düzen yapıda zaten kodludur.
- **Lazy temsil genel ilkesi:** "Materialize etme, gerektiğinde üret" — sıralı dizi (eager) yerine ağaç (lazy) seçmek, ortaya ekleme/silmeyi $O(n)$'den $O(h)$'ye düşürür. Aynı sezgi: streaming iterator'lar, generator'lar, sanal DOM diff'leri.

Tek cümle: *Örtük traversal, "sıralı görünmek" ile "sıralı dizi tutmak" arasındaki farktır — birincisi $O(h)$'de güncellenir, ikincisi $O(n)$'de.*



Şekil 16.4: In-order traversal: ağacın ÖRTÜK sırası (L6 §5; sol → düğüm → sağ). Üstte örnek ağaç (7 düğüm, kök D); her düğümün üst-sağında amber sıra rozeti ($A{\to}1, \ldots, G{\to}7$). Altta **hayalet** bir dizi şeridi $[A][B][C][D][E][F][G]$ — kesikli çerçeveli, çünkü bu dizi ASLA gerçekten kurulmaz; her düğümden kendi hücresine ince kesikli slate ok iner. Vurgu (amber kutu): sıra ÖRTÜKTÜR, işaretçilerde yaşar — diziye yazmak ortaya eklemeyi $O(n)$ yapardı, ağacın tüm amacı tam da bundan kaçınmaktır (Demaine 30:37).

16.7 subtree_first ve successor

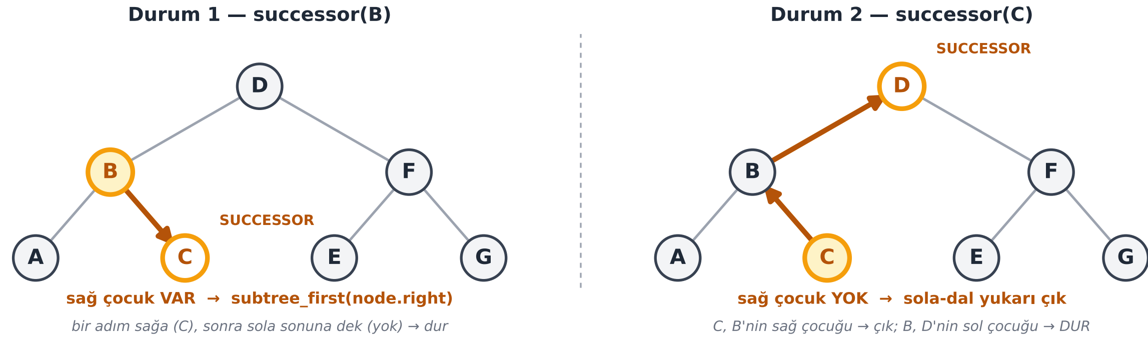
İki temel sorgu (ikisi de $O(h)$):

- **`subtree_first(node)`:** Bir alt ağacın traversal sırasındaki ilk düğümü. Tanım gereği sol her zaman önce gelir → mümkün olduğunca **sola git** (`node = node.left`), düşmeden bir önceki düğümde dur.
- **`successor(node)`:** Tüm ağacın traversal sırasında node'dan *sonraki* düğüm. İki durum:

> *"all of these operations are going to be order h."* — Demaine, 32:02

Çalışılan Örnek — successor. *Durum 1:* node'un sağ çocuğu varsa, successor = `subtree_first(node.right)` (sağ alt ağacın en solundaki düğüm — node'dan sonraki her şeyin ilki). *Durum 2:* sağ çocuk yoksa, **sola-dal yukarı çık**: `node = node.parent`, ta ki bir "sol dalı" yukarı çıkana dek (yani önceki düğüm parent'ın sol çocuğuysa). O parent, successor'dur. Sezgi: sağ alt ağaç yoksa, node'dan sonra gelen ilk şey, onu sol-tarafında bırakan en yakın atadır.

Örnek ağaçta (kök D): `successor(B) = C` Durum 1'dir (B'nin sağ çocuğu C var); `successor(C) = D` Durum 2'dir (C'nin sağ çocuğu yok, yol $C \to B \to D$). Şekil 16.5 bu iki durumu yan yana gösterir.



Şekil 16.5: Traversal successor — iki durum, her ikisi de $O(h)$ (L6 §6, **İMZA** figür; örnek ağaç $A..G$, $h = 2$). **Sol panel — Durum 1, successor(B):** B'nin sağ çocuğu VAR → successor sağ alt ağacın ilkidir: `subtree_first(node.right)`; bir adım sağa (C), sonra sola sonuna dek (yok) → dur. Yol $B \to C$ (amber), sonuç C. **Sağ panel — Durum 2, successor(C):** C'nin sağ çocuğu YOK → sola-dal yukarı çık: C, B'nin sağ çocuğu (devam et), B, D'nin SOL çocuğu (DUR). Yol $C \to B \to D$ (amber, yukarı), sonuç D. Sezgi: sağ alt ağaç yoksa, successor node'u sol tarafında bırakan en yakın atadır.

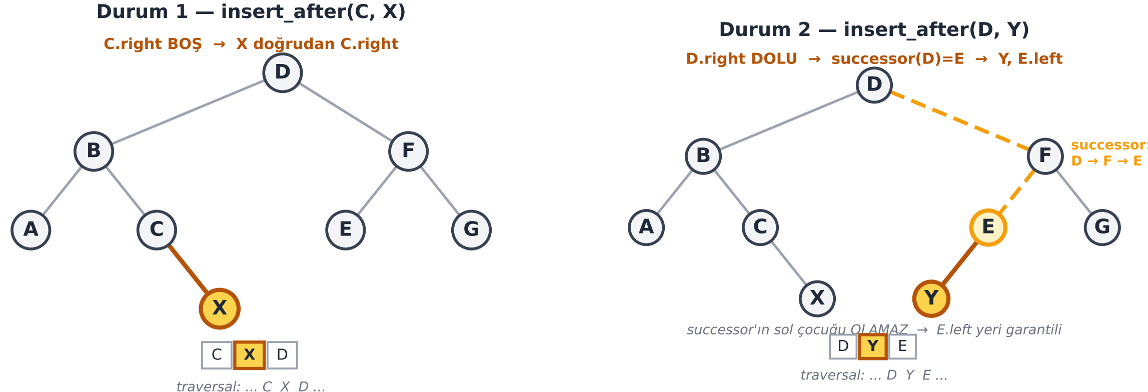
16.8 insert_after

`insert_after(node, new)`: yeni bir düğümü, traversal sırasında node'dan hemen sonraya ekle. İki durum (her ikisi de successor'a indirgenir, $O(h)$):

Çalışılan Örnek — insert_after. *Durum 1:* node'un **sağ çocuğu yoksa**, new'i doğrudan `node.right` yap. *Durum 2:* sağ çocuk varsa, node'un successor'ını bul (`subtree_first(node.right)`); bu successor'ın **sol çocuğu olmadığı garantilidir** (çünkü sağa bir kez, sonra sola sonuna dek giderek bulundu). O hâlde new'i successor'ın **sol çocuğu** yap. Yeni traversal sırası: node → new → eski successor → … Sabit-zaman iş + $O(h)$ successor = $O(h)$.

Örnek ağaçta `insert_after(C, X)` Durum 1'dir (C.right boş → X, C.right olur, traversal `...C, X, D...`); `insert_after(D, Y)` Durum 2'dir (D.right dolu (F), successor $= E$, $Y \to E$.left, traversal `...D, Y, E...`). Şekil 16.6 iki durumu yan yana gösterir.

insert_after: sabit işaretçi işi + O(h) successor arama = O(h)



Şekil 16.6: insert_after — sabit işaretçi işi + $O(h)$ successor arama $= O(h)$ (L6 §7; örnek ağaç $A..G$). **Sol panel — Durum 1, insert_after(C, X):** C.right BOŞ → X (amber yeni düğüm) doğrudan C.right olur; yeni traversal … C, X, D … **Sağ panel — Durum 2, insert_after(D, Y):** D.right DOLU (F) → successor$(D) = $ `subtree_first(D.right)` $= E$ (yol $D \to F \to E$, kesikli amber). Bu successor'ın sol çocuğu OLAMAZ (sağa bir kez + sola sonuna dek inilerek bulundu) → Y, E'nin SOL çocuğu olur; yeni traversal … D, Y, E … İki durumda da yalnız birkaç işaretçi değişir, maliyet successor aramasıyla sınırlı: $O(h)$.

16.9 delete

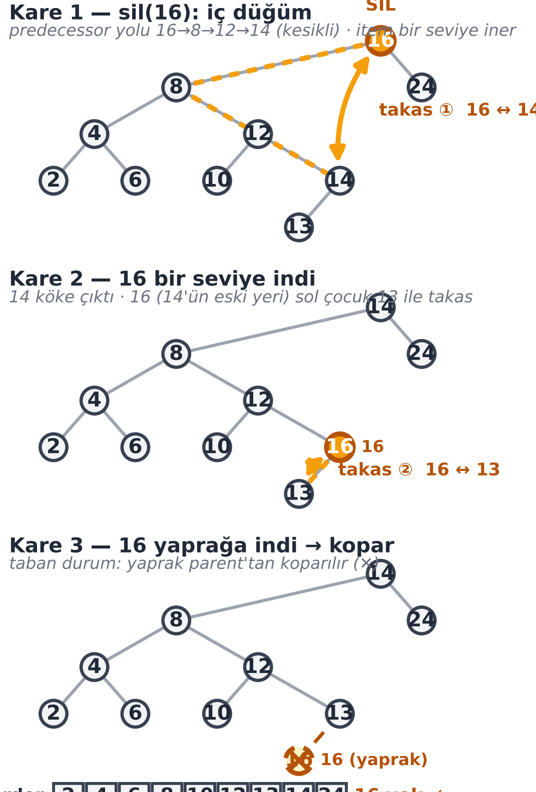
`delete(node)`: bir düğümü traversal sırasından çıkar. Düğüm bir **yaprak** ise basitçe parent'tan kopar (taban durum). Değilse, düğümü ağaçta **aşağı taşıyarak** yaprağa indirip silmek için takas yapılır:

- **Sol çocuk varsa:** predecessor (sıralamada önceki) sol alt ağaçtadır (daha aşağıda). node'un **item'ını predecessor'ın item'ıyla takas et**, sonra predecessor'ı **özyinelemeli sil**.
- **Sağ çocuk varsa (sol yoksa):** simetrik — successor ile takas et, successor'ı sil.

Her takasta düğüm bir seviye aşağı iner; toplam iş ağaç yüksekliğiyle orantılı $\to O(h)$. (Burada ilk kez düğümlerin *içeriğini* değiştiriyoruz, düğümün kendisini değil.)

> *"trees will not preserve connections — that's just the name of the game."* — Demaine, 47:02

Şekil 16.7 somut bir örnekte (`build_bst([16,8,24,4,12,2,6,10,14,13])`, kök 16'yı sil) bu takas zincirini üç kare hâlinde gösterir: $16 \leftrightarrow 14$, sonra $16 \leftrightarrow 13$, sonra yaprakta kopar; son traversal $[2, 4, 6, 8, 10, 12, 13, 14, 24]$.



Şekil 16.7: İç düğüm silme = item TAKASI zinciri (L6 §8, **İMZA** figür; `build_bst([16,8,24,4,12,2,6,10,14,13])`, $h = 4$, kök 16 silinir). Düğüm DAİRELERİ yapısal konumda sabit kalır; yalnız İÇERİK aşağı iner (Demaine 47:02). **Kare 1:** 16 (kök) amber dolgu "SİL"; predecessor yolu $16 \to 8 \to 12 \to 14$ (kesikli amber); çift-başlı takas oku $16 \leftrightarrow 14$. **Kare 2:** 14 köke çıktı, 16 bir seviye indi (14'ün eski yeri); 16'nın sol çocuğu 13 ile ikinci takas $16 \leftrightarrow 13$. **Kare 3:** 16 artık yaprakta → parent'tan koparılır ( ). Altta son in-order şeridi $[2, 4, 6, 8, 10, 12, 13, 14, 24]$ (16 yok ✓). Her takas item'ı bir seviye indirir → toplam $\leq h = 4$ takas $= O(h)$.

Builder Notu — İşaretçi Cerrahisi ve Korunan Değişmez

delete, ağaç işlemlerinin ortak iskeletini açığa çıkarır: her işlem birkaç işaretçiyi keser/bağlar, ve doğruluğu **korunan bir değişmezle (invariant)** kanıtlanır.

- **Parent-child invariant:** Her düğümde `node.left.parent is node` (ve right için). insert_after yeni bir bağ kurarken bu değişmezi *hem yukarı hem aşağı* tamamlamalı (`new.parent = ...` unutulursa successor yukarı-yürüyüşü kırılır). delete'te item takası bu değişmezi hiç bozmaz — çünkü daireler yerinde kalır, yalnız içerik değişir; bu yüzden takas "güvenli" bir tekniktir.
- **Her işlem bir ispat:** subtree_first/successor/insert_after/delete doğruluğu, "traversal sırası korunur" + "parent-child tutarlı" değişmezlerinin her adımda sağlandığını göstermekle ispatlanır. Bir builder için ders: işaretçi koduna güven, ispatla gelir — testte bu iki değişmezi assert et (`tree_traversal == beklenen`, `check_parent_child`).
- **Genel ilke:** Bağlı yapılarda (linked list, ağaç, graf) her mutasyon bir "cerrahi"dir; bug'lar neredeyse her zaman yarım kalmış bir bağ (dangling/half-updated pointer) yüzündendir. Değişmezi yaz, her işlemden sonra kontrol et.

Tek cümle: *İşaretçi cerrahisinin güvenliği, her kesimden sonra aynı değişmezi geri kurmaktan gelir — ağaç kodunun doğruluğu, hız değil, değişmez disiplinidir.*

16.10 Sequence ve Set Olarak Ağaç

Aynı ağaç iki arayüzü de gerçekleştirir, yalnızca traversal sırasının *anlamı* değişir:

- **Dizi (sequence):** traversal sırası = sequence sırası $(x_0, x_1, \ldots, x_{n-1})$.
- **Küme (set):** traversal sırası = **artan anahtar sırası**.

Set durumunda **BST özelliği (binary search tree property)** doğar: her düğümde, sol alt ağacın tüm anahtarları $<$ düğüm $<$ sağ alt ağacın tüm anahtarları.

> *"this is something called the binary search tree property, BST property."* — Demaine, 49:11

find(k): kökten başla, k'yı düğümle karşılaştır; küçükse sola, büyükse sağa in — ikili aramanın ağaç hâli, $O(h)$. **find_prev/find_next:** ağaçtan düşersen (k yoksa), son denenen yön sana önceki ya da sonrakini verir; diğerini predecessor/successor ile bulursun. (Diziyi tam gerçekleştirmek biraz daha iş ister — sonraki ders.)

Builder Notu — BST = Dinamik İkili Arama (B-tree/SQL İndeksin Atası)

BST özelliği, Ders 4'teki sıralı dizi ikili aramasını *dinamik* hâle getirir: arama hâlâ $O(h)$, ama artık ekleme/silme de $O(h)$ — kaydırma yok.

- **Dinamik ikili arama:** Sıralı dizi find'i $O(\log n)$ yapar ama insert/delete $O(n)$ (kaydırma). BST, find + find_prev/next + insert + delete'in *hepsini* $O(h)$ yapar; $h = O(\log n)$ sağlanırsa (Ders 10), sıralı dizinin tüm avantajı + dinamiklik elde edilir.
- **B-tree / B+-tree = disk-dostu genelleme:** Her düğümde 2 değil B çocuk tutarak ağaç sığlaşır ($h \approx \log_B n$); bu, bir disk bloğu = bir düğüm eşlemesiyle disk erişimini minimize eder. PostgreSQL, MySQL InnoDB, SQLite — hepsinin varsayılan indeksi bir B+-tree'dir; "varsayılan SQL indeksi

bir dengeli ağaçtır" sözünün kökü budur (hash/GIN gibi özel indeks türleri ayrı uzmanlıklardır).
- **find_prev/find_next = en yakın komşu:** Hash tablosunun yapamadığı "en yakın daha büyük/küçük anahtar" sorgusu (range query, `BETWEEN`, autocomplete) ağacın doğal yaptığı şeydir — bu yüzden veritabanı sıralı taramaları hash değil ağaç indeksi ister.

Tek cümle: *BST, ikili aramayı statik diziden kurtarıp dinamik yapar; B-tree onu diske taşır — modern her veritabanı indeksinin atası bu derstir.*

16.11 Bu Dersin Özeti

1. **İkili ağaç**: düğüm (item + `parent/left/right`); kök parent'sız, yaprak çocuksuz.
2. **İki işaretçi** (left/right) logaritmik yükseklik sağlar; bağlı listenin lineer derinliğini aşar.
3. **Derinlik** (köke kenar) ve **yükseklik** (en derin yaprağa kenar); ağacın yüksekliği h.
4. **Traversal sırası** (in-order): sol → düğüm → sağ; örtük, asla diziye yazılmaz.
5. **subtree_first / successor / insert_after / delete**: hepsi $O(h)$.
6. **Sequence**: traversal = sıra; **Set**: traversal = artan anahtar (BST özelliği).
7. **find / find_prev / find_next**: BST'de ikili arama, $O(h)$.

! Tek Bir Cümle

İkili ağaç, sıralı bir düzeni örtük traversal sırasıyla temsil eder ve tüm işlemleri ağaç yüksekliği h ile sınırlar; geriye tek soru kalır — h'yi nasıl küçük ($\log n$) tutarız?

16.12 Kontrol Soruları

i Soru 1: Traversal (geziş) sırası neden açıkça bir dizide tutulmaz?

Cevap: Traversal sırasını bir dizide tutmak, ortaya ekleme/silmede tüm öğeleri kaydırmayı gerektirir → $O(n)$. İkili ağacın tüm amacı bu düzeni *örtük* (işaretçilerle) tutmaktır; böylece insert_after, delete gibi işlemler yalnızca birkaç işaretçi değiştirerek $O(h)$ zamanda düzeni günceller. Sıra "kafadadır"; bilgisayarda yalnızca düğümler ve işaretçiler saklanır.

i Soru 2: successor'da neden iki durum var? Her birinde nereye bakılır?

Cevap: *Durum 1 — sağ çocuk var:* node'dan sonra gelen her şey sağ alt ağaçtadır; bunların ilki (en solu) successor'dır → `subtree_first(node.right)`. *Durum 2 — sağ çocuk yok:* node'dan sonra gelecek hiçbir şey altında değil; sola-dal yukarı çıkıp, node'u sol tarafında bırakan en yakın atayı buluruz (o atanın sol alt ağacı node'u içerir, kendisi sonra gelir). İkisi de $O(h)$.

Soru 3: delete'te neden item takası yapılır, düğüm doğrudan silinmez?

Cevap: Yaprak olmayan bir düğümü doğrudan silmek ağacı koparır (çocuklarına giden işaretçiler boşa düşer). Bunun yerine, düğümün item'ını predecessor/successor'ın item'ıyla takas ederiz; bu, "silinecek" item'ı ağaçta bir seviye aşağı taşır. Tekrarlayarak item bir yaprağa iner ve yaprak güvenle silinir. Düğüm daireleri yerinde kalır, yalnızca içerikleri değişir; toplam iş $O(h)$.

Soru 4: Bir kümeyi sıralı dizi yerine ikili ağaçla tutmanın kazancı nedir?

Cevap: Sıralı dizi find'i $O(\log n)$ (ikili arama) yapar ama insert/delete $O(n)$'dir (kaydırma). İkili ağaç (BST), find'i ve find_prev/next'i $O(h)$ yapar *ve* insert/delete'i de $O(h)$'de tutar — düzeni kaydırma olmadan, işaretçilerle günceller. $h = O(\log n)$ sağlanırsa (sonraki ders), tüm işlemler $O(\log n)$ olur; sıralı dizinin statik kısıtı kalkar.

16.13 Egzersizler

Egzersiz 1. Verilen bir ikili ağaç için traversal (in-order) sırasını elle çıkar; sonra `subtree_first(kök)` ve `successor` ile baştan sona dolaşarak aynı sırayı yeniden üret.

Egzersiz 2. `subtree_last` ve `predecessor` işlemlerini, `subtree_first/successor`'ın simetriği olarak tanımla (sağ $\leftrightarrow$ sol).

Egzersiz 3. insert_after Durum 2'de, successor'ın neden sol çocuğu olamayacağını tek cümlede ispatla (`subtree_first`'in tanımına dayan).

Egzersiz 4. Python'da bir BST düğüm sınıfı yaz ve `find(k)` ile `find_next(k)`'yi gerçekleştir:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = self.right = self.parent = None

def find(node, k):
    while node and node.key != k:
        node = node.left if k < node.key else node.right
    return node
```

Egzersiz 5. delete'te, hem sol hem sağ çocuğu olan bir düğüm için neden "tek bir durum" (sadece sol VEYA sadece sağ) yeterli olur? Demaine'in "iki durumdan biri beni mutlu eder" gözlemini açıkla.

16.14 Sonraki Ders İçin Hazırlık

Ders 10 (L7): İkili Ağaçlar — Bölüm 2

Erik Demaine ile, bugün açık bıraktığımız soruyu kapatıyoruz: ağacın yüksekliği h'yi $O(\log n)$ olmaya nasıl zorlarız? **AVL ağaçları** (dengeli ikili ağaçlar) ve **rotasyon** işlemiyle, tüm $O(h)$ işlemleri $O(\log n)$ garantisine çeviriyoruz. (Not: ders akışında araya **Problem Oturumu 4** girer.)

Ders 10 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle **Egzersiz 4**'ü (BST find) çöz.
- **Dört $O(h)$ işlemini** (subtree_first, successor, insert_after, delete) ezberden anlat.
- Ana cümleyi tekrar oku: *"Geriye tek soru kalır — h'yi nasıl küçük tutarız?"*

16.15 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
İkili ağaç (düğüm)	item + parent/left/right; kök parent'sız, yaprak çocuksuz	Böl. 2
Derinlik / yükseklik	Köke kenar / en derin yaprağa kenar; ağaç yüksekliği h	Böl. 4
Traversal sırası	In-order: sol → düğüm → sağ; örtük	Böl. 5
subtree_first	Alt ağacın ilki; sola git; $O(h)$	Böl. 6
successor	Sıradaki düğüm; sağ varsa subtree_first, yoksa sola-dal yukarı; $O(h)$	Böl. 6
insert_after	Sağ yoksa `node.right`; varsa successor'ın sol çocuğu; $O(h)$	Böl. 7
delete	Yaprak→kopar; değilse pred/succ item takası + özyinele; $O(h)$	Böl. 8
BST özelliği	sol anahtarlar $<$ düğüm $<$ sağ anahtarlar; find $O(h)$	Böl. 9

16.16 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "sıralı düzeni nasıl ucuza dinamik tutarız" sezgisini kurar — köprülerin özeti:

1. **BST / dengeli ağaç** → veritabanı indeksleri (B-tree, B+-tree): varsayılan SQL indeksi ve dosya sistemi dizinleri bu fikrin disk-dostu hâli.
2. **Traversal sırası** → sıralı yineleme: bir set'i sıralı gezmek (range query, `BETWEEN`) ağaçta doğal $O(n)$ traversal.
3. **find_prev / find_next** → "en yakın komşu" sorguları: hash tablosunun yapamadığı, ağacın doğal yaptığı şey.
4. $O(h) \to O(\log n)$ → denge disiplini: bir veri yapısının garantisi, en kötü durum yüksekliğini

sınırlamaya bağlıdır (sonraki ders).

5. **İşaretçi manipülasyonu + invariant** → her ağaç işleminin doğruluğu, korunan bir değişmezle (BST özelliği, parent-child tutarlılığı) ispatlanır.
6. **Örtük düzen** → genel ilke: pahalı bir gösterimi (sıralı dizi) açıkça tutmak yerine, ucuz güncellenebilir bir yapıyla (ağaç) örtük temsil etmek.

! Tek bir şey alıp gideceksen

İkili ağaç, "sıralı dizi gibi hızlı ara ama bağlı liste gibi esnek değiştir" hayalini gerçekleştirir — düzeni işaretçilerle örtük tutarak. Tüm işlemler $O(h)$'dir; tek görev, h'yi $O(\log n)$'de tutmak (bir sonraki ders: AVL).

17 İkili Ağaçlar — Bölüm 2: AVL

Dizi ağaçları (subtree_at, size), alt ağaç zenginleştirme (O(1) güncelleme kuralı), rotasyon (traversal sırası korunur), AVL skew ∈ {−1, 0, +1} ve N_h Fibonacci-benzeri üstel büyüme → h ≤ 2 log n garantisi

ℹ Bölüm bilgisi

- **Demaine'in videosu:** YouTube — Lecture 7: Binary Trees, Part 2: AVL (≈54 dk)
- **OCW sayfası:** MIT 6.006 Lecture 7: Binary Trees, Part 2: AVL
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 10 (L7)
- **Hoca:** Erik Demaine
- **Okuma süresi:** ≈26 dk

17.1 Bu Derste Ne Var?

Ders 9 (L6) ikili ağaçları kurdu: tüm işlemler $O(h)$ (h = yükseklik). Açık kalan tek soru vardı — h'yi nasıl küçük tutarız? Bu ders onu kapatır: **AVL ağaçları** ve **rotasyon** ile $h = O(\log n)$ garanti edilir, böylece tüm $O(h)$ işlemler $O(\log n)$ olur.

Üç temel kavram bu derste yan yana gelir:

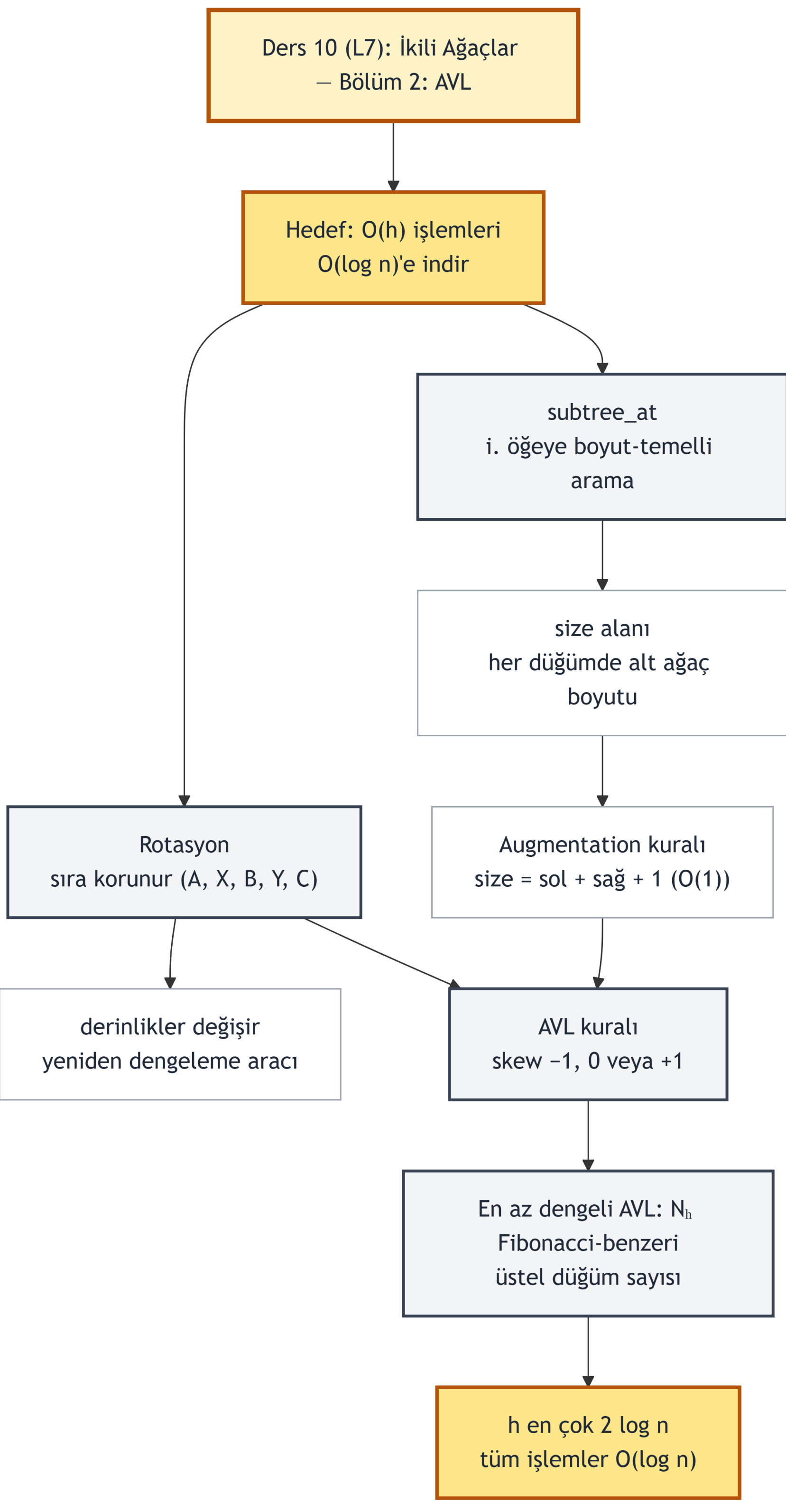
1. **Dizi ağaçları + alt ağaç zenginleştirme** — `size` alanıyla i. öğeye $O(h)$'de erişim (subtree_at).
2. **Rotasyon** — traversal sırasını bozmadan ağacı yeniden dengeleyen araç.
3. **AVL / yükseklik dengesi** — her düğümde skew $\in \{-1, 0, +1\}$; bu, $h \leq 2 \log n$'i garanti eder.

> *"the goal of today is to take all of these operations that run in order h time and get them to run in order log n time."* — Demaine, 3:05

Builder Notu — Denge = Garanti Mühendisliği

İkili ağaç tek başına $O(h)$'dir; bugün yapılan tek şey, o h'ye bir **üst sınır garantisi** eklemek — yapıyı yeniden kurmadan, sadece dengeleyerek.

- **Geriye → Ders 3 (ikili arama):** BST'deki find, sıralı dizideki ikili aramanın ağaç hâlidir — ama artık *dinamik* (insert/delete $O(\log n)$).
- **Geriye → Ders 9 (L6):** bugün, dünkü $O(h)$ işlemlerin üstüne yalnızca denge ekliyoruz; alt yapı (düğüm, traversal, successor) aynı.
- **İleriye → veritabanı:** B-tree / B+-tree, dengeli ağacın disk-dostu hâlidir; çoğu SQL indeksi ve dosya sistemi dizini bu fikre dayanır (varsayılan indeks tipik olarak bir dengeli ağaçtır).



Şekil 17.1: Ders 10'un (L7) kavram haritası: hedef, dünkü O(h) işlemleri O(log n)'e indirmek. İki köprü ekleniyor — (1) dizi ağacında subtree_at, her düğümdeki size alanıyla boyut-temelli ikili arama yapar; size ise alt ağaç zenginleştirme kuralından (node.size = sol + sağ + 1, O(1) güncelleme) gelir. (2) rotasyon, traversal sırasını (A, X, B, Y, C) ASLA bozmadan derinlikleri değiştirir → yeniden dengeleme aracı. AVL kuralı bunu skew $\in \{-1, 0, +1\}$ ile sabitler; en az dengeli AVL'nin düğüm sayısı N_h Fibonacci-benzeri (üstel) olduğundan yükseklik $h \le 2 \log n$ çıkar. Sonuç: her sequence/set işlemi O(log n).

- **İleriye → order-statistics tree:** `size` zenginleştirmesi, "rank" ve "k. en küçük" sorgularını $O(\log n)$'de yapar — sıralama istatistikleri.

Tek cümle: *İkili ağaca `size/height` gibi alt ağaç özellikleri ekleyip rotasyonla yükseklik dengesi (AVL) tutarsak, her işlem $O(\log n)$ olur — dengeli ağacın tüm gücü budur.*

17.2 Geçen Dersten: O(h) İşlemler ve Bugünkü Hedef

Ders 9'da (L6) ikili ağaç düğümü `item` + `node.left/node.right/node.parent` tutuyordu; **yükseklik** (en uzun aşağı yoldaki kenar sayısı) tanımlandı; **traversal sırası** (sol → kök → sağ) örtük düzeni kodluyordu. subtree_first/last, predecessor/successor, insert/delete — hepsi $O(h)$.

Sorun: en kötü durumda h lineerdir (yalnız sağ işaretçiler kullanılan zincir ağaç). Bugün $h = O(\log n)$ garanti edip tüm işlemleri $O(\log n)$'e indireceğiz — veri yapısını yeniden kurmadan, sadece dengeleyerek.

17.3 Küme Ağaçları = İkili Arama Ağaçları (BST)

Küme için traversal sırasını **artan anahtar** yaparsak, `subtree_find` bir **ikili arama** olur: kökten başla, aranan anahtar düğümünkinden küçükse `node.left`'e, büyükse `node.right`'e in, eşitse bulundu.

> *"set binary trees are called binary search trees, because they're the tree version of binary search."*
> — Demaine, 6:30

Bunu doğru kılan **BST özelliği**: bir düğümün sol alt ağacındaki tüm anahtarlar $<$ düğüm $<$ sağ alt ağacındaki tüm anahtarlar (özyinelemeli). Bu, traversal sırası tanımının (sol önce, sağ sonra) doğrudan sonucudur. find_prev/find_next: ağaçtan düşersen, son denenen yön sana komşuyu verir. $O(h)$.

17.4 Dizi Ağaçları: subtree_at

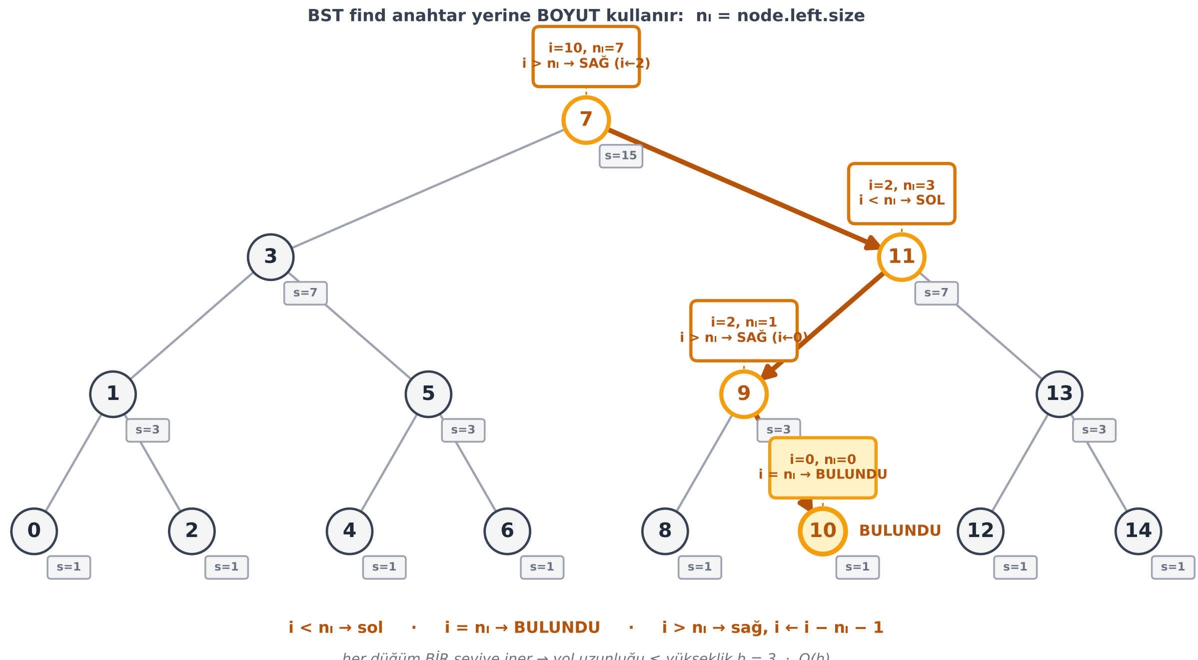
Dizi için traversal sırasını **sequence sırası** yaparız; ama "i. öğeyi getir" (`subtree_at`) için artık anahtar yok — *sıra numarasıyla* aramamız gerekir. Anahtar fikir: her düğümde **alt ağaç boyutu** `size`'ı bil.

Çalışılan Örnek — subtree_at. n_ℓ = `node.left.size` (sol alt ağaçtaki düğüm sayısı) olsun. i. öğeyi ararken:

- $i < n_\ell$ → öğe sol alt ağaçta → `subtree_at(node.left, i)`.
- $i = n_\ell$ → kökün indeksi tam n_ℓ'dir → node'u döndür.
- $i > n_\ell$ → öğe sağ alt ağaçta → `subtree_at(node.right, i - n₁ - 1)` (sol için n_ℓ, kök için 1 çıkar).

Bu, BST find'in *anahtar yerine boyut* kullanan ikizidir; $O(h)$. Bununla get_at/set_at (düğümü bul, item'ı oku/değiştir) ve — ilk kez! — insert_at/delete_at (i'yi bul, subtree_insert_before ile araya ekle) yapılır; tüm indeksler kendiliğinden güncellenir, çünkü indeks saklanmıyor.

Şekil 17.2 bunu somut bir örnekte gösterir: `build_avl(range(15))` mükemmel dengeli ağacında (kök 7, $h = 3$) $i = 10$ aranır; yol $7 \to 11 \to 9 \to 10$ ile öğe bulunur.



Şekil 17.2: subtree_at — size ile boyut-temelli ikili arama, O(h) (L7 §3). Deterministik örnek `build_avl(range(15))`: kök 7, h = 3, size = 15. Aranan i = 10. Her yol düğümünün üstünde karar kutusu (motordan): **7** (n_l = 7, i = 10) → i > n_l → SAĞ, i ← 10 − 7 − 1 = 2; **11** (n_l = 3, i = 2) → i < n_l → SOL; **9** (n_l = 1, i = 2) → i > n_l → SAĞ, i ← 2 − 1 − 1 = 0; **10** (n_l = 0, i = 0) → i = n_l → BULUNDU (amber dolgu). Her düğümün yanında küçük slate size rozeti (s = N). BST find anahtar yerine BOYUT kullanır: n_l = node.left.size. Her adım bir seviye iner → yol uzunluğu ≤ h = 3 → O(h).

17.5 Alt Ağaç Zenginleştirme (Subtree Augmentation)

Peki `size`'ı nasıl $O(1)$'de biliriz? **Alt ağaç zenginleştirmesiyle**: her düğüm sabit sayıda ekstra alan tutar. Bir **alt ağaç özelliği (subtree property)**, düğümün *çocuklarının* özelliklerinden $O(1)$'de hesaplanabilen, "aşağı bakan" bir niceliktir.

> *"I'm going to define a subtree property to be something that can be computed from the properties of the node's children."* — Demaine, 16:10

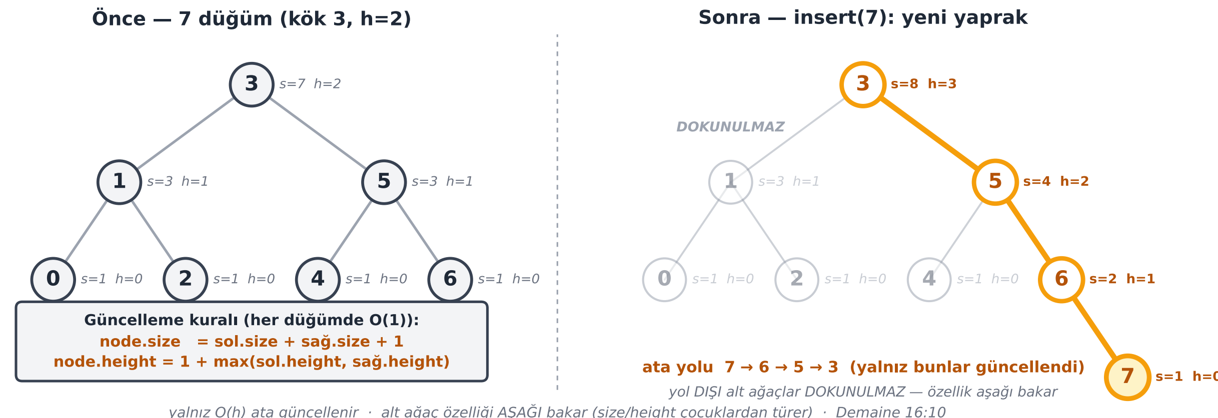
`size` tam da böyledir:

$$\texttt{node.size} = \texttt{node.left.size} + \texttt{node.right.size} + 1$$

“node.size equals node.left.size plus node.right.size plus 1.” — Demaine, 17:43

Bu bir **güncelleme kuralıdır** ($O(1)$). `size` saklanır, özyinelemeli yeniden hesaplanmaz (`return node.size` → $O(1)$). Insert/delete sonunda bir **yaprak ekler/siler**; yalnızca o yaprağın **ataları** (ancestors) değişir → ata yolunu yukarı çıkıp her birinde güncelleme kuralını uygula, $O(h)$. Bu, tümevarımla doğrudur: alt çocuklar zaten güncelse, kural ile düğüm $O(1)$'de güncellenir.

Şekil 17.3 iki panelde bunu gösterir: `build_avl(range(7))` (kök 3, $h = 2$) ağacına 7 eklendiğinde yalnız ata yolu $7 \to 6 \to 5 \to 3$ güncellenir; yolun dışındaki alt ağaçlar dokunulmaz kalır.



Şekil 17.3: Alt ağaç zenginleştirme — yaprak değişimi yalnız ATALARI günceller, O(h) (L7 §4). **Sol panel:** insert öncesi `build_avl(range(7))` (kök 3, h = 2); her düğümün yanında size + height rozeti; altta O(1) güncelleme kuralı kutusu (node.size = sol.size + sağ.size + 1; node.height = 1 + max(sol.height, sağ.height)). **Sağ panel:** avl_insert(r, 7) sonrası; yeni yaprak 7 amber DOLGU; ata yolu 7 → 6 → 5 → 3 (yapraktan köke) amber KALIN kenar — yalnız bu düğümlerin rozetleri güncellendi. Yolun DIŞINDAki alt ağaçlar soluk + DOKUNULMAZ etiketi. Alt ağaç özelliği AŞAĞI bakar (size/height çocuklardan türer) → yalnız O(h) ata güncellenir (Demaine 16:10).

Builder Notu — Order-Statistics: rank ve k. En Küçük

`size` zenginleştirmesi, dengeli ağacı bir **order-statistics tree**'ye çevirir: yalnız “i. öğe” değil, ters yönü de — bir öğenin **rank**'ı (kaçıncı sırada) ve “k. en küçük” sorguları $O(\log n)$'de yanıtlanır.

- **rank(x):** Kökten x'e inerken, her sola dönüşte o düğümün $n_\ell + 1$'ini (sol alt ağaç + düğüm kendisi) atlamış olursun; toplam atlanan, x'in sıra numarasıdır. Tam `subtree_at`'in tersi.
- **k. en küçük:** doğrudan `subtree_at(kök, k)` — Şekil 17.2'teki yürüyüşün ta kendisi.
- **Pratik kullanım:** “medyanın altında kaç öğe var”, “yüzdelik dilim”, “şu fiyattan ucuz kaç ürün” gibi sorgular; bir sıralı dizide $O(n)$ olurdu, order-statistics tree'de $O(\log n)$. (Araya giren Problem Oturumu 4 tam bu tür zenginleştirme problemlerini işler.)

Tek cümle: *tek bir `size` alanı, ağacı hem "i. öğe"yi hem "bu öğe kaçıncı"yı $O(\log n)$'de bilen bir sıralama-istatistiği yapısına dönüştürür.*

17.6 Hangi Özellikler Tutulabilir?

Alt ağaç özelliği olan her şey: **toplam, çarpım, min, max, kare toplamı** — alt ağaçtaki düğümlerin herhangi bir alanı üzerinden. (`size` aslında "her düğüm için 1'in toplamı"dır.)

Ama **tutulamayanlar** vardır:

- **İndeks (index):** *global*'dir — başa bir düğüm eklersen *tüm* düğümlerin indeksi değişir. Bir düğümün indeksi, solundaki tüm düğümlere (alt ağaç dışı) bağlıdır.
- **Derinlik (depth):** benzer şekilde yukarı-bakan, alt ağaç özelliği değil.

> *"Index is not a subtree property, and that's why we can't maintain it — because it depends on all of the nodes in the tree."* — Demaine, 25:35

Kural: yalnızca **aşağı-bakan (alt ağaca özgü)** özellikler tutulabilir; global özellikler değil.

Builder Notu — Yerel (Kompozisyonel) Olan Tutulur

Bu, veri yapısı tasarımının ötesine geçen bir ilke: *bir niceliği verimli güncel tutabilmen, onun yerel (kompozisyonel) olmasına bağlıdır.* Yerel = "yalnızca alt parçalardan hesaplanır"; global = "tüm bütüne bağlı".

- **Aşağı-bakan vs global:** `size`/`sum`/`max` çocuklardan $O(1)$'de türer → yaprak değişiminde yalnız $O(h)$ ata güncellenir. `index`/`depth` tüm ağaca bağlı → tek ekleme her şeyi kaydırır, $O(n)$.
- **Dağıtık sistem analojisi:** Bir ağaç-toplama (tree aggregation) veya MapReduce'da yalnız **birleştirilebilir (associative/composable)** agregalar — sum, count, min, max — kısmi sonuçlardan ucuza güncellenir; "global sıra numarası" gibi nicelikler her düğüme tüm veriyi gerektirir.
- **Genel ders:** "Bunu $O(\log n)$'de tutabilir miyim?" sorusunun cevabı, "bu nicelik çocuklarımdan hesaplanabilir mi?" sorusuyla aynıdır.

Tek cümle: *verimli bakım, yerelliğin (kompozisyonelliğin) bir sonucudur — global bir nicelik hiçbir akıllı işaretçi numarasıyla ucuza tutulamaz.*

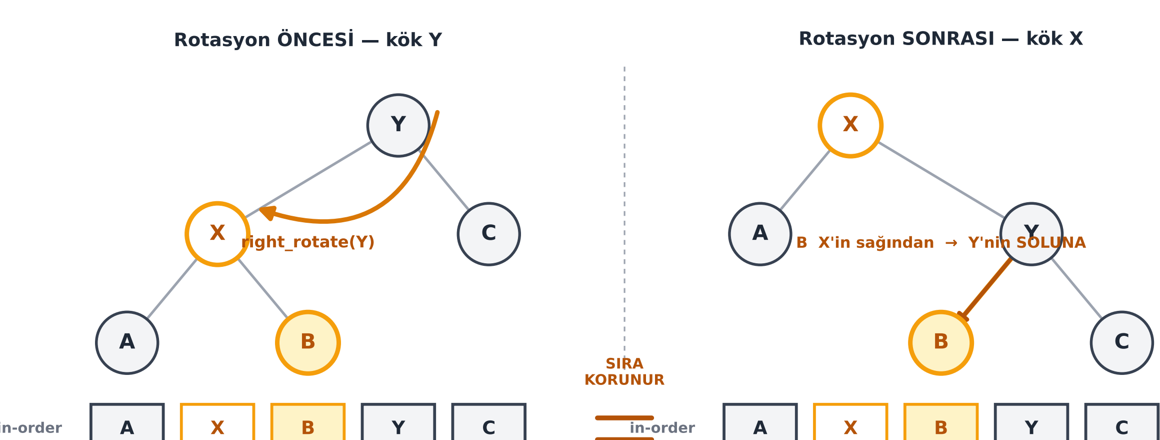
17.7 Ağaç Rotasyonu

Dengeyi sağlamak için yeni bir araç gerekir: **rotasyon**. Tek işi ağacı yeniden dengelemek; **traversal sırasını asla değiştirmemelidir** (traversal sırası temsil edilen veridir — dokunulmaz).

> *"a rotation… a tool for rebalancing the tree. So it should not change the data that's represented by the tree. … Traversal order is sacrosanct."* — Demaine, 30:07

Çalışılan Örnek — rotasyon. Bir y düğümü ve sol çocuğu x düşün; alt ağaçlar A (x'in solu), B (x'in sağı), C (y'nin sağı). **right_rotate(y)**: x yukarı, y aşağı geçer; B, x'ten kopup y'nin soluna bağlanır. Her iki çizimde de traversal sırası A, X, B, Y, C kalır (üçgenler özyinelemeli). Yani rotasyon sırayı korur ama derinlikleri değiştirir: x yukarı (sığlaşır), y aşağı (derinleşir), bu da yeniden dengeleme imkânı verir. (Simetriği **left_rotate(x)**.) Aynı ağaç düzeni üstel sayıda farklı ağaçla temsil edilebilir; rotasyon bu fazlalığı kullanır.

Şekil 17.4 iki paneli yan yana koyar (öncesi/sonrası); alttaki in-order şerit A, X, B, Y, C her iki panelde aynen aynı kalır.



Şekil 17.4: right_rotate — yapı değişir, traversal sırası KORUNUR (A, X, B, Y, C) (L7 §6, İMZA figür). **Sol panel (öncesi):** kök Y, sol çocuğu X; alt ağaçlar A = X.left, B = X.right (amber dolgu = yer değiştirecek), C = Y.right. Kavisli amber ok right_rotate(Y)'yi gösterir. **Sağ panel (sonrası):** kök X (amber çerçeve); B artık X'in sağından Y'nin SOLUNA taşındı (amber taşıma oku). HER iki panelin altında AYNI in-order şeridi A | X | B | Y | C; ortada SIRA KORUNUR rozeti + eşittir işareti. Rotasyon = O(1) işaretçi takası + augmentation güncelle; in-order dizilim kutsaldır (Demaine 30:07). Motor doğrulaması: right_rotate öncesi avl_to_list = sonrası avl_to_list = [A, X, B, Y, C].

Builder Notu — Rotasyon: Tüm Dengeli Ağaçların Ortak İlkeli

Rotasyon, AVL'ye özgü değil — **tüm kendini-dengeleyen ağaçların** ortak ilkel (primitive) işlemidir. "Sırayı bozmadan derinlik değiştir" mekanizması her yerde aynıdır; ağaçlar yalnızca *ne zaman* döndüreceklerine dair kuralda ayrışır.

- **red-black tree:** Renk kuralları + rotasyon; çoğu standart kütüphane `map/set`'i (örneğin C++ STL, Java TreeMap) red-black ağacıdır.
- **splay tree:** Her erişimde erişilen düğümü rotasyonlarla köke taşır — "son kullanılan hızlı" (self-adjusting).
- **treap:** Rastgele öncelik + BST anahtarı; rotasyonla heap-özelliği korunur.

Hepsinde değişmez aynı: rotasyon $O(1)$'dir ve traversal sırasını korur; yalnız hangi düğümde tetiklendiğinin politikası farklıdır.
Tek cümle: *rotasyonu öğrenmek tek bir ağacı değil, kendini-dengeleyen ağaç ailesinin tamamını öğrenmektir — fark, kuralda; ilkel hep aynı.*

17.8 AVL / Yükseklik Dengesi

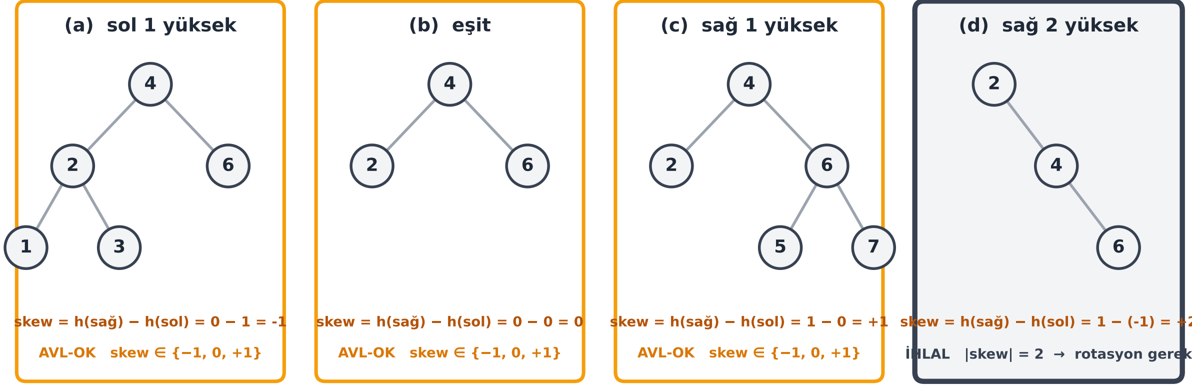
Hangi denge kuralı? Her düğüm için **skew** tanımla:

$$\text{skew}(\text{node}) = \text{height}(\text{node.right}) - \text{height}(\text{node.left})$$

AVL kuralı: her düğümde skew $\in \{-1, 0, +1\}$ — yani sol ve sağ alt ağaç yükseklikleri en fazla 1 fark eder.

> *"skew of the node — I want this to always be minus 1, 0, or plus 1."* — Demaine, 35:11

Şekil 17.5 skew tanımını dört mini ağaç üzerinde gösterir: üçü kural içinde $(-1, 0, +1)$, biri ihlal $(+2)$.



Şekil 17.5: AVL kuralı — her düğümde skew ∈ {−1, 0, +1} (L7 §7; Demaine 35:11). Dört mini ağaç (skew = h(sağ) − h(sol), motordan okunur): **(a)** sol 1 yüksek → skew = 0 − 1 = −1, AVL-OK; **(b)** iki taraf eşit → skew = 0 − 0 = 0, AVL-OK; **(c)** sağ 1 yüksek → skew = 1 − 0 = +1, AVL-OK; **(d)** sağ 2 yüksek → skew = 1 − (−1) = +2, İHLAL (kalın slate çerçeve) → rotasyon gerek. İlk üç panel amber ONAY çerçevesi, dördüncü kalın slate çerçeve. Yükseklikler height alanından okunur (augmentation) → skew O(1); |skew| = 2 olunca ata yolunda rotasyon onarır.

17.9 Yükseklik Dengesi → Denge

Çalışılan Örnek — neden $h \le 2\log n$? En *az* dengeli yükseklik-dengeli ağacı düşün: her düğümde sağ, soldan 1 yüksek. Yükseklik h olan böyle bir ağaçtaki minimum düğüm sayısı N_h:

$$N_h = N_{h-1} + N_{h-2} + 1$$

Bu **Fibonacci benzeri** bir yinelemedir ($+1$ olmadan tam Fibonacci); Fibonacci sayıları altın oran kuvvetiyle, yani **üstel** büyür. Üstel alt sınırı basitçe görmek için: $N_{h-1} \ge N_{h-2}$ olduğundan $N_h > 2 \cdot N_{h-2}$; bu da $N_h \ge 2^{h/2}$ verir.

> *"It's like Fibonacci numbers… Fibonacci numbers grow as a golden ratio to the n… this is exponential."* — Demaine, 39:48

N_h düğüm sayısı h'de üstel olduğundan, h düğüm sayısında **logaritmiktir**: $h \le 2\log n$. Yani AVL ağaçları her zaman dengelidir (gereken seviye sayısının en çok iki katı).

> *"h is at most 2 log n. So AVL trees are always quite balanced."* — Demaine, 42:09

Şekil 17.6 bunu iki panelde gösterir: solda `build_fibonacci_tree(4)` ($h = 4$, $n = N_4 = 12$), sağda N_h dizisinin $2^{h/2}$ alt sınırına karşı üstel büyümesi.

Builder Notu — Üstel Düğüm = Logaritmik Yükseklik

Bu, analiz disiplininin saf bir örneğidir: *bir yapının yükseklik garantisi, "en kötü durumda en az kaç düğüm gerekir" sorusunu cevaplamaktan gelir.*

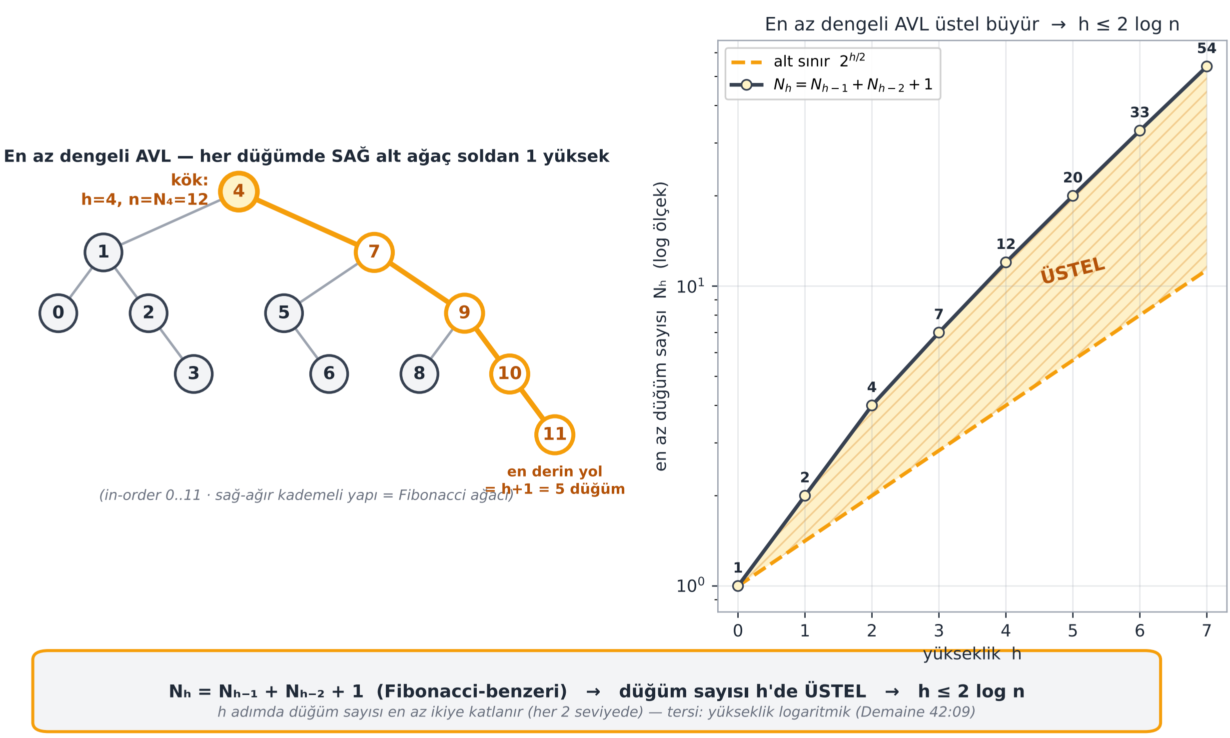
- **Mantık zinciri:** "yükseklik h için minimum düğüm N_h üstel" $\Longrightarrow$ "n düğüm en çok logaritmik yükseklik üretir". Üstel alt sınır (düğüm tarafında) doğrudan logaritmik üst sınıra (yükseklik tarafında) çevrilir.
- **Aynı kalıp her yerde:** ikili arama ağacının $\Omega(\log n)$ karşılaştırma alt sınırı (karar ağacı yaprak sayısı $\ge n + 1$), heap'in $\log n$ yüksekliği, B-tree'nin $\log_B n$ derinliği — hepsi "az düğümle çok yükseklik yapılamaz" argümanının çeşitlemeleridir.
- **Builder dersi:** Bir veri yapısının hız garantisini ispatlamak istiyorsan, çoğu zaman doğrudan zamanı değil, **boyut/yükseklik arasındaki üstel ilişkiyi** sınırlarsın.

Tek cümle: *"$h \le 2\log n$" bir hesaplama değil, bir sayma argümanıdır — en az dengeli ağacın bile üstel düğüm istemesi, yüksekliği logaritmaya hapseder.*

17.10 Yükseklik de Bir Alt Ağaç Özelliği

Dengeyi kontrol etmek için her düğümün yüksekliğini bilmeliyiz. İyi haber: **yükseklik bir alt ağaç özelliğidir**:

$$\texttt{node.height} = 1 + \max(\texttt{node.left.height}, \texttt{node.right.height})$$



Şekil 17.6: En az dengeli AVL = Fibonacci ağacı → yükseklik h ≤ 2 log n (L7 §8; Demaine 42:09). **Sol panel:** build_fibonacci_tree(4) — h = 4, n = N_4 = 12; her düğümde sağ alt ağaç soldan TAM 1 yüksek (en az dengeli yükseklik-dengeli ağaç, sağ-ağır kademeli); in-order 0..11; en derin yol (sağ omurga, h + 1 = 5 düğüm) amber vurgulu. **Sağ panel (semilog):** N_h noktaları (motor dizisi [1, 2, 4, 7, 12, 20, 33, 54], h = 0..7) + alt sınır 2^(h/2) (amber kesikli); aradaki taralı bölge ÜSTEL büyümeyi gösterir; her h için N_h ≥ 2^(h/2). $N_h = N_{h-1} + N_{h-2} + 1$ (Fibonacci-benzeri) → düğüm sayısı h'de ÜSTEL → tersi: yükseklik logaritmik, h ≤ 2 log n.

> *"node.height equals 1 plus max of node.left.height and node.right.height."* — Demaine, 43:33

Bu da $O(1)$ güncelleme kuralıdır → her düğümün yüksekliğini (ve dolayısıyla skew'ini) zenginleştirmeyle tutabiliriz. (Rotasyon sırasında da etkilenen düğümlerin yükseklik/size alanları güncellenmelidir.)

17.11 Rotasyonlarla Dengeyi Koruma

Insert/delete bir yaprak ekler/siler → bazı ataların skew'i ± 2 olabilir (denge bozulur). Ata yolunu **alttan yukarı** kontrol et, **en alttaki dengesiz düğümü** x bul (skew $= \pm 2$). Yaprak değişimi yüksekliği yalnızca 1 değiştirdiğinden, bozulma da tam ± 2'dir. $y = x$'in (yüksek taraftaki) çocuğu olsun; üç durum:

- **Durum 1-2 (skew(y) = +1 veya 0):** Tek rotasyon (x üzerinde) dengeyi geri getirir.
- **Durum 3 (skew(y) = −1):** Tek rotasyon işi kötüleştirir; **çift rotasyon** gerekir (önce $z = y$'nin çocuğu üzerinde, sonra x üzerinde). Ezberlenmesi gereken tek özel durum budur.

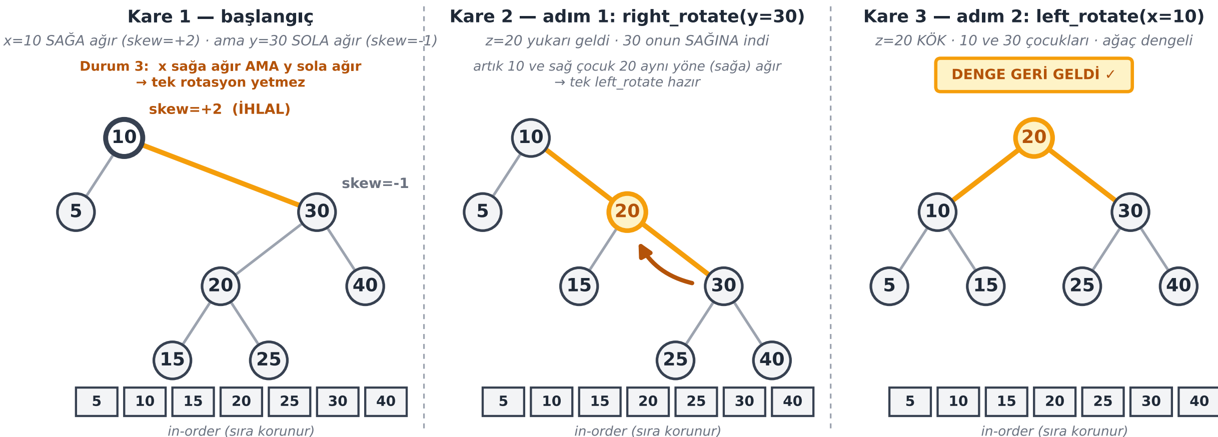
Her düzeltme $O(1)$ (sabit sayıda işaretçi); ama bir düzeltme kökün yüksekliğini değiştirebileceğinden, yukarı çıkıp parent'ı da kontrol ederiz (ve zenginleştirmeleri güncelleriz). Toplam $O(h) = O(\log n)$ sonra yükseklik dengesi geri gelir; böylece tüm işlemler $O(\log n)$ olur.

Şekil 17.7 Durum 3'ü deterministik bir senaryoda ($x = 10$, $y = 30$, $z = 20$; skew(x) $= +2$, skew(y) $= -1$) üç karede gösterir: tek rotasyonun neden yetmediği, sonra çift rotasyonun (z sonra x) dengeyi nasıl geri getirdiği — traversal $[5, 10, 15, 20, 25, 30, 40]$ üç karede de değişmez.

17.12 Bu Dersin Özeti

1. **Dizi ağacı**: `subtree_at` ile i. öğe; n_ℓ = `node.left.size` ile boyut-temelli ikili arama; $O(h)$.
2. **Alt ağaç zenginleştirme**: çocuklardan $O(1)$ hesaplanan özellik (`size`, `height`); insert/delete sonrası ata yolunu güncelle.
3. **Tutulabilir**: sum/min/max/size (aşağı-bakan); **tutulamaz**: index/depth (global).
4. **Rotasyon**: traversal sırasını koruyarak (A, X, B, Y, C) yeniden dengeleme aracı.
5. **AVL / skew** $\in \{-1, 0, +1\}$: yükseklik dengesi.
6. **Yükseklik dengesi →** $h \leq 2 \log n$: N_h Fibonacci-benzeri, üstel; h logaritmik.
7. **Dengeyi koru**: en alttaki dengesiz düğümde 1 veya 2 rotasyon; $O(\log n)$.

> **! Tek Bir Cümle**
>
> İkili ağaca `size/height` zenginleştirmesi ekleyip, AVL skew kuralını rotasyonla korursak, $h = O(\log n)$ garanti olur ve tüm sequence/set işlemleri $O(\log n)$'e iner.



Şekil 17.7: Durum 3 — tek rotasyon yetmez: çift rotasyon, right_rotate(30) → left_rotate(10) (L7 §10, İMZA figür). Deterministik build_case3_tree (motordan): skew($x = 10$) = +2, skew($y = 30$) = −1. **Kare 1 (başlangıç):** $x = 10$ kök, skew = +2 (kalın slate çerçeve = İHLAL); sağ çocuk $y = 30$, skew = −1; x SAĞA ağır ama y SOLA ağır → tek rotasyon yetmez. **Kare 2 (adım 1):** right_rotate($y = 30$) sonrası — $z = 20$ yukarı geldi, 30 onun SAĞINA indi; artık 10 ve sağ çocuk 20 aynı yöne ağır. **Kare 3 (adım 2):** left_rotate($x = 10$) sonrası — $z = 20$ KÖK (amber dolgu), 10 ve 30 çocukları, ağaç dengeli (DENGE GERİ GELDİ). HER karenin altında AYNI in-order şeridi [5, 10, 15, 20, 25, 30, 40]. Tek left_rotate(10) işi KÖTÜLEŞTİRİRDİ — ezberlenecek tek özel durum (Demaine Durum 3); motor doğrulaması: rebalance_node sonrası kök 20, check_avl_invariants True.

17.13 Kontrol Soruları

> **i** Soru 1: İndeks (index) neden bir alt ağaç özelliği değildir, ama size öyledir?
>
> **Cevap:** `size` *aşağı bakar*: bir düğümün boyutu yalnızca çocuklarının boyutlarından (`node.left.size + node.right.size + 1`) hesaplanır; başka bir yere düğüm eklemek bu alt ağacı değiştirmez. İndeks ise *globaldir*: bir düğümün indeksi, solundaki tüm düğümlere (alt ağaç dışındakiler dahil) bağlıdır; başa tek bir ekleme tüm indeksleri kaydırır. Sadece aşağı-bakan özellikler, ata yolunu $O(h)$'de güncelleyerek korunabilir.

> **i** Soru 2: Rotasyon traversal sırasını nasıl korur? right_rotate örneğiyle açıkla.
>
> **Cevap:** y'nin sol çocuğu x, alt ağaçlar A (x'in solu), B (x'in sağı), C (y'nin sağı) olsun. Rotasyon öncesi traversal: A, X, B, Y, C. `right_rotate(y)` sonrası x yukarı, y aşağı geçer; B, y'nin soluna bağlanır — ama traversal yine A, X, B, Y, C'dir. Sıra korunduğu için BST/sequence anlamı bozulmaz; yalnızca düğümlerin *derinlikleri* değişir, bu da dengeleme imkânı verir.

> **i** Soru 3: AVL ağacında h ≤ 2 log n olduğunu kabaca nasıl gösteririz?
>
> **Cevap:** En az dengeli AVL ağacında (her düğüm skew ± 1) yükseklik h için minimum düğüm $N_h = N_{h-1} + N_{h-2} + 1$ (Fibonacci benzeri). $N_{h-1} \geq N_{h-2}$ olduğundan $N_h > 2 \cdot N_{h-2} \rightarrow N_h \geq 2^{h/2}$. Düğüm sayısı h'de üstel olduğundan, h düğüm sayısında logaritmiktir: $h \leq 2 \log n$. Yani en kötü AVL ağacı bile, optimumun en çok iki katı yüksekliktedir.

> **i** Soru 4: Bir insert sonrası dengesizlik (skew = ±2) neden yalnızca ata yolunda aranır ve neden en fazla 1-2 rotasyon yeter?
>
> **Cevap:** Yükseklik bir alt ağaç özelliği olduğundan, bir yaprak eklendiğinde yalnızca o yaprağın **atalarının** yüksekliği (ve skew'i) değişebilir — diğer alt ağaçlar dokunulmaz. Bu ata yolu $O(\log n)$ uzunluğundadır. Yaprak ekleme yüksekliği yalnızca 1 değiştirir, dolayısıyla bozulma tam ± 2'dir; en alttaki dengesiz düğümde tek (Durum 1-2) ya da çift (Durum 3) rotasyon dengeyi geri getirir. Yukarı çıkarken her atayı kontrol ederiz, ama her biri sabit sayıda rotasyon ister $\rightarrow O(\log n)$.

17.14 Egzersizler

Egzersiz 1. Bir dizi ağacında `subtree_at(node, i)` algoritmasını, n_ℓ = `node.left.size` kullanarak elle bir örnek üzerinde yürüt ($i < n_\ell$, $i = n_\ell$, $i > n_\ell$ üç durumunu da göster).

Egzersiz 2. `node.size` güncelleme kuralının insert (yaprak ekleme) sonrası neden yalnızca $O(h)$ atayı etkilediğini tümevarımla göster.

Egzersiz 3. "Alt ağaçtaki maksimum anahtar" bir alt ağaç özelliği midir? Güncelleme kuralını yaz. "Alt ağaçtaki ikinci en küçük anahtar" için ne olur?

Egzersiz 4. Python'da bir AVL düğümü için `height` ve `skew` güncellemesini yaz:

```
def update(node):
    lh = node.left.height if node.left else -1
    rh = node.right.height if node.right else -1
    node.height = 1 + max(lh, rh)
    node.skew = rh - lh
```

Egzersiz 5. AVL'de Durum 3 (skew(y) = −1) neden tek rotasyonla çözülmez? Çift rotasyonun (önce z üzerinde, sonra x üzerinde) traversal sırasını koruduğunu, iki tekil rotasyona indirgenerek olduğunu açıkla.

17.15 Sonraki Ders İçin Hazırlık

Ders 12 (L8): İkili Yığınlar (Binary Heaps)

Erik Demaine ile, **öncelik kuyruğu (priority queue)** arayüzüne ve onu çözen **ikili yığına** geçiyoruz: en büyük (veya en küçük) öğeyi $O(\log n)$'de çek/ekle. AVL'nin tam gücüne gerek olmayan, diziye gömülü zarif bir dengeli-ağaç. (Not: ders akışında araya **Problem Oturumu 4** girer — kitap düzeninde bu, araya giren **Ders 11 (PS4)**'tür; İkili Yığınlar onu izleyen **Ders 12**'dir.)

Ders 12 (L8) Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle **Egzersiz 4**'ü (AVL update) çöz.
- Üç fikri (subtree_at, augmentation, rotation) ve nasıl $O(\log n)$ verdiklerini ezberden anlat.
- Ana cümleyi tekrar oku: *"AVL skew kuralını rotasyonla korursak, $h = O(\log n)$ garanti olur."*

17.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
subtree_at(node, i)	i. öğe; n_ℓ ile boyut-temelli ikili arama; $O(h)$	Böl. 3
Alt ağaç özelliği	Çocuklardan $O(1)$ hesaplanan, aşağı-bakan nicelik	Böl. 4
size güncelleme	`node.size = node.left.size + node.right.size + 1`	Böl. 4
Tutulamaz	index, depth (global; tüm ağaca bağlı)	Böl. 5
Rotasyon	Traversal'ı (A, X, B, Y, C) koruyan dengeleme	Böl. 6
skew	height(sağ) – height(sol); AVL: skew $\in \{-1, 0, +1\}$	Böl. 7
Yükseklik dengesi	N_h Fibonacci-benzeri üstel → $h \leq 2 \log n$	Böl. 8
Dengeyi koru	En alttaki dengesizde 1-2 rotasyon; $O(\log n)$	Böl. 10

17.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, “dengeyi nasıl garanti ederiz ve zenginleştirme ile ne kazanırız” sezgisini kurar — köprülerin özeti:

1. **Dengeli ağaç (AVL/red-black)** → veritabanı indeksleri ve dosya sistemleri (B-tree, B+-tree): disk-dostu denge; çoğu SQL motorunun varsayılan indeksi bir dengeli ağaçtır (hash/GIN gibi özel türler ayrı uzmanlık).
2. **Subtree augmentation (`size`)** → order-statistics tree: “rank(x)” ve “k. en küçük” sorguları $O(\log n)$.
3. **Rotasyon** → tüm kendini-dengeleyen ağaçların (red-black, splay, treap) ortak ilkel işlemi.
4. **Aşağı-bakan vs global özellik** → genel tasarım: bir niceliği verimli tutmak, onun yerel (kompozisyonel) olmasına bağlıdır.
5. **Fibonacci → log yükseklik** → analiz disiplini: bir yapının garantisi, en kötü durum boyutunu (üstel düğüm) sınırlamaktan gelir.
6. **Traversal sırası = değişmez** → her ağaç işleminin doğruluğu, korunan bir değişmezle (sıra, BST özelliği) ispatlanır.

Tek bir şey alıp gideceksen

İkili ağaç tek başına $O(h)$’dir; onu güçlü kılan iki ekleme — `size/height` **zenginleştirmesi** (i. öğe, denge bilgisi) ve **rotasyon** (sırayı bozmadan dengeleme). AVL skew kuralı bunları birleştirip $h = O(\log n)$ garanti eder; gerisi her yerde $O(\log n)$’dir.

18 Problem Oturumu 4

Ders 9-10'un uygulaması: AVL ağaçlarının dört uygulaması — çift rotasyonlu silme, öncelik kuyruğu ve kayan pencere, çoklu yapı ve cross-linking, iç içe ağaçlar ve rank sorgusu

> **Oturum bilgisi**
>
> - **Ku'nun videosu:** YouTube — Problem Session 4 (≈60 dk)
> - **OCW sayfası:** MIT 6.006 Problem Session 4
> - **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 11 (Problem Oturumu 4)
> - **Hoca:** Jason Ku
> - **Okuma süresi:** ≈26 dk

18.1 Bu Problem Oturumu Ne Hakkında?

Dördüncü problem oturumu (Jason Ku) **ikili ağaçların** (AVL) uygulamalarına odaklanır; yanında bir de **ikili yığın (binary heap)** önizlemesi yapar. Yığın henüz işlenmedi — Problem 2'de bir **kara kutu** olarak kullanılır ve onu açan ders, kitap düzeninde bu oturumu izleyen **Ders 12 (L8)**'dir. Bu hafta Demaine'in set/sequence veri yapısı tablosu tamamlandı; AVL ağacı hemen her işlemde $O(\log n)$ ile dengeli bir performans verir.

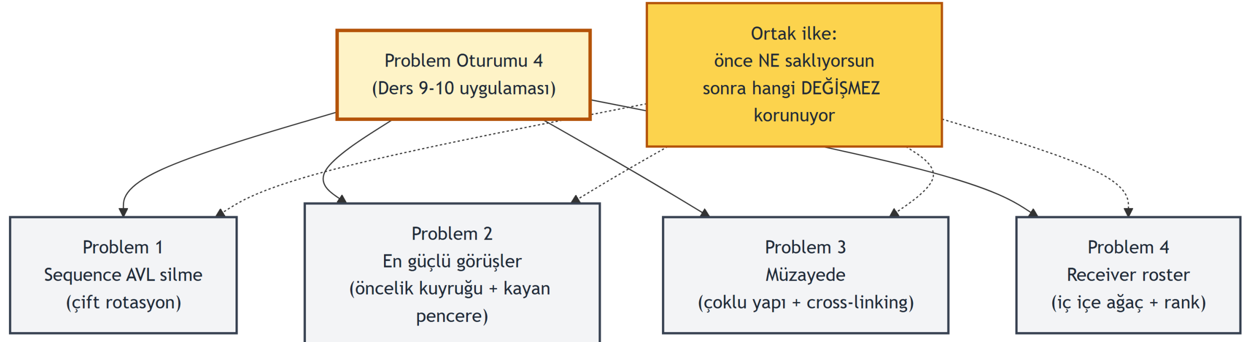
> *"Welcome to our fourth problem session. We're going to be talking about binary trees mostly, today."* — Ku, 0:21

Tüm oturuma damga vuran genel ders şudur (Şekil 39.1): her veri yapısı probleminde önce **ne sakladığını** ve hangi **değişmezleri (invariant)** koruduğunu söyle; sonra dinamik işlemler bu değişmezleri korur, sorgular onlara dayanır. Dört problem birer "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir.

18.2 Problem 1: Sequence AVL — delete_at ve Çift Rotasyon

İfade. Bir **sequence AVL ağacında** `delete_at(8)` (sıradaki 8. öğeyi sil) işlemini yürüt; sonucu yükseklik-dengeli tut.

> Yaklaşım — İki augmentation: height denge için, size sıra-indeksi için
>
> Sequence AVL ağacı **iki** alanla zenginleştirilir: `height` (rotasyonlarda dengeyi kontrol için) ve `size` (sıra-indeksiyle arama için). 8. öğeyi `subtree_at` ile bul (her düğümde sol alt ağaç boyutuna bakarak in), sil, sonra ata yolunu yukarı çıkıp dengeyi rotasyonlarla onar. İki augmentation aynı düğümde yaşar



Şekil 18.1: Problem Oturumu 4'ün kavram haritası: kök (PS4) dört probleme dallanır ve ortadaki ortak ilke düğümü her dördünü birden yönlendirir. Problem 1 sequence AVL'de çift rotasyonlu silme yapar (height ve size birlikte); Problem 2 mutlak değerce en güçlü k görüşü öncelik kuyruğuyla, sonra log n bellekle kayan pencerede bulur; Problem 3 müzayedede çoklu yapıyı cross-linking ile bağlar ve toplamı zenginleştirme olarak tutar; Problem 4 oyuncu performanslarını iç içe ağaçlarda tutup k. en yükseği rank sorgusuyla çeker. Ortadaki ilke — önce ne sakladığını, sonra hangi değişmezleri koruduğunu söyle — Ku'nun her probleme aynı kapıdan girmesini sağlar.

> ama farklı işe yarar: `size` *nereye ineceğini*, `height` *nasıl dengeleyeceğini* söyler.

> *"Sequence AVL trees… are augmented by two things… I need to store heights… and the sequence requires me to store their subtree numbers."* — Ku, 10:29

Çözüm. Silme sonrası bir düğüm dengesizleşir (skew ±2). Bu, AVL'nin **zor durumudur** (Ders 10 Durum 3): düğüm sağa ağır ama sağ çocuğu sola ağır. Tek rotasyon işi tersine bozar; **çift rotasyon** gerekir — önce sağ çocukta sağ-rotasyon, sonra düğümde sol-rotasyon.

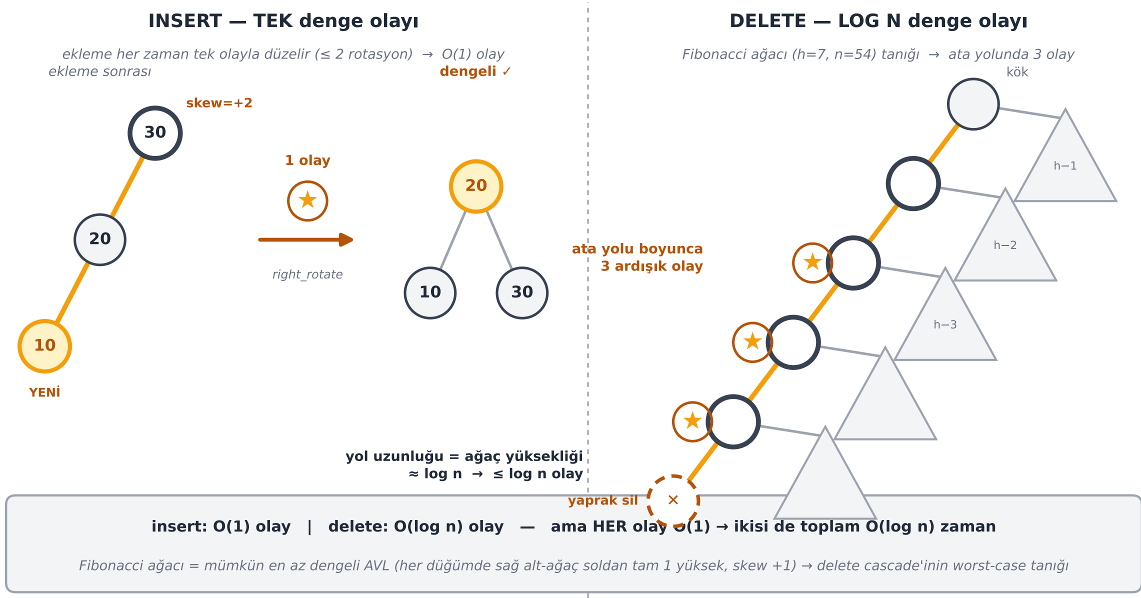
> *"I am badly skewed to the right but my right subtree is skewed to the left — I have to do a right rotation here, and then do a rotation."* — Ku, 15:53

Her rotasyonda yalnızca etkilenen 2-3 düğümün zenginleştirmesi (`height`, `size`) çocuklardan yeniden hesaplanır. Verilen ağaç bir **Fibonacci ağacıdır** — belli düğüm sayısı için en yüksek (en az dengeli) AVL; bu yüzden silme **logaritmik sayıda** rotasyon gerektirebilir. İlginç fark: insert her zaman *tek* bir denge işlemiyle (≤ 2 rotasyon içeren bir olay) düzelir; delete ata yolu boyunca log n ayrı olay isteyebilir.

> *"in a delete operation, it's possible that you have to do a logarithmic number of these rotations… An insertion operation will always re-balance after one re-balance operation."* — Ku, 24:34

Şekil 18.2 bu farkı motordan **gerçek** verilerle gösterir: 400 rastgele ekleme boyunca her ekleme en çok **1 olayla** kapanır (sol panel); buna karşılık bir Fibonacci ağacında (h = 7, n = 54) en soldaki yaprağı silmek ata yolu boyunca tam **3 ardışık olay** doğurur (sağ panel) — silmenin kademeli (cascade) onarım yapabildiğinin somut tanığı. İkisi de toplam $O(\log n)$ zamandır, çünkü her olay $O(1)$'dir; fark, *olay sayısındadır*.

Karmaşıklık. `delete_at` $O(\log n)$ (subtree_at ile bul + ata yolunda $O(\log n)$ rotasyon, her rotasyon $O(1)$).



Şekil 18.2: AVL onarımı — insert TEK olay vs delete LOG N olay (Ku 24:34) — Problem 1 İMZA. SOL panel INSERT: küçük bir AVL'ye ekleme kökte skew=+2 (slate çerçeve, İHLAL) doğurur; TEK right_rotate olayıyla (amber yıldız) dengelenir — ekleme her zaman O(1) olayla kapanır. SAĞ panel DELETE: Fibonacci ağacı (h=7, n=54; mümkün en az dengeli AVL, her düğümde sağ alt-ağaç soldan tam 1 yüksek) sembolik kademe şeması; en soldaki yaprak silinince (kesik amber daire) ata yolu boyunca motordan GERÇEK 3 ardışık seviyede skew=±2 doğar → 3 amber yıldız (3 olay). Alt şerit: insert O(1) olay, delete O(log n) olay — ama her olay O(1) olduğundan ikisi de toplam O(log n) zaman; fark olay SAYISINDADIR. Sol paneldeki 400 rastgele eklemenin hiçbiri 1 olayı aşmaz (motor tanığı).

18.3 Problem 2: En Güçlü Görüşler

İfade. "Fick Nury" (Nick Fury), n kahramanın görüşlerini (güçlü-pozitif veya güçlü-negatif) içeren *salt-okunur* bir diziden, **mutlak değerce en güçlü log n** kahramanı bulmak ister. (a) doğrusal zamanda. (b) en fazla **log n ek bellek** kısıtıyla.

> Yaklaşım — (a) build O(n) yığın; (b) sabit-boyutlu küme + kayan pencere
>
> (a) **Öncelik kuyruğu (priority queue)** — ikili yığının çözdüğü arayüz; build ve delete_max sunar. Yığın bu oturumda **kara kutu**dur (Ders 12'de açılır). (b) bellek kısıtı çıktı boyutuna (log n) bağlı bir yapıyı zorlar: en fazla log n öğelik bir set AVL ağacı + **kayan pencere** ile akışı bir kez gez, her an top-k'yı tut.

> *"It's implementing what we call a priority queue interface… delete_max."* — Ku, 29:14

Çözüm.

(a) İkili yığınla: Tüm görüşlerin *mutlak değerini* bir diziye kopyala, ikili yığın olarak **build** et — $O(n)$. Sonra **delete_max**'i log n kez çağır — en güçlü log n görüşü çeker.

> *"a binary heap does build in linear time and delete_max in log n time."* — Ku, 35:13

İkili yığının zarafeti: build $O(n)$ (set AVL'nin $O(n \log n)$'inden iyi), delete_max $O(\log n)$. Toplam $O(n + \log n \cdot \log n) = O(n + \log^2 n) = O(n)$ ($\log^2 n \ll n$).

```
def strongest_opinions_heap(opinions, k):       # (a) yığınla — O(n)
    h = MaxHeap([abs(x) for x in opinions])     # build: O(n) (heapify)
    return [h.delete_max() for _ in range(k)]   # k = log n çekiş — O(log²n)
```

(b) log n bellek + kayan pencere: En fazla log n öğelik bir **set AVL ağacı** tut (yüksekliği log log n). İlk log n öğeyle doldur; sonra kalan her öğe için: ekle, sonra **en küçüğü** sil. Değişmez: ağaç her an *o ana kadar görülen* k = log n en güçlü görüşü tutar.

> *"the invariant I'm maintaining here is that my thing always has the k largest opinions of the ones that I've processed so far."* — Ku, 48:14

```
class TopKTree:                                 # (b) kayan pencere — O(log n) bellek
    def process(self, value):                   # akıştan tek öğe
        self.root = avl_insert(self.root, value)
        self.n += 1
        if self.n > self.k:                     # taştıysa
            smallest = subtree_first(self.root).item
            self.root = avl_delete(self.root, smallest)   # en küçüğü düşür
            self.n -= 1
```

Şekil 18.3 bu izi akış $[5, 1, 8, 3, 9, 2, 7]$ ve k = 3 üzerinde motordan **gerçek** çalıştırarak gösterir: ağaç içeriği adım adım $[5] \to [1, 5] \to [1, 5, 8] \to [3, 5, 8] \to [5, 8, 9] \to [5, 8, 9] \to [7, 8, 9]$ olur. Her adımda

değişmez tutar — ağaç, o ana kadar görülenlerin tam olarak top-3'üdür (akışın 6. öğesi olan 2, mevcut minimum 5'ten küçük olduğundan girip hemen çıkar; içerik değişmez).

Karmaşıklık. (a) $O(n)$ (ikili yığın). (b) $O(n \log\log n)$ zaman (her öğe için log log n yükseklikli ağaca insert/delete), yalnızca $O(\log n)$ bellek.

18.4 Problem 3: Müzayede — Çoklu Yapı ve Cross-Linking

İfade. Bir müzayedede teklif verenler (benzersiz bidder ID + tamsayı teklif). new_bid ve update_bid $O(\log n)$; ihale kapanınca **en yüksek k teklifin toplamı** (get_revenue) $O(1)$. (k önceden sabittir.)

Yaklaşım — İki anahtar → çoklu yapı; önce ne saklanır ve hangi değişmez korunur

İki "anahtar" var: bidder ID (arama için) ve teklif (sıralama için) → tek bir ağaç yetmez, birden çok veri yapısı gerekir. Her zamanki gibi önce **ne saklandığını ve hangi değişmezin korunduğunu** netleştir: en yüksek k teklif bir kümede, geri kalanlar başka bir kümede, toplam ayrı bir sayıda. Yapıları bağlayan şey **cross-linking**'tir.

Çözüm. Dört bileşen:

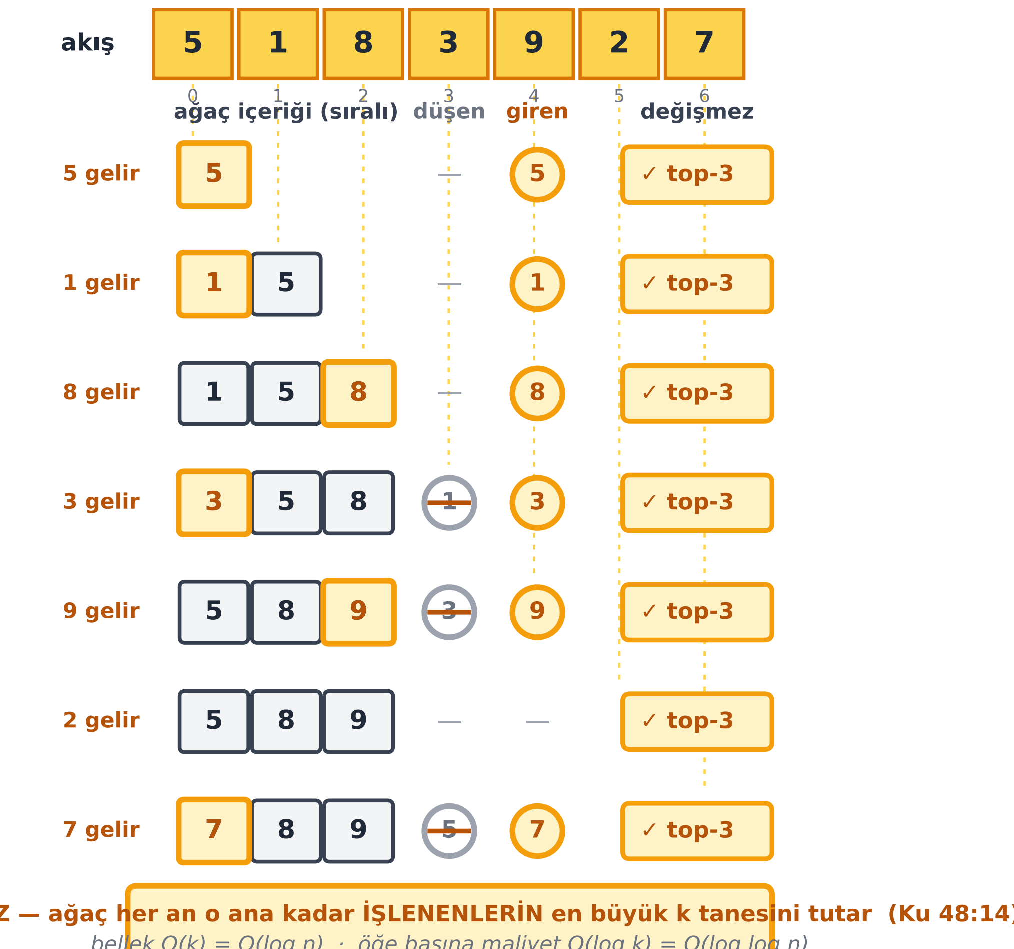
1. **Sözlük (set AVL), bidder ID anahtarlı:** bir teklifçiyi bul; her düğüm, teklifçinin diğer yapılardaki yerine bir **işaretçi** tutar (cross-linking).
2. **k-en-yüksek set AVL** (teklif anahtarlı): şu anki en yüksek k teklif.
3. **(n−k)-kalan set AVL** (teklif anahtarlı): geri kalanlar.
4. **`total` (t)**: k-en-yüksek kümesinin toplamı — bir sayı zenginleştirmesi.

new_bid: yeni teklif, k-en-yüksek'in minimumundan büyükse, o minimumu (n−k)'ye taşı ve yeniyi k-en-yüksek'e koy; değilse doğrudan (n−k)'ye. Her hareket `total`'ı $O(1)$ günceller. get_revenue sadece `total`'ı döndürür — $O(1)$.

> *"This is called cross-linking."* — Ku, 1:07:08

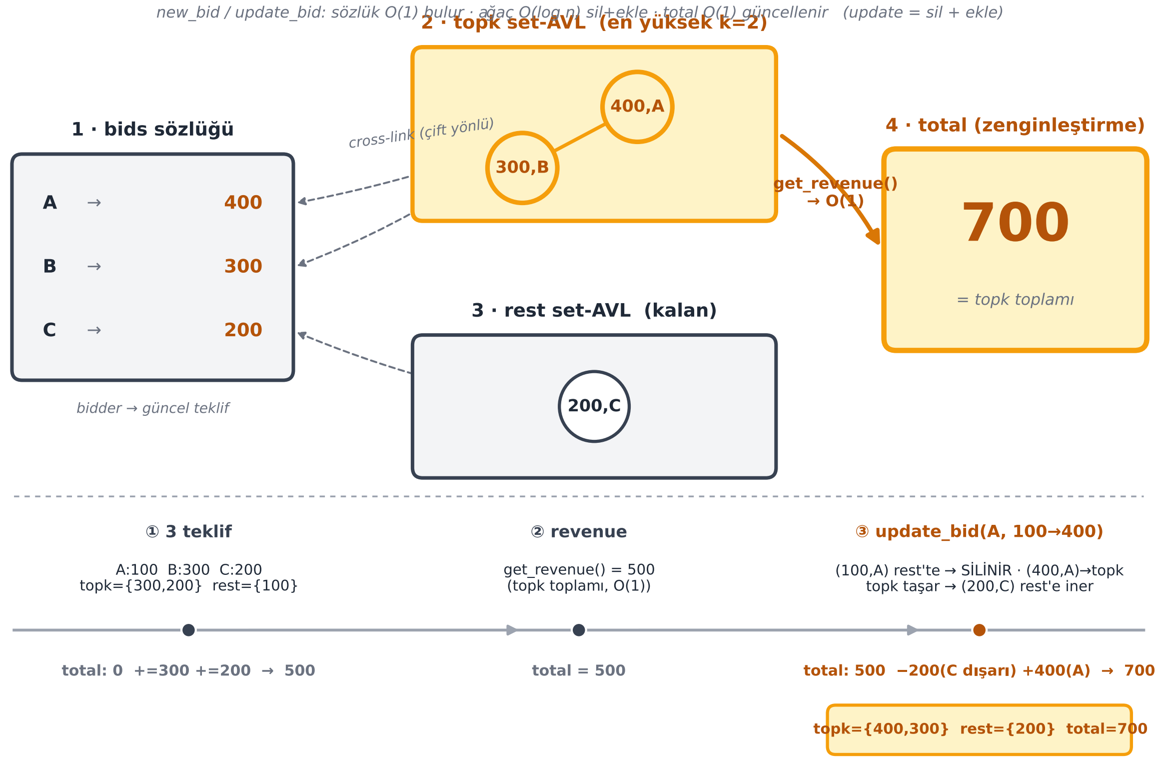
```
def new_bid(self, bidder, amount):                # O(log n)
    if bidder in self.bids:                       # cross-link: sözlük O(1) bulur
        old = (self.bids[bidder], bidder)         # eski anahtarı sil (O(log n))
        ...
    self.bids[bidder] = amount
    self._topk_insert((amount, bidder))           # yeni anahtar (O(log n))
    self._rebalance_sets()                         # topk ↔ rest sınırını koru

def get_revenue(self):
    return self.total                             # zenginleştirme → O(1)
```

Şekil 18.4 dört bileşeni ve cross-link'leri motordan **gerçek** bir senaryoyla gösterir: k = 2 ile A:100, B:300, C:200 teklifleri girilince topk = $\{300, 200\}$, rest = $\{100\}$, revenue = 500. Sonra A'nın teklifi 400'e güncellenince A'nın eski kaydı (100) rest'ten **silinir**, (400, A) topk'ye girer, taşan minimum (200, C) rest'e iner ve revenue **700** olur — her adım `total`'ı $O(1)$ günceller.



Şekil 18.3: k-en-iyi kayan set-AVL + DEĞİŞMEZ (Ku 48:14) — Problem 2b İMZA. Akış $[5, 1, 8, 3, 9, 2, 7]$, k=3 motordan GERÇEK çalıştırılır. Üstte akış şeridi; altında 7 adım satırı: her satır o adımdan sonra ağaç içeriğini (sıralı mini-set), düşen öğeyi (gri daire + üzeri çizgi) ve kalıcı giren öğeyi (amber daire) gösterir. Sağdaki amber rozet DEĞİŞMEZİ doğrular: ağaç her an o ana kadar görülenlerin top-3'üdür (hepsi ✓). Akışın 6. öğesi olan 2, mevcut minimum 5'ten küçük olduğundan girip hemen çıkar → içerik [5,8,9] değişmez. Alt amber kutu: bellek O(k)=O(log n), öğe başına maliyet O(log k)=O(log log n).

Karmaşıklık. new_bid/update_bid $O(\log n)$ (sabit sayıda AVL işlemi); get_revenue $O(1)$ (değişmez `total`).



Şekil 18.4: Müzayede: çoklu yapı + cross-linking + total zenginleştirmesi → get_revenue() O(1) — Problem 3 İMZA. Dört bileşen güncelleme SONRASI durumda (motordan GERÇEK): (1) bids sözlüğü bidder→güncel teklif (A→400, B→300, C→200); (2) topk set-AVL en yüksek k=2 {(400,A),(300,B)}; (3) rest set-AVL kalan {(200,C)}; (4) total = 700 amber kutu. İnce kesikli çift yönlü oklar CROSS-LINK'tir: sözlükteki her bidder ilgili ağaç anahtarına bağlı → güncellemede sözlük O(1) bulur, ağaç O(log n) sil+ekle, total O(1) güncellenir. Alttaki zaman çizgisi MOTOR GERÇEĞİ: 3 teklif → revenue 500; update_bid(A, 100→400) → (100,A) rest'ten silinir, (400,A) topk'ye girer, (200,C) rest'e iner → revenue 700.

> **Builder Notu — cross-linking = veritabanı ikincil indeksi / foreign key**
>
> Cross-linking, veritabanı dünyasındaki **ikincil indeks (secondary index)** ve **foreign key** ile birebir örtüşür. Bir tablo birincil anahtarıyla (bidder ID) saklanırken, sık sorgulanan başka bir alan (teklif) üzerine ayrı bir indeks kurulur; ikincil indeks **aynı kaydı farklı bir anahtarla** bulmanı sağlar — tıpkı buradaki teklif-anahtarlı topk/rest ağaçlarının sözlükteki aynı teklifçiye bir işaretçiyle bağlanması gibi. Veri tek yerde tutulur, ona iki ayrı yoldan erişilir; güncellemede her iki yapı da senkron tutulmalıdır (foreign key bütünlüğü). Bu, “aynı veriyi iki anahtarla aranabilir kılmanın” maliyeti ve gücüdür.

18.5 Problem 4: Receiver Roster — İç İçe Ağaçlar ve Rank Sorgusu

İfade. Bir futbol koçu, oyuncuların **k. en yüksek performansını** (= ortalama puan = puan/oyun, bir *rasyonel*) $O(\log n)$'de sorgulamak ister. Veriler dinamik güncellenir (oyun ekle/sil).

> Yaklaşım — Rasyonel için çapraz çarpım; rank için size augmentation
>
> Performans rasyoneldir → bölme (kayan nokta hatası) yerine **çapraz çarpım** ile karşılaştır: w_1/f_1 ile w_2/f_2'yi kıyaslarken bölmek yerine $w_1 \cdot f_2$ ile $w_2 \cdot f_1$'i karşılaştır (paydalar pozitif olduğundan işaret korunur; motorda `Fraction` bunun exact eşdeğeridir). k. en büyüğü (rank sorgusu) bulmak için ise `size` zenginleştirmesi gerekir — sequence ağacının `subtree_at`'iyle aynı mekanizma.

Çözüm. İç içe bir yapı kurulur:

1. **Set AVL (receiver ID anahtarlı):** oyuncuları bul.
2. Her oyuncuyla **iç içe bir set AVL (game ID anahtarlı):** o oyuncunun oyunları (bir oyunu $O(\log n)$'de silmek için liste değil ağaç).
3. Her oyuncuda iki sayı zenginleştirmesi: **toplam puan** ve **oyun sayısı** → performans (puan/oyun) çapraz-çarpımla karşılaştırılır.
4. **Performansa göre set AVL** (çapraz-çarpım komparatörü) + **`size` zenginleştirmesi**: bu, sequence ağacının `subtree_at`'iyle aynı **rank-find** işlemini verir — k. en büyük performansı $O(\log n)$'de bulur. (Ders 10'daki `subtree_at` ile aynı mekanizma: sondan k. öğe = in-order'da indeks n−k.)
5. Cross-linking işaretçileri (bir oyuncunun performans ağacındaki yeri).

> *"you can think of the size augmentation finding-rank as a one-sided range query."* — Ku, 1:29:19

```
def kth_best(self, k):                           # k. en yüksek performans — O(log n)
    # size-rank: in-order'da sondan k. = indeks n-k (Ders 10'daki subtree_at)
    return subtree_at(self.perf, self.perf_n - k).item

def _perf_key(self, pid):                         # rasyonel anahtar: çapraz çarpım = Fraction
    p = self.players[pid]
    return (Fraction(p["points"], len(p["games"])), pid)   # exact, bölme YOK
```

Karmaşıklık. Tüm güncellemeler ve k. en yüksek sorgusu $O(\log n)$ (iç içe aramalar log n × log n değil, çünkü oyun sayısı toplamı n).

18.6 Ne Öğrendik?

> ❗ Altı Taşınabilir Araç
>
> Bu oturum, Ders 9-10'un AVL teorisini dört somut problemde uyguladı ve altı taşınabilir araç kazandırdı:
>
> 1. **Sequence AVL = height + size:** indeksle arama (`subtree_at`) ve denge birlikte; çift rotasyon (Durum 3) ezberlenir.
> 2. **Priority queue / binary heap:** build $O(n)$ + delete_max $O(\log n)$ — set AVL'nin $O(n \log n)$

build'inden üstün (önizleme).

3. **Kayan pencere + değişmez:** sabit boyutlu (log n) yapıyla "o ana kadarki en iyi k" tutmak; az bellek.
4. **Çoklu yapı + cross-linking:** aynı veriyi farklı anahtarlarla iki/üç ağaçta tutup işaretçilerle bağlamak.
5. **Augmentation ile $O(1)$ sorgu:** bir toplamı (`total`) zenginleştirmeyle tutup sorguyu sabit zamana indirmek.
6. **`size` augmentation = rank sorgusu:** k. en büyük / "kaç tanesi büyük" (one-sided range query) — sequence `subtree_at`'in seti.

Builder Notu — kayan pencere = streaming leaderboard

Problem 2'nin (b) şıkkı, modern sistemlerin **streaming leaderboard** (akış sıralaması) deseninin tam kalbidir: sonsuz bir olay akışında (oyun skorları, satışlar, beğeniler) "şu ana kadarki en iyi k" sabit boyutlu bir yapıda tutulur ve her yeni olayda sıfırdan değil artımlı güncellenir — yeni öğe girer, eşik altına düşen en küçük çıkar. Tüm geçmişi bellekte tutmak yerine yalnızca $O(\log n)$ öğe saklamak, milyarlarca olaylık akışları sabit bellekle işlenebilir kılar; değişmez ("yapı her an o ana kadarki top-k") da sonucun doğruluğunu garanti eder.

18.7 Sonraki

Ders 12 (L8) — İkili Yığınlar (Binary Heaps)

Sırada **Ders 12 (L8): İkili Yığınlar (Binary Heaps)** var — Erik Demaine ile, bu oturumda Problem 2'de kara kutu olarak kullandığımız **öncelik kuyruğunu** açıyoruz: ikili yığın, diziye gömülü bir tam-ağaçla build $O(n)$ ve delete_max $O(\log n)$ verir. (Not: Notion kaynağı bu dersi "Ders 8 (L8)" diye anar, ama kitap düzeninde L8 = Ders 12'dir.) Bu oturumdaki augmentation ve invariant sezgileri, yığının "tahtadan-aşağı" (heapify) mantığını anlamana yarayacak.

19 İkili Yığınlar (Binary Heaps)

Öncelik kuyruğu (insert + delete_max) = set'in alt kümesi; complete binary tree diziye örtük gömülür (left 2i+1, right 2i+2, parent (j−1)//2); max-heap özelliği + heapify up/down O(log n); yerinde heapsort O(n log n) ve doğrusal build = yüksekliklerin toplamı = O(n)

Bölüm bilgisi

- **Demaine'in videosu:** YouTube — Lecture 8: Binary Heaps (≈51 dk)
- **OCW sayfası:** MIT 6.006 Lecture 8: Binary Heaps
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 12 (L8)
- **Hoca:** Erik Demaine
- **Okuma süresi:** ≈26 dk

19.1 Bu Derste Ne Var?

Ders 10 (L7) AVL ağaçlarıyla her sequence/set işlemini $O(\log n)$'e indirdi. Bugün, AVL'nin **basitleştirilmiş bir hâlini** kuruyoruz: **ikili yığın (binary heap)**. Aynı $O(\log n)$ sınırlarını verir ama iki üstünlüğü vardır — **doğrusal-zaman build** ve, asıl önemlisi, **yerinde (in-place) sıralama** (heapsort).

> *"Today, we're going to cover a different kind of tree-like data structure called a heap — a binary heap. It's going to let us solve sorting problem in a new way."* — Demaine, 0:19

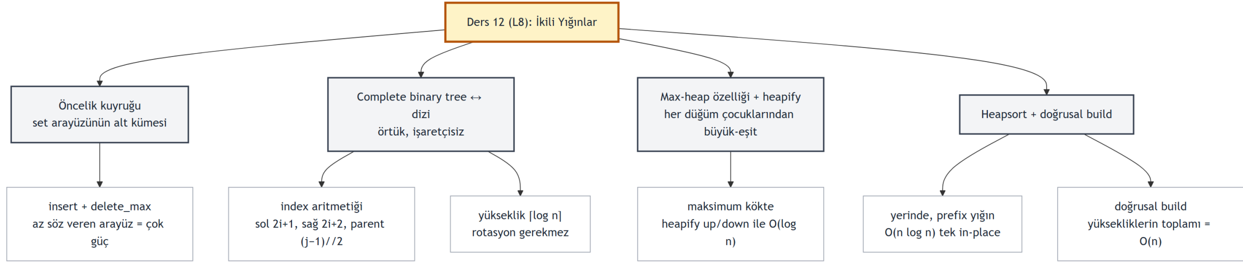
Üç temel kavram bu derste yan yana gelir:

1. **Öncelik kuyruğu (priority queue)** — set arayüzünün alt kümesi: insert + delete_max; bu kısıtlama yeni güç verir.
2. **Complete binary tree ↔ dizi** — yığın, diziye gömülü tam-ağaçtır (işaretçisiz, örtük; index aritmetiği).
3. **Max-heap özelliği + heapify** — her düğüm çocuklarından büyük-eşit; ekle/sil heapify up/down ile $O(\log n)$.

Builder Notu — Sadeleştirme = Güç

İkili yığın, AVL'nin *basitleştirilmiş* hâli — tüm set yerine yalnızca "maksimumu bul/sil" istediğimiz için, complete binary tree'yi (rotasyonsuz!) koruyabiliriz. Az söz veren bir arayüz, daha güçlü optimizasyon imkânı açar.

- **Geriye → Ders 10 (AVL):** AVL keyfi BST şekliyle rotasyon gerektirir; yığın şekli n tarafından zaten sabittir → rotasyon yok, yalnız heapify (takas).



Şekil 19.1: Ders 12'nin (L8) kavram haritası: kök = ikili yığın. Dört dal — (1) öncelik kuyruğu, set arayüzünün alt kümesidir (insert + delete_max); daha az söz veren arayüz daha güçlü optimizasyon açar. (2) complete binary tree, diziye örtük (işaretçisiz) gömülür; düğüm bağları yalnız index aritmetiğiyle bulunur (left 2i+1, right 2i+2, parent (j−1)//2) → yükseklik log n , rotasyon gerekmez. (3) max-heap özelliği (her düğüm çocuklarından büyük-eşit, maksimum kökte) heapify up/down ile korunur, insert/delete_max O(log n). (4) heapsort yerinde (prefix yığın) ve O(n log n); doğrusal build ise yüksekliklerin toplamıdır = O(n). Sonuç: öncelik kuyruğu + bedava in-place sıralama.

- **İleriye → graph algoritmaları:** Dijkstra ve Prim, öncelik kuyruğunu (yığını) doğrudan kullanır — "en yakın sıradakini al" mantığı.
- **İleriye → sistem:** router paket önceliği, işletim sistemi process scheduling, olay-tabanlı simülasyon hep öncelik kuyruğudur.
- **İleriye → heapsort:** $O(n \log n)$ *ve* in-place — ek bellek ayıramayan ortamlarda (gömülü, çekirdek) merge sort'a tercih edilir.

Tek cümle: *Öncelik kuyruğunu, diziye gömülü dengeli bir tam-ağaçla (yığın) çözersek, ekle/sil* $O(\log n)$ *olur ve yerinde* $O(n \log n)$ *sıralama (heapsort) bedava gelir.*

19.2 1. Öncelik Kuyruğu Arayüzü

Öncelik kuyruğu, öğeleri anahtarlarına (önceliklerine) göre tutar ve iki temel işlem sunar: insert (öğe ekle) ve delete_max (en yüksek öncelikli öğeyi sil ve döndür). Bu, set arayüzünün bir **alt kümesidir** — ve alt kümeler ilginçtir, çünkü daha basit/hızlı çözülebilirler.

> *"priority queue… This is a subset of the set interface."* — Demaine, 0:40

Motivasyon her yerde: router'da paket önceliği, işletim sisteminde hangi process'i çalıştıracağını seçmek, olayları zaman sırasına göre işlemek — ve bu sınıfta ileride çizge algoritmaları.

19.3 2. Set AVL ile Çözüm

Set AVL ağacı bu işlemleri zaten yapar: insert/delete_max $O(\log n)$, build $O(n \log n)$ (önce sıralama gerekir). find_max'i bir **alt ağaç zenginleştirmesiyle** (her düğümde alt ağacın maksimum anahtarı) $O(1)$'e bile indirebiliriz.

> *"set AVL is our most powerful data structure."* — Demaine, 3:33

Yani aslında “işimiz bitti”. Ama bugün **ikili yığın** öğreneceğiz: AVL kadar güçlü değil, sadece priority queue’yu çözüyor — ama daha basit, build’i bir log faktör hızlı, ve **in-place sıralama** veriyor.

19.4 3. Priority Queue Sort

Öncelik kuyruğu arayüzünü gerçekleştiren *herhangi* bir yapıdan bir sıralama algoritması doğar: tüm öğeleri ekle, sonra hepsini delete_max ile çıkar. delete_max büyükten küçüğe verdiği için, ters-sıralı çıkar; lineer zamanda ters çevir → sıralı.

> *“given any data structure that implements a priority queue interface… I can make a sorting algorithm. Insert all the items, delete all the items.”* — Demaine, 8:29

Çalışma süresi: $T_{\text{build}}(n) + n \cdot T_{\text{delete_max}}$ — veya $n \cdot (T_{\text{insert}} + T_{\text{delete_max}})$. Hangi yapıyı koyarsan, o sıralamayı alırsın.

19.5 4. Üç Sıralama: Birleştirici Çerçeve

Priority queue sort, daha önce gördüğümüz üç sıralamayı **tek çerçevede** birleştirir:

- **Set AVL** (insert/delete $O(\log n)$) → **AVL sort**, $O(n \log n)$.
- **Sırasız dizi** (insert $O(1)$, delete_max $O(n)$ — maksimumu tara, sona taşı) → **selection sort**, $O(n^2)$.
- **Sıralı dizi** (insert $O(n)$ — kaydır, delete_max $O(1)$ — son öğe) → **insertion sort**, $O(n^2)$.

> *“arrays give us selection sort… insertion sort.”* — Demaine, 11:53

Selection ve insertion sort **in-place** (sabit ek alan) ama $O(n^2)$; AVL sort ve merge sort $O(n \log n)$ ama in-place *değil*. Bugünkü hedef: $n \log n$ ***ve* in-place** — ikili yığın ile (heapsort).

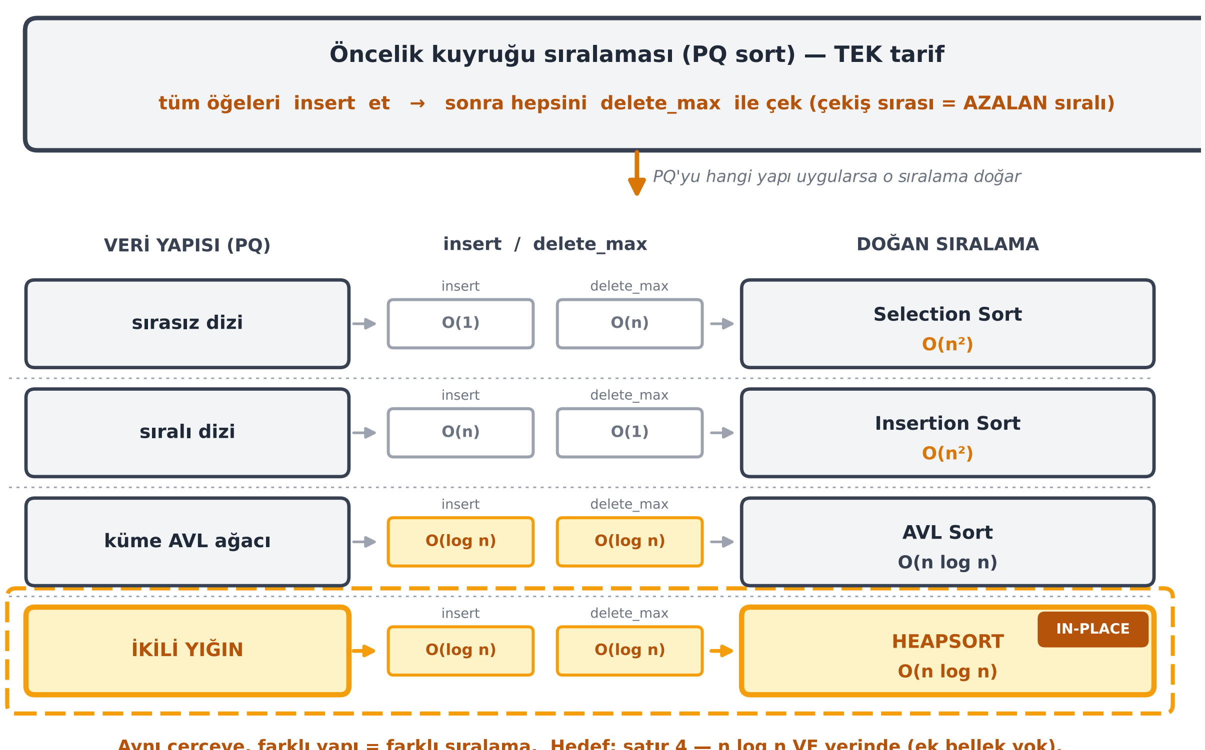
Şekil 19.2 bunu tek çerçevede toplar: aynı “hepsini insert, sonra hepsini delete_max” tarifi; hangi veri yapısını koyarsan o sıralama doğar. Hedef satır 4 — ikili yığın, iki işlemi de $O(\log n)$ dengeler ve sonuç hem $O(n \log n)$ hem yerinde.

19.6 5. Hedef: n log n + Yerinde (Complete Binary Tree)

In-place olmak için, n öğeyi tam n dizi gözünde tutmalıyız. $\log n$ performans için de bir **ağaç** gerekir. Çözüm: ağacı bir **diziye gömmek**. Hangi ağaç? **Complete binary tree (tam ikili ağaç)**: her i. derinlikte 2^i düğüm, *son* seviye hariç; son seviye **sola yaslı (left-justified)**.

> *“these two properties together is what I call a complete binary tree.”* — Demaine, 17:10

Complete binary tree her zaman dengelidir — yüksekliği tam $\lceil \log n \rceil$ (mümkün olan en iyi). Rotasyona gerek yoktur.



Şekil 19.2: Öncelik kuyruğu sıralaması — birleştirici çerçeve: PQ'yu hangi yapı uygularsa o sıralama doğar (L8 §3-4). Üst kutu TEK tarifi anlatır: tüm öğeleri insert et → sonra hepsini delete_max ile çek (çekiş sırası = azalan sıralı). Altta 4 satır yapı → sıralama: (1) sırasız dizi [insert O(1) / delete_max O(n)] → Selection Sort $O(n^2)$; (2) sıralı dizi [insert O(n) / delete_max O(1)] → Insertion Sort $O(n^2)$; (3) küme AVL ağacı [O(log n)/O(log n)] → AVL Sort O(n log n); (4) İKİLİ YIĞIN [O(log n)/O(log n)] → HEAPSORT O(n log n) + IN-PLACE rozeti (amber vurgu, hedef satır). Dizi PQ'lar bir işlemi ucuz yapar ama diğerini O(n)'e iter → $O(n^2)$; yığın iki işlemi de O(log n) dengeler → O(n log n). Motor kanıtı: Q=[]; insert [4,1,7,3,9,2,6]; drain ×7 = [9,7,6,4,3,2,1] = sorted(desc).

19.7 6. Complete Binary Tree ↔ Dizi

Tam ikili ağaç ile dizi arasında bir **birebir eşleme (bijection)** vardır: düğümleri **derinlik sırasında** (level order — A, sonra B, C, sonra D, E, F, G…) diziye yaz. n verilince tek bir ağaç şekli vardır (yukarıdan aşağı, soldan sağa dolar).

> *"between complete binary trees and arrays is a bijection."* — Demaine, 18:33

Bu yüzden işaretçi saklamaya gerek yok — sadece dizi: **örtük veri yapısı (implicit data structure)**.

> *"This is what we call an implicit data structure… no pointers, just an array of the n items."* — Demaine, 20:28

Çalışılan Örnek — index aritmetiği. İşaretçileri index hesabıyla bul. i. düğüm için:

- Sol çocuk: $2i + 1$.
- Sağ çocuk: $2i + 2$ (sol çocuğun sağ kardeşi).
- Parent (j çocuksa): $(j - 1)//2$ (tamsayı bölme).

Örnek: index 4'teki düğümün sol çocuğu $2 \cdot 4 + 1 = 9$; sağ çocuğu $2 \cdot 4 + 2 = 10$ — dizi boyutunu aşıyorsa o çocuk yoktur. Ters yön de tutarlı: parent$(9) = (9 - 1)//2 = 4$. Tüm bunlar sabit zaman; complete binary tree'nin özel yapısı sayesinde mümkün.

Şekil 19.3 bu bijeksiyonu somutlar: 11 düğümlü bir max-heap, hem ağaç hem dizi olarak; index 4 düğümü amber dolgu, çocukları 9 ve 10 amber çerçeve, formül kutusunda left$(4) = 9$, right$(4) = 10$, parent$(9) = 4$ doğrulanır.

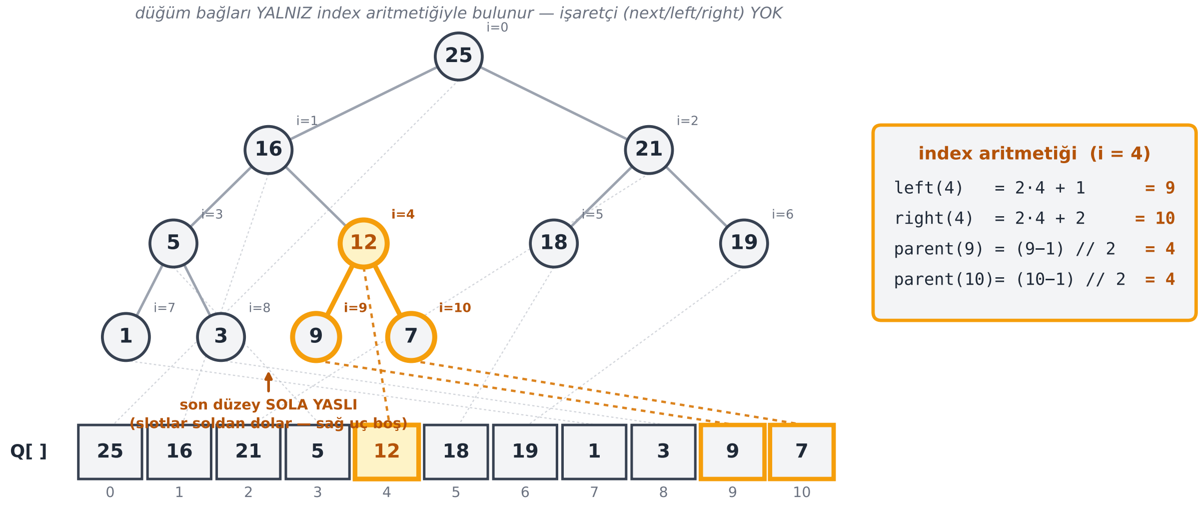
19.8 7. Max-Heap Özelliği

Son bir kural: **max-heap özelliği** — her düğüm i için $Q[i]$.key $\geq Q[j]$.key (j sol veya sağ çocuk). Yani her düğüm iki çocuğundan da büyük-eşittir; iki çocuğun birbirine göre sırası *umursanmaz* (BST'den farkı budur).

> *"the max-heap property: Q[i] is greater than or equal to Q[j] for both children."* — Demaine, 26:02

Önemli lemma: bu özellik her yerde sağlanıyorsa, her düğüm **alt ağacındaki tüm düğümlerden** büyük-eşittir (geçişlilikle: aşağı yolda her kenar anahtarı azaltmaz). Dolayısıyla **maksimum kökte** durur.

> *"every node i is greater than or equal to all nodes in its subtree."* — Demaine, 27:39



Şekil 19.3: Complete binary tree   dizi: işaretçisiz örtük yapı (L8 §5-6, İMZA figür). Üst: 11 düğümlü complete binary tree, 4 düzey, son düzey SOLA YASLI (etiketli); her düğüm dairesinde değer, yanında index rozeti (i=0..10). Alt: aynı 11 değerin dizi şeridi (her hücre altında index). Her ağaç düğümünden dizi hücresine ince bağ çizgisi (level-order bijeksiyon). index 4 düğümü amber DOLGU; çocukları index 9 ve 10 amber ÇERÇEVE; sağda formül kutusu: left(4) = 2·4+1 = 9, right(4) = 2·4+2 = 10, parent(9) = (9−1)//2 = 4, parent(10) = (10−1)//2 = 4. Düğüm bağları YALNIZ index aritmetiğiyle bulunur — işaretçi (next/left/right) YOK. Yükseklik = log n → işlemler O(log n); ağaç dolu ve sola yaslı → dengeleme rotasyonu gerekmez (BST/AVL aksine). Motor: build_heap_linear([25,16,21,5,12,18,19,1,3,9,7]) zaten max-heap; is_max_heap True; heap_left(4)=9, heap_right(4)=10, heap_parent(9)=4, heap_parent(10)=4 (Demaine 20:28).

19.9 8. insert ve max_heapify_up

insert(x): yeni öğeyi yalnızca dizinin **sonuna** ekleyebiliriz (son yaprak, index $=$ size $-$ 1). Ama bu max-heap özelliğini bozabilir → düzelt.

Çalışılan Örnek — max_heapify_up. Eklenen düğümden başla. Parent'ının anahtarı çocuktan küçükse (kural ihlali), ikisini **takas et**; sonra parent ile **özyinele** (yukarı çık). Öğe en fazla kök seviyesine kadar "kabarır". Yalnızca bu tek öğe yanlış yerde olduğundan, hareket durduğunda max-heap özelliği sağlanmıştır.

> *"max_heapify_up… it goes up one… the running time is the height of the tree, which is log n."* — Demaine, 36:06

Süre: ağaç yüksekliği = $O(\log n)$.

Şekil 19.4 bu kabarmayı somut bir örnekte gösterir: $[9, 5, 8, 3, 1]$ yığınına 10 eklenir; 10 sona (index 5) konur, parent 8'den büyük olduğu için takas edilir (index 2), sonra parent 9'dan da büyük olduğu için bir kez daha takas edilir — toplam 2 kabarma — ve 10 köke ulaşır.

insert + kabarma (heapify_up): yeni öğe sona eklenir, parent'ından büyükçe yukarı süzülür

1) 10 SONA eklenir — *index 5 (tek geçerli yer)* — parent 8 < 10 → İHLAL kabarma gerekir — *heap dizisi (10 → index 5)*

2) 8 ↔ 10 takas — *10 yukarı → index 2* — parent 9 < 10 → hâlâ İHLAL bir kez daha kabar — *8 ile yer değiştirdi*

3) 9 ↔ 10 takas — *10 KÖKTE → index 0* — max-heap ONARILDI ✓ — *9 ile yer değiştirdi*

her takas en fazla bir seviye çıkar → takas sayısı ≤ ağaç derinliği = O(log n) (Demaine 36:06)

Şekil 19.4: insert + kabarma (heapify_up): yeni öğe sona eklenir, parent'ından büyükçe yukarı süzülür (L8 §8). Üç panel yan yana; her panel: yığının complete-tree çizimi (6 düğüm) + altında o anki dizi şeridi. Motor izi heapify_up_trace([9,5,8,3,1], 10): **(1) 10 SONA eklenir** index 5 (tek geçerli yer) → dizi [9,5,8,3,1,10]; parent index 2 = 8 < 10 → İHLAL, kabarma gerekir. **(2) 8 ↔ 10 takas** → 10 yukarı index 2; dizi [9,5,10,3,1,8]; parent index 0 = 9 < 10 → hâlâ İHLAL. **(3) 9 ↔ 10 takas** → 10 KÖKTE index 0; dizi [10,5,9,3,1,8]; max-heap ONARILDI. Kabaran öğe 10 amber dolgu; İHLAL eden kenar amber + 'İHLAL' etiketi. Her takas en fazla bir seviye çıkar → takas sayısı (2) ≤ ağaç derinliği = O(log n). Motor: final kök = 10, is_max_heap(final) True (Demaine 36:06).

19.10 9. delete_max ve max_heapify_down

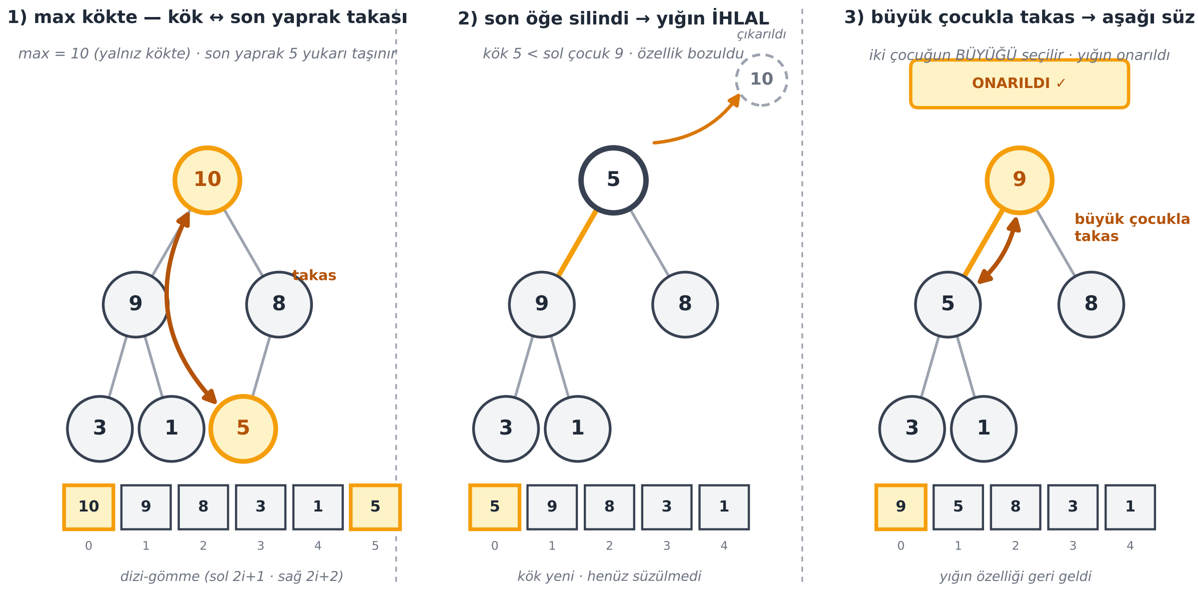
delete_max: maksimum kökte (index 0), ama dizide yalnızca **son** öğeyi verimli silebiliriz. Çözüm: kökü son yaprakla **takas et**, son öğeyi sil (delete_last). Şimdi köke küçük bir değer geldi → düzelt.

max_heapify_down: kökten başla; iki çocuktan **büyüğüyle** takas et (böylece her iki kenar da kuralı sağlar), sonra o çocukta özyinele (aşağı in). Yaprağa ulaşınca dur.

> *"Heapify down. We're going to take that item and push it down... until max-heap property is satisfied."* — Demaine, 39:16

Süre yine $O(\log n)$. (Yığında hiç rotasyon yok — yalnızca takas.)

Şekil 19.5 bu süzmeyi gösterir: $[10, 9, 8, 3, 1, 5]$ yığınında max 10 kökte; kök son yaprak 5 ile takas edilip 10 silinir; köke gelen 5, iki çocuk 9 ve 8'in büyüğü olan 9 ile takas edilir (1 süzme) ve kök 9 olur — final $[9, 5, 8, 3, 1]$, hâlâ max-heap.



Şekil 19.5: delete_max → aşağı süzme (heapify_down): kök↔son takas, sil, büyük çocukla süz (L8 §9, İMZA figür). Üç panel soldan sağa; motor izi heapify_down_trace([10,9,8,3,1,5]). **(1) max kökte** — kök 10 + son yaprak 5 amber dolgu, aralarında çift yönlü takas oku; max = 10 (yalnız kökte), son yaprak 5 yukarı taşınır; dizi-gömme sol 2i+1 · sağ 2i+2. **(2) son öğe silindi → İHLAL** — 10 çıkarıldı (soluk düğüm + dışarı ok), 5 kökte ama sol çocuk 9 > 5 → yığın özelliği bozuldu (kalın slate çerçeve); ara dizi [5,9,8,3,1]. **(3) büyük çocukla takas → aşağı süz** — 5 ↔ 9 takası (iki çocuk 9 ve 8'in BÜYÜĞÜ 9 seçilir, böylece her iki kenar da kuralı sağlar); final [9,5,8,3,1], 9 kök, ONARILDI. En fazla yükseklik kadar takas = O(log n); rotasyon YOK, yalnız takas. Motor: removed_max = 10, final kök = 9, is_max_heap(final) True (Demaine 39:16).

19.11 10. Yerinde Heapsort ve Doğrusal Build

In-place heapsort: Diziyi büyütüp küçültmek yerine, yığını dizinin bir **önekinde (prefix)** tut. insert = size'ı artır (sonraki A öğesini yığına kat); delete_max = size'ı azalt (son öğeyi düşür). Hiç yeniden boyutlandırma yok → her işlem **en kötü durum** $O(1)$ ek. Tüm öğeleri yığına al, sonra delete_max ile çıkar; en büyük öğe sona gider → dizi **yukarı sıralı** olur.

> *"that is what's normally called heapsort."* — Demaine, 48:05

Bu, $n \log n$ ***ve*** **in-place** (bu sınıftaki tek böyle algoritma).

Şekil 19.6 tek dizi içinde prefix yığının küçülüp sıralı sonekin büyümesini gösterir: $[2, 7, 1, 9, 3, 6]$ önce doğrusal build ile $[9, 7, 6, 2, 3, 1]$ yığınına dönüşür; her adım kökü (max) prefix-sonu ile takas eder, yığını 1 küçültür, kökü süzer — çıktı $[1, 2, 3, 6, 7, 9]$.

Doğrusal build: Öğeleri tek tek ekleyip yukarı-heapify yaparsan toplam derinliklerin toplamı = $n \log n$. Bunun yerine, hepsini diziye koyup **aşağıdan yukarıya heapify-down** yap. Bu, **yüksekliklerin toplamıdır** ve şaşırtıcı biçimde $O(n)$'dir (yapraklar çok ama yükseklikleri küçük; kök yüksek ama tek).

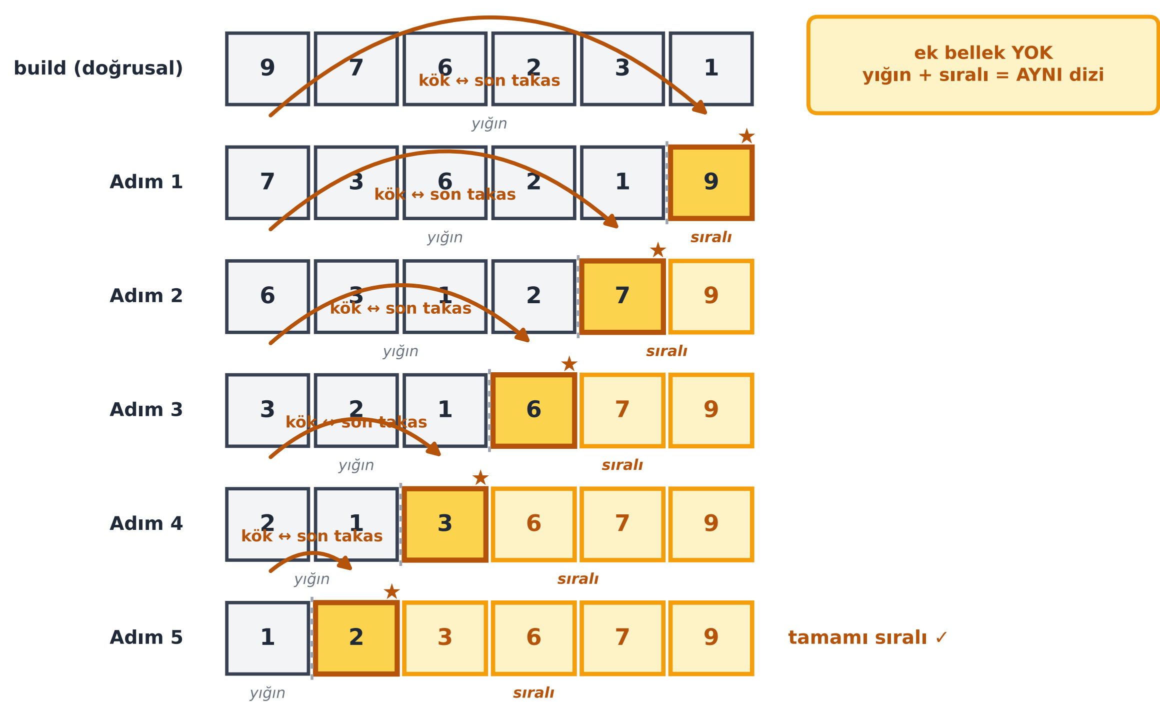
> *"heapify down from the bottom up... that's better. Because this is the sum of the heights of the nodes. And that turns out to be linear."* — Demaine, 49:24

Şekil 19.7 bu üstünlüğü ölçer: $n = 15$ complete tree'de Σ derinlik $= 34$ (tek tek insert, $\approx n \log n$) iken Σ yükseklik $= 11$ (bottom-up build, $\leq 2n = 30$); $n = 1023$'te uçurum büyür — 8194'e karşı 1013 (hâlâ ≤ 2046).

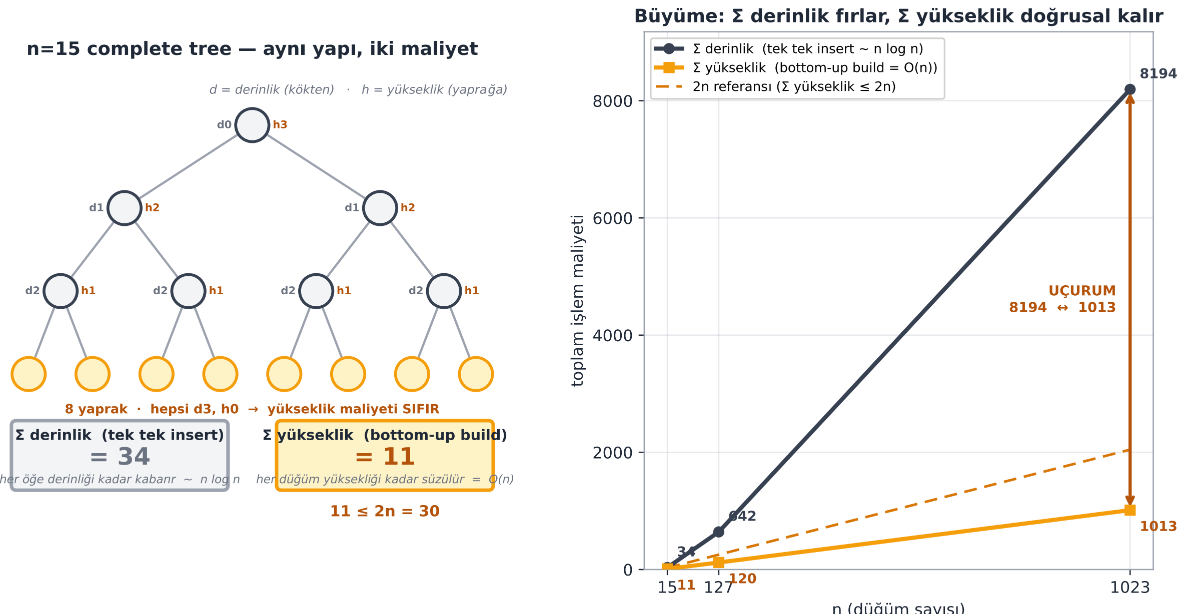
19.12 Bu Dersin Özeti

1. **Öncelik kuyruğu**: insert + delete_max; set arayüzünün alt kümesi.
2. **Priority queue sort**: ekle-hepsini + delete_max-hepsini → AVL/selection/insertion sort birleşir.
3. **Complete binary tree**: her derinlik dolu, son seviye sola yaslı; yükseklik $\lceil \log n \rceil$, dengeli.
4. **Dizi ↔ ağaç bijection**: derinlik sırası; örtük (işaretçisiz); left $2i + 1$, right $2i + 2$, parent $(j - 1)//2$.
5. **Max-heap özelliği**: $Q[i] \geq$ çocukları; maksimum kökte; düğüm $\geq$ tüm alt ağacı.
6. **insert/delete_max**: max_heapify_up / max_heapify_down (takas + özyinele); $O(\log n)$.
7. **Heapsort**: prefix yığın, in-place, $O(n \log n)$; **doğrusal build** = yüksekliklerin toplamı = $O(n)$.

> **!** Tek Bir Cümle
>
> İkili yığın, dengeli bir tam-ağacı diziye gömüp (işaretçisiz) max-heap özelliğini heapify ile korur; sonuç hem $O(\log n)$ öncelik kuyruğu hem de yerinde $O(n \log n)$ sıralama (heapsort).



Şekil 19.6: Yerinde heapsort (in-place): prefix yığın küçülür, sıralı sonek büyür (L8 §10, İMZA figür). 6 satır dizi şeridi, her satır 6 hücre = TEK dizi (in-place). Motor izi heapsort_trace([2,7,1,9,3,6]). Satır 1 = doğrusal build sonrası: tamamı yığın [9,7,6,2,3,1] (slate hücreler, SOL etiket 'yığın'). Her sonraki satır: kök ↔ prefix-sonu takas oku → prefix yığın 1 hücre küçülür, kopan max sıralı soneke geçer (yeni amber hücre yıldızlı); yığın slate (SOL etiket), sonek amber (SAĞ etiket 'sıralı'). Son satır: tamamı sıralı [1,2,3,6,7,9] (amber). Sağda kutu: ek bellek YOK, yığın + sıralı = AYNI dizi. Her adım kökü (max) yığının sonuna takas eder → sıralı soneğe katar, yığını 1 küçültür, kökü süzer; n log n VE yerinde (bu sınıfın tek böyle algoritması). Motor: build = [9,7,6,2,3,1], 6 adım, output = [1,2,3,6,7,9] (Demaine 48:05).



Şekil 19.7: Doğrusal build = yüksekliklerin toplamı, Σ yükseklik ≤ 2n = O(n) (L8 §10). Sol panel: n=15 complete tree (4 seviye); iç düğümlerin yanında d=derinlik (slate) + h=yükseklik (amber) rozetleri; yapraklar (h=0) amber dolgu; altta iki toplam kutusu: Σderinlik = 34 (tek tek insert ~ n log n, slate) vs Σyükseklik = 11 (bottom-up build = O(n), amber) + '11 ≤ 2n = 30' etiketi; 8 yaprak hepsi h0 → yükseklik maliyeti sıfır. Sağ panel: büyüme grafiği n = 15, 127, 1023; Σderinlik (slate, n=1023 → 8194 fırlar ~ n log n) vs Σyükseklik (amber, n=1023 → 1013 doğrusal kalır) + 2n referans kesikli çizgi; n=1023'te UÇURUM 8194 ↔ 1013 vurgulanır. Yapraklar ÇOK ama yükseklikleri KÜÇÜK (h=0) — çok olandan az öde; bottom-up build O(n), tek tek insert ise n log n. Motor: complete_tree_depth_height_sums(15)=(34,11), (1023)=(8194,1013) (Demaine 49:24).

19.13 Kontrol Soruları

Soru 1: İkili yığın neden rotasyona ihtiyaç duymaz, oysa AVL ağacı duyar?

Cevap: Yığın bir **complete binary tree**'dir — şekli n tarafından zaten benzersiz biçimde belirlenir (yukarıdan aşağı, soldan sağa dolu) ve her zaman dengelidir (yükseklik $\lceil \log n \rceil$). Şekil sabit olduğundan dengeleme (rotasyon) gerekmez; yalnızca anahtarların yerini (heapify up/down ile takas) düzeltiriz. AVL ise keyfi şekilli bir BST olduğundan, dengeyi korumak için rotasyon şarttır. Yığının bu lüksü, yalnızca priority queue (set'in alt kümesi) çözmesinden gelir.

Soru 2: index 4'teki düğümün çocukları ve index 9'un parent'ı nedir? Formülleri uygula.

Cevap: index 4'ün sol çocuğu $2 \cdot 4 + 1 = 9$, sağ çocuğu $2 \cdot 4 + 2 = 10$. index 9'un parent'ı $(9-1)//2 = 4$ — tutarlı (9, 4'ün sol çocuğuydu). index 10'un parent'ı $(10-1)//2 = 4$ (tamsayı bölme) — sağ çocuk da aynı parent'a döner. Bir çocuk index'i dizi boyutunu aşıyorsa o çocuk yoktur (yaprak).

Soru 3: max-heap özelliği ile BST özelliği arasındaki fark nedir?

Cevap: BST: sol alt ağaç $<$ düğüm $<$ sağ alt ağaç — *yatay* sıralama, traversal sırası anlamlı, find/range yapılır. **Max-heap**: düğüm $\geq$ her iki çocuğu — yalnızca *dikey* (ata-torun) ilişki; iki çocuğun birbirine göre sırası umursanmaz. Heap, anahtarları sıralı tutmaz, sadece "maksimum köktedir" garantisi verir; bu yüzden find(k) veya find_next yapamaz, ama find_max/delete_max'i çok ucuz yapar.

Soru 4: Doğrusal build (heapify-down, bottom-up) neden O(n)'dir, oysa tek tek insert O(n log n)'dir?

Cevap: Tek tek insert + heapify-up, her öğe için **derinlik** kadar iş yapar; toplam = derinliklerin toplamı = $O(n \log n)$ (çoğu öğe yapraktadır, derinliği $\sim \log n$). Bottom-up heapify-down ise her düğüm için **yükseklik** kadar iş yapar; toplam = yüksekliklerin toplamı. Yapraklar çoktur ama yükseklikleri 0-1; yüksek düğümler azdır. Bu ağırlıklı toplam $O(n)$'e yakınsar — "çok olandan az öde, az olandan çok öde".

19.14 Egzersizler

Egzersiz 1. Verilen bir diziyi complete binary tree olarak çiz (derinlik sırası); index aritmetiğiyle her düğümün çocuklarını/parent'ını doğrula.

Egzersiz 2. Bir max-heap'e yeni bir maksimum öğe ekle ve `max_heapify_up`'ı adım adım yürüt (köke kadar kabarma).

Egzersiz 3. Bir max-heap'ten delete_max yap: kök-son takas, delete_last, `max_heapify_down` (büyük çocukla takas). Sonucun hâlâ heap olduğunu doğrula.

Egzersiz 4. Python'da örtük yığın için index aritmetiğini ve heapify_down'ı yaz:

```
def heapify_down(Q, i, n):
    while True:
        l, r = 2*i + 1, 2*i + 2
        big = i
        if l < n and Q[l] > Q[big]: big = l
        if r < n and Q[r] > Q[big]: big = r
        if big == i: return
        Q[i], Q[big] = Q[big], Q[i]
        i = big
```

Egzersiz 5. Heapsort'un neden in-place olduğunu (prefix yığın) açıkla. Merge sort neden in-place değildir? İkisinin de $O(n \log n)$ olmasına rağmen hangi durumda heapsort tercih edilir?

19.15 Sonraki Ders İçin Hazırlık

Ders 13 (L9): Çizgeler ve BFS (Breadth-First Search)

Justin Solomon ile, veri yapılarından **çizge (graph) algoritmalarına** geçiyoruz: düğümler ve kenarlar, ve bir kaynaktan **enine arama (BFS)** ile ağırlıksız en kısa yollar. Öncelik kuyruğu, ilerideki ağırlıklı en kısa yollarda (Dijkstra) yeniden karşımıza çıkacak. (Not: ders akışında araya **Problem Oturumu** ve **Quiz 1 tekrarı** girer — kitap düzeninde Quiz 1 tekrarı araya giren **Ders 14 (Quiz 1 Review)**'tür.)

Ders 13 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle **Egzersiz 4**'ü (heapify_down) çöz.
- Üç fikri (complete tree $\leftrightarrow$ array, max-heap özelliği, heapify) ve nasıl in-place heapsort verdiklerini ezberden anlat.
- Ana cümleyi tekrar oku: *"Dengeli tam-ağacı diziye gömüp heapify ile koru."*

19.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Öncelik kuyruğu	insert + delete_max; set'in alt kümesi	Böl. 1
Priority queue sort	Ekle-hepsini + delete_max $\rightarrow$ AVL/selection/insertion	Böl. 3-4
Complete binary tree	Her derinlik dolu, son seviye sola yaslı; yükseklik $\lceil \log n \rceil$	Böl. 5
Dizi $\leftrightarrow$ ağaç (örtük)	left $2i+1$, right $2i+2$, parent $(j-1)//2$	Böl. 6
Max-heap özelliği	$Q[i] \geq$ çocukları; maksimum kökte	Böl. 7

Kavram	Tanım	Sayfada
max_heapify_up/down	Takas + özyinele; insert/delete_max $O(\log n)$	Böl. 8-9
Heapsort	Prefix yığın; in-place; $O(n \log n)$	Böl. 10
Doğrusal build	Bottom-up heapify-down = yüksekliklerin toplamı = $O(n)$	Böl. 10

19.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "az söz veren arayüz ne kadar güç açar ve örtük yapı ne kazandırır" sezgisini kurar — köprülerin özeti:

1. **Öncelik kuyruğu** → Dijkstra/Prim (graph), event-driven simülasyon, OS scheduler, router QoS — her yerde "en öncelikliyi al".
2. **Heapsort (in-place, $n \log n$)** → bellek-kısıtlı sistemler (gömülü, çekirdek); ek tampon ayıramayan ortamlar.
3. **Örtük veri yapısı** → cache-dostu tasarım: işaretçisiz dizi, bellek yerelliği yüksek (heap, segment tree).
4. **Complete tree** ↔ **array** → segment tree / Fenwick tree'nin temeli; aynı index-aritmetiği indeksleme.
5. **Alt küme = daha fazla güç** → API tasarım ilkesi: daha az söz veren arayüz, daha güçlü garanti/optimizasyon sunabilir.
6. **Yüksekliklerin toplamı = $O(n)$** → amortize/ağırlıklı analiz sezgisi: "çok olandan az, az olandan çok öde".

! Tek bir şey alıp gideceksen

İkili yığın, "her şeyi" değil yalnızca "maksimumu" istediğin için, dengeli bir tam-ağacı diziye işaretçisiz gömebilir ve rotasyonsuz korur. Bu kısıtlama iki armağan verir: doğrusal build ve — tek in-place $O(n \log n)$ sıralaman olan — heapsort.

20 Çizgeler ve Enine Arama (BFS)

Çizge G=(V,E), yönlü (sıralı) / yönsüz (sırasız); basit çizgede |E|=O(V^2); derece toplamı 2|E| → komşu döngüsü O(E); komşuluk listesi (+hash) O(1) kenar sorgusu; en kısa yol ağacı = öncül P(v), O(V) yer (optimal alt yapı); BFS seviye kümeleri L_k katman katman → δ/P/L tek geçişte, O(V+E) doğrusal

ℹ Bölüm bilgisi

- **Solomon'un videosu:** YouTube — Lecture 9: Breadth-First Search (≈53 dk)
- **OCW sayfası:** MIT 6.006 Lecture 9: Breadth-First Search
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 13 (L9)
- **Hoca:** Justin Solomon (Demaine/Ku değil)
- **Okuma süresi:** ≈25 dk

  Bu ders, kursun **çizge (graph) ünitesinin açılışıdır** — veri yapılarından çizge algoritmalarına geçiş.

20.1 Bu Derste Ne Var?

Ders 12 (L8) ikili yığınla öncelik kuyruğunu çözdü. Bugün Justin Solomon ile kursun **ikinci bölümünü** açıyoruz: **çizge algoritmaları**. Önce çizge terminolojisini kurarız (düğüm, kenar, yönlü/yönsüz), sonra bir kaynaktan **tüm düğümlere en kısa yolu** hesaplayan ilk algoritmayı veririz: **enine arama (BFS, breadth-first search)**.

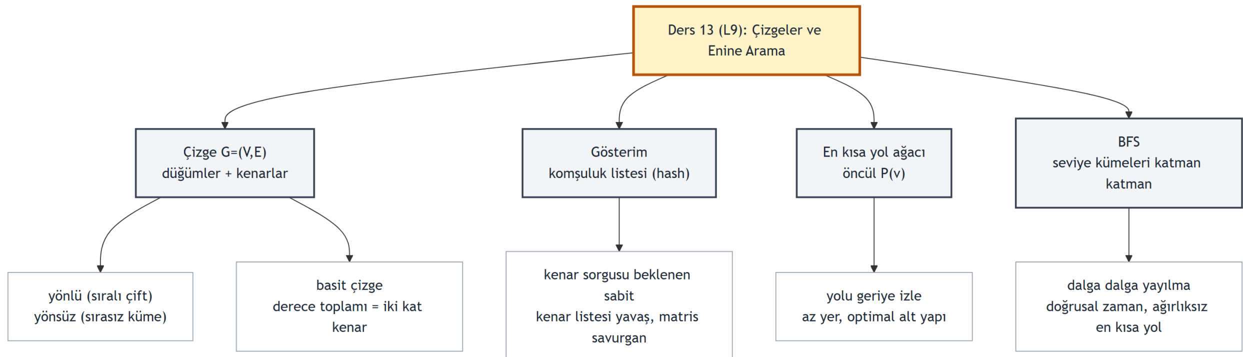
> *"we're officially starting part two of this class… our new unit which is graph theory."* — Solomon, 0:57

Üç temel kavram bu derste yan yana gelir:

1. **Çizge ve gösterimi** — $G = (V, E)$; komşuluk listesi (adjacency list) ile $O(1)$ kenar sorgusu.
2. **En kısa yol ağacı** — her düğümde sadece "öncül" $P(v)$ tutarak yolu $O(V)$ yerde sakla.
3. **BFS** — seviye kümelerini (level sets) katman katman üreterek ağırlıksız en kısa yolu $O(V + E)$'de bul.

Builder Notu — Bağlı Her Şey Bir Çizgedir

Çizge, "birbirine bağlı her şeyin" evrensel soyutlamasıdır — ağ topolojisi, sosyal graf, bağımlılık grafı, durum-geçiş uzayı. Önce nasıl saklayacağını (gösterim), sonra nasıl gezeceğini (BFS) sorarız.



Şekil 20.1: Ders 13'ün (L9) kavram haritası: kök = çizgeler ve enine arama. Dört dal — (1) çizge G=(V,E), düğüm + kenar; kenar yönlü (sıralı çift) ya da yönsüz (sırasız küme); basit çizgede |E| = O(V^2). (2) gösterim: komşuluk listesi düğümü komşularına eşler, hash ile beklenen O(1) kenar sorgusu; kenar listesi O(E), komşuluk matrisi O(V^2) yer. (3) en kısa yol ağacı: her düğümde yalnız öncül P(v) tutulur, yol geriye izlenerek kurulur, O(V) yer (optimal alt yapı). (4) BFS: seviye kümeleri L_k katman katman üretilir, δ ve öncül tek geçişte dolar, doğrusal zaman O(V+E). Sonuç: bağlı her şeyin evrensel soyutlaması + ağırlıksız en kısa yol.

- **Geriye → Ders 2-5 (veri yapıları):** çizge de bir veri yapısı tasarım problemi; "hangi sorguyu hızlandıracağım?" sorusu gösterim seçimini (kenar listesi / komşuluk listesi / matris) belirler.
- **Geriye → Ders 8 (PS3, reduction):** üç çizge problemi (erişilebilirlik ⊂ tek-çift en kısa yol ⊂ tek-kaynak en kısa yol) birbirine *indirgenir* — birini çözen diğerini çözer.
- **İleriye → Ders 16-19 (L11-L13, Dijkstra):** BFS, ağırlıksız en kısa yoldur; ağırlıklara geçince öncelik kuyruğu (yığın) devreye girer (PS5 = Ders 17 arada).
- **İleriye → her yer:** ağ yönlendirme (router, Google Maps), sosyal ağ, durum-geçiş (Rubik küpü), 3B model (üçgen ağ), seçim bölgesi — hepsi çizge.

Tek cümle: *Çizgeyi komşuluk listesiyle saklayıp BFS ile katman katman gezersek, bir kaynaktan tüm düğümlere ağırlıksız en kısa yolu $O(V+E)$'de buluruz — ve sadece "öncül"leri tutarak yolları $O(V)$ yerde saklarız.*

20.2 1. Çizge Nedir?

Bir **çizge (graph)** $G = (V, E)$ iki şeyden oluşur: bir **düğüm (vertex)** kümesi V ve bir **kenar (edge)** kümesi $E \subseteq V \times V$. Her kenar iki düğümü birbirine bağlar.

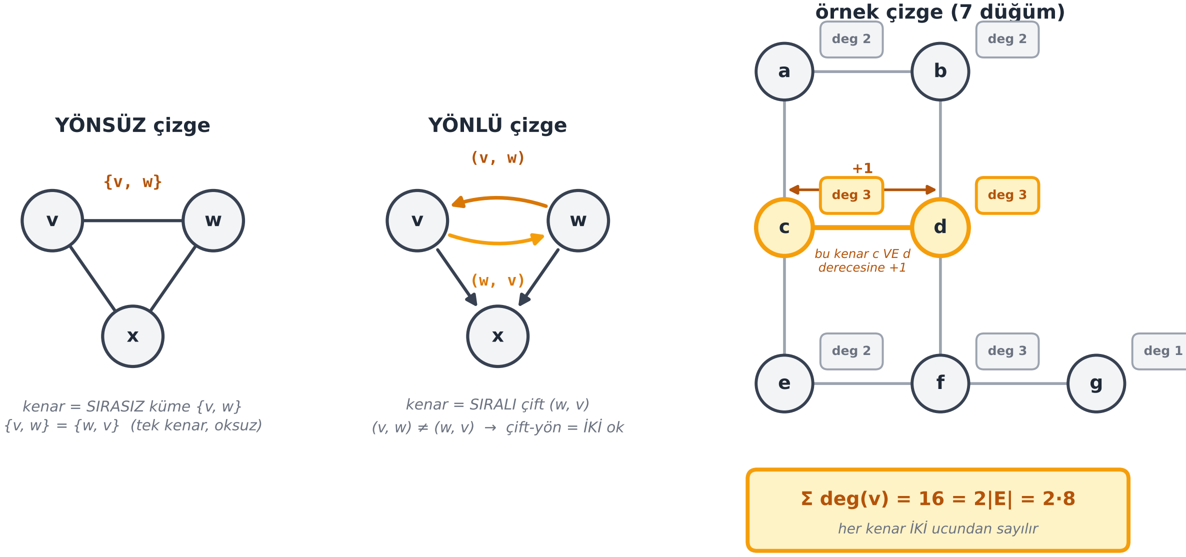
> *"a graph… is a collection of two things: a set of vertices and a set of edges."* — Solomon, 2:01

İki tür:

- **Yönlü (directed):** kenar bir *sıralı çift* (w, v); (w, v) ile (v, w) **farklıdır** (su akışına karşı yüzemezsin).
- **Yönsüz (undirected):** kenar bir *sırasız küme* $\{v, w\}$; yön yoktur, yalnızca bağlantı.

> *"in a directed graph… the edge from w to v is different than the edge from v to w."* — Solomon, 3:53

Şekil 20.2 bu iki türü yan yana koyar: solda yönsüz kenar (sırasız küme $\{v, w\}$, oksuz), ortada yönlü kenar (sıralı çift (w, v), oklu — $(v, w) \neq (w, v)$), sağda bu dersin örnek çizgesi (7 düğüm) derece rozetleriyle ve $\sum \text{deg}(v) = 16 = 2|E|$ kimliğiyle.



Şekil 20.2: Çizge anatomisi: yönsüz {v,w} vs yönlü (w,v) + derece toplamı kimliği (L9 §1/§4). SOL panel — YÖNSÜZ mini çizge: kenar = SIRASIZ küme {v, w}; düz (oksuz) çizgi, {v,w} = {w,v} aynı kenardır. ORTA panel — YÖNLÜ mini çizge (aynı düğümler): kenar = SIRALI çift (w, v); oklu kenar, (v,w) ≠ (w,v) → çift-yön İKİ ayrı ok ile (amber vurgu). SAĞ panel — bu dersin örnek çizgesi (7 düğüm sabit yerleşim): her düğüm yanında derece rozeti a:2 b:2 c:3 d:3 e:2 f:3 g:1; c-d kenarı amber işaretli, iki ucuna +1 (kenar İKİ uçtan sayılır); alt kutu Σ deg(v) = 16 = 2|E| = 2·8. Bu kimlik 'her düğümün her komşusunu gez' döngüsünü TOPLAM O(E) yapar. Motor kanıtı: degree_sums(build_example_graph()) → 16; dereceler {a:2,b:2,c:3,d:3,e:2,f:3,g:1}; make_directed'da (v,w) ile (w,v) iki AYRI kenar (Solomon 2:01 / 3:53 / 17:37).

20.3 2. Çizgeler Her Yerde

Birbirine bağlı her şey bir çizgedir: bilgisayar ağları (düğüm = makine, kenar = kablo), sosyal ağlar (düğüm = kişi, kenar = arkadaşlık), yol ağları (Google Maps en kısa yol çözer), Rubik küpü (düğüm = konfigürasyon, kenar = bir twist), 3B modeller (üçgen ağ), seçim bölgesi haritaları (komşuluk).

> *"graphs are literally everywhere in our everyday life."* — Solomon, 5:26

20.4 3. Basit Çizge ve |E| = O(V^2)

Bu derste **basit çizge** varsayarız: özdöngü (self-loop) yok, her kenar farklı (çoklu kenar yok). Sonuç: kenar sayısı $|E| = O(V^2)$ (yönlü için $|E| \leq 2\binom{V}{2}$; yönsüz için $|E| \leq \binom{V}{2}$; ikisi de en fazla V^2).

> *"the edges are big O of v squared."* — Solomon, 11:44

Bu yüzden çizge algoritmalarında **iki** boyut vardır: V ve E. Bir çizge **seyrek (sparse)** ise ($E \ll V^2$), E'ye bağlı bir algoritma, V^2'ye bağlı olandan çok daha hızlı olabilir.

20.5 4. Komşular ve Derece

- **Çıkış komşuları** $\text{Adj}^+(u)$: u'dan çıkan kenarların hedefleri. **Giriş komşuları** $\text{Adj}^-(u)$: u'ya giren kenarlar. (Yönsüzde ayrım yok, sadece $\text{Adj}(u)$.)
- **Derece (degree):** komşu kümesinin boyutu — çıkış derecesi (out-degree), giriş derecesi (in-degree).

> *"the out degree is the number of edges that point out of a vertex."* — Solomon, 17:37

Kritik kimlik (çizge algoritmalarında sürekli kullanılır): tüm düğümlerin derecelerinin toplamı $= 2|E|$ (yönsüz, her kenar iki düğüme sayılır) veya $|E|$ (yönlü, çıkış derecesi). Bu, "her düğüm için her komşusunu gez" döngüsünün $O(E)$ olduğunu söyler. Bu kimliği Şekil 20.2'nin sağ panelinde somut olarak görmüştük: örnek çizgede $\sum \deg(v) = 16 = 2 \cdot 8$.

20.6 5. Çizge Veri Yapıları

- **Kenar listesi (edge list):** tüm kenarların düz listesi. "$v \to w$ kenarı var mı?" sorgusu $O(E)$ — kötü.
- **Komşuluk listesi (adjacency list):** her düğüm u'yu, komşularına eşleyen bir küme. Komşu kümesini **hash tablosu** tutarsa, kenar sorgusu *beklenen* $O(1)$ (önce u'yu bul $O(1)$, sonra w'yi sorgula $O(1)$). $V^2 \to 1$.

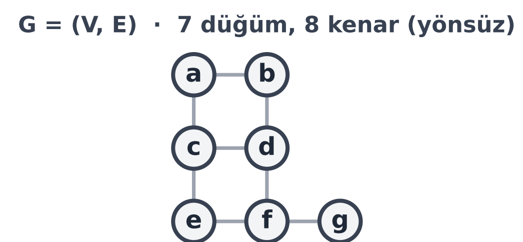
> *"an adjacency list… a set that maps a vertex u to everything adjacent to u."* — Solomon, 23:08

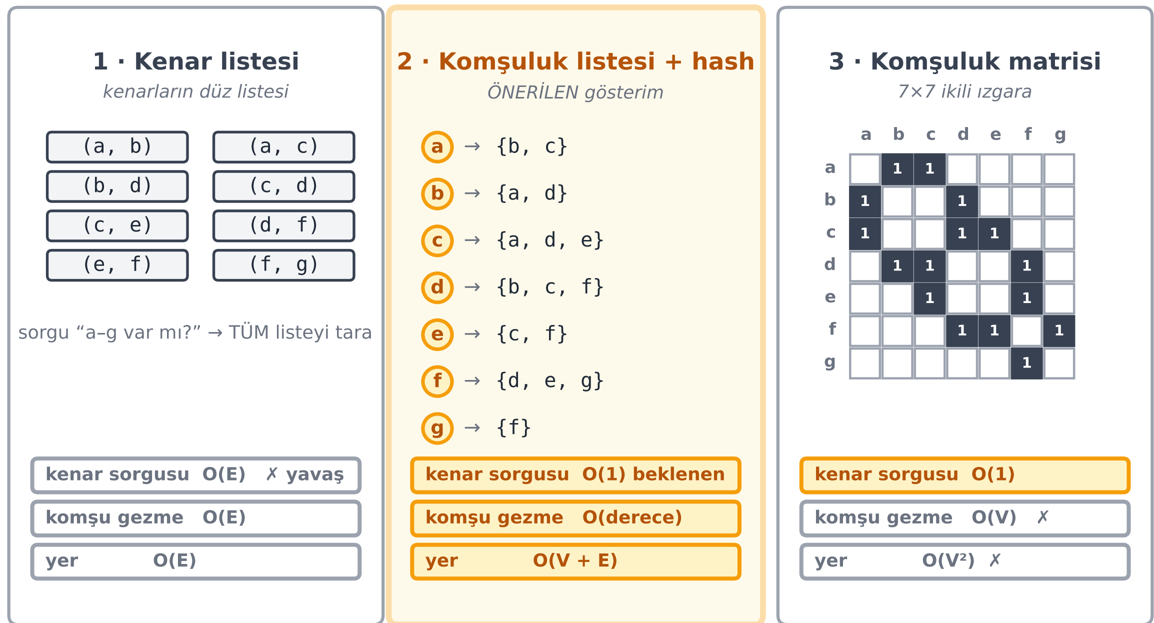
- **Komşuluk matrisi (adjacency matrix):** $V \times V$ ikili dizi; kenar sorgusu $O(1)$ ama komşuları gezmek $O(V)$ ve $O(V^2)$ yer — büyük çizgeler için savurgan.

Genel kural: komşuluk listesi (+ hash) en dengeli seçim; gösterim, hangi sorguyu sık yapacağına bağlıdır.

Şekil 20.3 aynı çizgeyi üç gösterimle yan yana koyar — kenar listesi (her sorguda yavaş), komşuluk listesi + hash (önerilen, amber çerçeve), komşuluk matrisi (sorgu hızlı ama yer ve komşu-gezme pahalı). Gösterim seçimi, hangi sorguyu hızlandıracağına bağlıdır.

Aynı çizge, üç gösterim: kenar listesi · komşuluk listesi · komşuluk matrisi





gösterim seçimi = hangi sorguyu hızlandıracağın — kenar sorgusu mu, komşu gezme mi, bellek mi? (Solomon 23:08)

Şekil 20.3: Aynı çizge, üç gösterim: kenar listesi · komşuluk listesi · komşuluk matrisi (L9 §5, İMZA figür). Üstte referans çizge (7 düğüm, 8 yönsüz kenar). Altta 3 panel: (1) KENAR LİSTESİ — 8 kenarın düz listesi; ‘a–g var mı?’ sorgusu TÜM listeyi tarar → kenar sorgusu O(E), komşu gezme O(E), yer O(E) (slate ×). (2) KOMŞULUK LİSTESİ + hash (ÖNERİLEN, amber çerçeve) — her düğüm → komşu kümesi; kenar sorgusu beklenen O(1), komşu gezme O(derece), yer O(V+E) (amber). (3) KOMŞULUK MATRİSİ — 7×7 ikili ızgara; kenar sorgusu O(1) AMA komşu gezme O(V) ve yer O(V^2) (×). Komşuluklar MOTORDAN: adj[a]={b,c}, adj[c]={a,d,e}, adj[f]={d,e,g}, adj[g]={f}; yönsüz kayıt 2|E| = 16. Gösterim seçimi = hangi sorguyu hızlandıracağın (Solomon 23:08).

20.7 6. Yollar ve En Kısa Yol

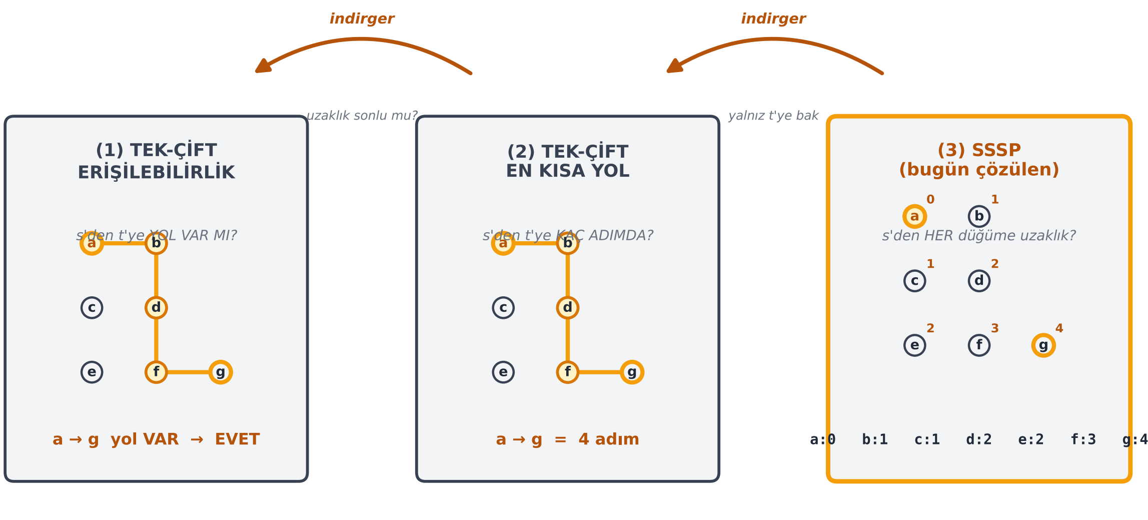
Bir **yol (path)**, ardışık çiftleri kenar olan bir düğüm dizisidir. **Uzunluğu** = kenar sayısı (= düğüm sayısı −1; sık yapılan hata: +1). **En kısa yol**, iki düğüm arasında en az kenarlı yoldur.

> *"a path is nothing more than a sequence of vertices where every pair of adjacent vertices is an edge."* — Solomon, 27:15

Üç model problem (artan zorlukta): **(1) tek-çift erişilebilirlik** (s'den t'ye yol var mı?), **(2) tek-çift en kısa yol**, **(3) tek-kaynak en kısa yol (SSSP)** — s'den *her* düğüme en kısa mesafe. Bunlar **indirgemeyle** bağlıdır: SSSP çözülürse (2) çözülür (yalnız t'ye bak), (2) çözülürse (1) çözülür (∞ ise erişilemez). Bugün (3)'ü çözeriz.

> *"a key idea in an algorithms class is this idea of reduction — that I can use one function to solve another."* — Solomon, 31:12

Şekil 20.4 üç problemi soldan sağa artan güçle dizer ve aralarındaki indirgeme oklarını gösterir: en güçlü problem (SSSP) çözülünce daha güçsüz ikisi de "bedava" çözülür.



Şekil 20.4: Üç çizge problemi + indirgeme zinciri: SSSP çözünce üçü birden çözülür (L9 §6, İMZA figür). Soldan sağa artan güç: (1) TEK-ÇİFT ERİŞİLEBİLİRLİK — s'den t'ye yol VAR mı? örnek a→g → EVET. (2) TEK-ÇİFT EN KISA YOL — kaç adımda? a→g = 4 adım. (3) SSSP (amber çerçeve, bugün çözülen) — s'den HER düğüme uzaklık: a:0 b:1 c:1 d:2 e:2 f:3 g:4. Kutular arası SOLA dönük amber 'indirger' okları: 3→2 ('yalnız t'ye bak'), 2→1 ('uzaklık sonlu mu?'). İndirgeme = bir problemi, başka bir problemin çözümünü ÇAĞIRAN bir fonksiyonla çözmek. Veri MOTORDAN: bfs(build_example_graph(),'a') → delta[g]=4, tablo a:0 b:1 c:1 d:2 e:2 f:3 g:4; reconstruct_path → a→g yolu VAR (erişilebilir) (Solomon 31:12).

20.8 7. En Kısa Yol Ağacı

Her düğüme giden tam yolu saklamak $O(V^2)$ yer ister (her yol $O(V)$, V yol). Daha iyisi: her düğümde yalnızca **öncülünü (predecessor)** $P(v)$ tut — en kısa yolda v'den bir önceki düğüm. Yolu, $P(v), P(P(v)), \ldots$ ile geriye izleyerek kurarsın. Yer: $O(V)$.

Çalışılan Örnek — optimal alt yapı. Bu, en kısa yolların güzel özelliğine dayanır: örnek çizgemizde a'dan g'ye en kısa yol $a \to b \to d \to f \to g$'dir; onun bir öneki ($a \to b \to d$) de a'dan d'ye en kısa yoldur. (Aksi halde daha kısa bir $a \rightsquigarrow d$ parçasını araya ekleyip $a \to g$ yolunu kısaltırdık — en kısa olmasıyla çelişir.) Bu **optimal alt yapı** sayesinde, tüm yol yerine tek bir "geri ok" yeterlidir. Sonuçta oluşan yapı bir **ağaçtır** (döngü olamaz — geri izleme kaynağa varır).

> *"this object is called the shortest path tree."* — Solomon, 38:31

(Uyarı: tek bir kenar eklemek *her* düğümün en kısa yolunu değiştirebilir; kaynağı değiştirmek de — o zaman her şeyi yeniden hesaplarız.)

Şekil 20.5 örnek çizge üzerinde öncül ağacını çizer: kalın geri oklar her düğümden ebeveynine ($P(v)$), amber yol $a \to g$ geriye izlenir ($g \to f \to d \to b \to a$, ters çevir $\to [a, b, d, f, g]$), sağda öncül tablosu ve "$O(V)$ yer vs üstü çizili $O(V^2)$" karşılaştırması.

20.9 8. Seviye Kümeleri ve BFS

Seviye kümesi L_k: kaynaktan tam k uzaklıktaki düğümler. $L_0 = \{\text{kaynak}\}$. BFS, bu kümeleri **katman katman** üretir.

> *"the level set… all the vertices that are distance k away from my source."* — Solomon, 42:08

Çalışılan Örnek — BFS algoritması. Başlat: $L_0 = \{s\}$, $\delta(s, s) = 0$, P boş. Sonra, önceki seviye boş olmayana dek ($i = 1, 2, \ldots$): önceki seviyedeki her u için, u'nun **henüz görülmemiş** her komşusu v'ye: v'yi L_i'ye ekle, $\delta(s, v) = i$ yap, $P(v) = u$ ata. "Henüz görülmemiş" şart — bir düğümü iki seviyeye koymayız (ilk görüldüğü uzaklık en kısadır).

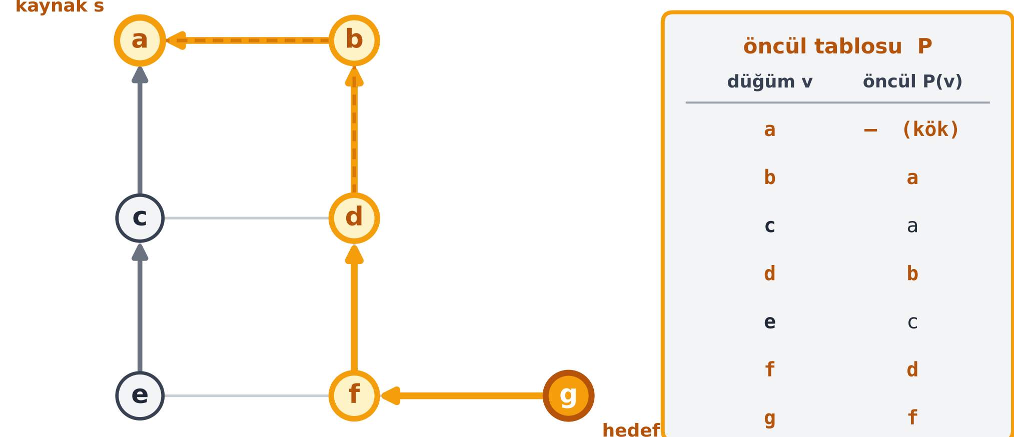
> *"Breadth-First search… an algorithm for computing all of those level sets."* — Solomon, 43:35

Mantık tümevarımsaldır: u'ya $i - 1$ adımda ulaşılıyorsa, komşusuna i adımda ulaşılır. Algoritma δ (mesafe), P (öncül) ve L (seviye) bilgilerini tek seferde doldurur.

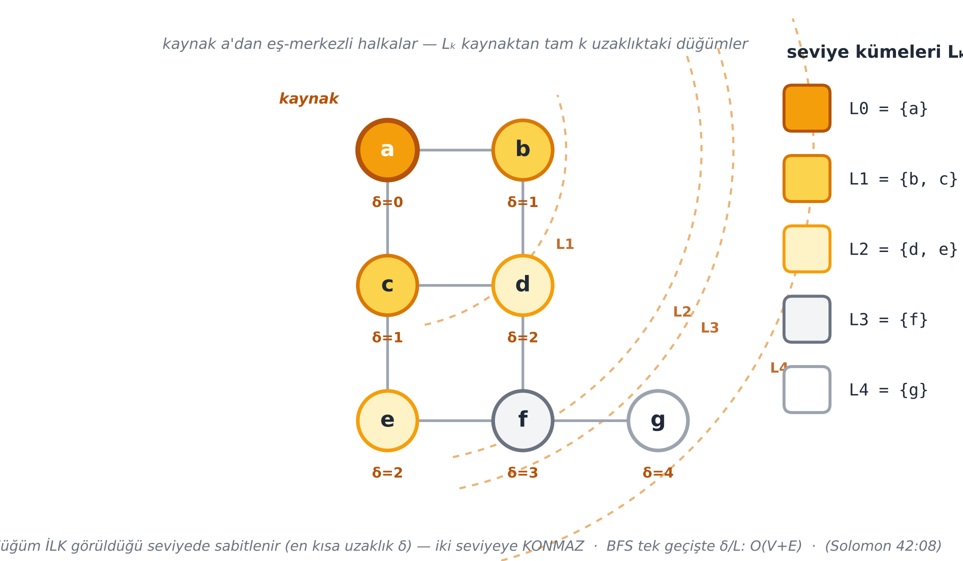
Şekil 20.6 örnek çizgede seviye kümelerini "dalga dalga" gösterir: kaynak a'dan eş-merkezli halkalar, her düğüm bulunduğu seviyenin tonuyla, altında δ rozeti.

BFS'i komşuluk listesi üzerinde Python'da yazmak kısadır (kuyruk = FIFO; "henüz görülmemiş" kontrolü δ sözlüğünde):

```
from collections import deque

def bfs(adj, s):
    delta = {s: 0}
    parent = {s: None}
```



Şekil 20.5: En kısa yol ağacı: öncül P(v) ile geriye izleme → yol [a, b, d, f, g] (L9 §7, İMZA figür). Sol/orta: örnek çizge; tüm kenarlar soluk slate; ÖNCÜL kenarları (b→a, c→a, d→b, e→c, f→d, g→f) kalın GERİ OKLARI (çocuktan ebeveyne) → bu oklar en kısa yol AĞACINI kurar. a→g yolu amber vurgu: g→f→d→b→a geriye izle, ters çevir → [a,b,d,f,g]; a→d öneki amber kesikli (optimal alt yapı). Sağ panel: öncül tablosu P (a kök, b:a, c:a, d:b, e:c, f:d, g:f); ‘yer O(V)’ vs üstü çizili ‘tüm yollar O(V^2)’. Alt kutu: OPTİMAL ALT YAPI — en kısa yolun her öneki de en kısadır → tek geri ok yeter. Veri MOTORDAN: parent = {a:None, b:a, c:a, d:b, e:c, f:d, g:f}; reconstruct_path(P,a,g) = [a,b,d,f,g] (Solomon 38:31).



Şekil 20.6: BFS = dalga dalga yayılma: seviye kümeleri L_k (kaynaktan tam k uzaklık) (L9 §8, İMZA figür). Örnek çizge (7 düğüm, 8 yönsüz kenar) kaynak a'dan eş-merkezli KESİKLİ dalga yaylarıyla — bir taşın suya düşmesiyle oluşan halkalar gibi. Her düğüm bulunduğu seviyenin tonuyla dolar: L0 a amber dolgu (kaynak), L1 b,c amber-300, L2 d,e amber-100, L3 f slate açık, L4 g beyaz+slate çerçeve. Her düğümün altında δ rozeti: a:0 b:1 c:1 d:2 e:2 f:3 g:4. Sağda seviye tablosu L0={a} L1={b,c} L2={d,e} L3={f} L4={g}. Bir düğüm İLK görüldüğü seviyede sabitlenir (en kısa uzaklık δ) — iki seviyeye KONMAZ. Veri MOTORDAN: bfs_levels(build_example_graph(),'a') = [[a],[b,c],[d,e],[f],[g]] (Solomon 42:08).

```
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v in adj[u]:
            if v not in delta:        # henuz gorulmemis
                delta[v] = delta[u] + 1
                parent[v] = u
                queue.append(v)
    return delta, parent
```

20.10 9. BFS Çalışma Süresi O(V + E)

İki bileşen: (1) V boyutlu mesafe/öncül dizilerini ayırmak $O(V)$; (2) "her düğüm için her komşusunu gez" döngüsü — her düğüm tam bir kez görülür (seviye sırasında), komşu gezmelerinin toplamı $=$ derecelerin toplamı $= O(E)$. Toplam:

$$T(\text{BFS}) = O(V + E)$$

> *"our algorithm takes big O of mod v plus mod e time."* — Solomon, 51:40

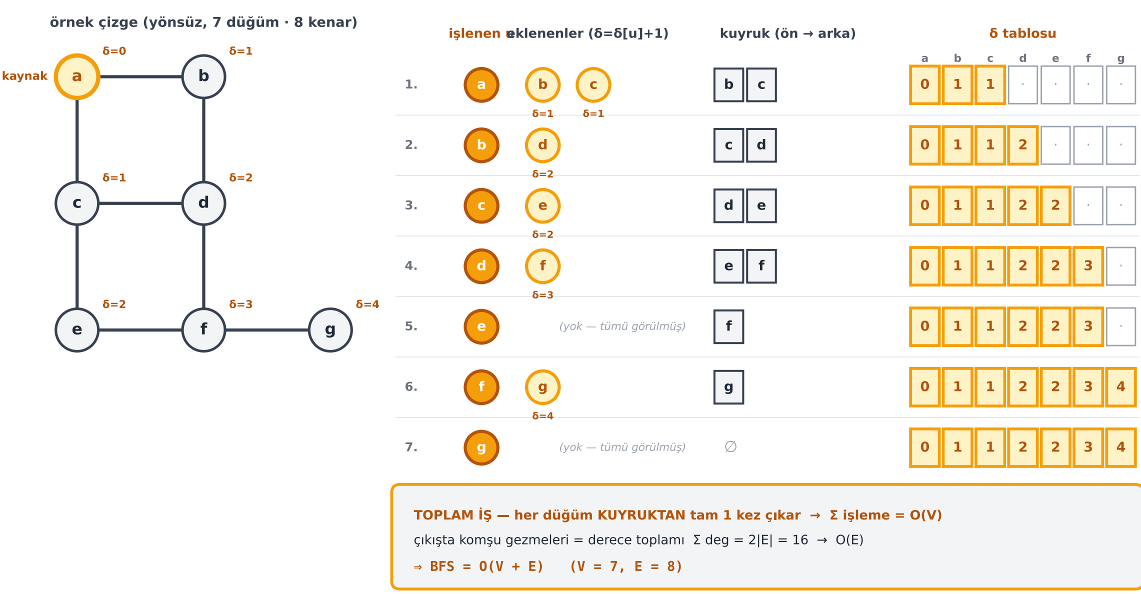
Bu sınıfta buna **doğrusal zaman** denir (çizgeyi saklamak için kullanılan yerde doğrusal). İki terim de gereklidir: kenarsız çizgede V baskın; kenar arttıkça E (V^2'ye kadar) baskın — V^2 demekten daha bilgilendirici bir ifade.

Şekil 20.7 bu yürütmeyi adım adım izler: kuyruk (FIFO) katman katman boşalır, δ tablosu tek geçişte soldan sağa dolar; toplam-iş kutusu $\sum \text{deg} = 2|E| = 16$ ile $O(V + E)$ sonucunu özetler.

20.11 Bu Dersin Özeti

1. **Çizge** $G = (V, E)$; kenar = düğüm çifti; **yönlü** (sıralı) veya **yönsüz** (sırasız).
2. **Basit çizge**: özdöngüsüz, çoklu-kenarsız; $|E| = O(V^2)$; seyrek/yoğun ayrımı.
3. **Derece toplamı** $= 2|E|$ (yönsüz) / $|E|$ (yönlü) → "her düğüm-her komşu" döngüsü $O(E)$.
4. **Gösterim**: kenar listesi ($O(E)$ sorgu), **komşuluk listesi + hash** ($O(1)$ sorgu), komşuluk matrisi ($O(1)$ sorgu, $O(V)$ komşu).
5. **Yol** = ardışık-kenarlı düğüm dizisi; uzunluk = kenar sayısı; 3 problem indirgemeyle bağlı.
6. **En kısa yol ağacı**: öncül $P(v)$ ile $O(V)$ yer; optimal alt yapı.
7. **BFS**: seviye kümeleri L_k; $\delta/P/L$'yi doldurur; $O(V + E)$ (doğrusal).

> ❗ Tek Bir Cümle
>
> BFS, kaynaktan dışa doğru "dalga dalga" yayılarak (seviye kümeleri) ağırlıksız en kısa yolu $O(V+E)$'de bulur; öncül ağacı sayesinde yolları da $O(V)$ yerde saklar.



Şekil 20.7: BFS yürütme izi: kuyruk katman katman boşalır, δ tablosu tek geçişte dolar (L9 §8-9, İMZA figür). Sol üst: örnek çizge (7 düğüm, 8 yönsüz kenar) + her düğümün δ değeri. Sağ: 7 satırlık adım izi (her satır = bir adım). Her düğüm KUYRUKTAN tam BİR kez çıkar (işlenir, amber dolu daire); çıkışta HENÜZ GÖRÜLMEMİŞ komşulara δ[v]=δ[u]+1 yazılır (amber-100 daireler) ve kuyruk SONUNA eklenir. Adımlar: 1.a→ekle b,c | 2.b→ekle d | 3.c→ekle e | 4.d→ekle f | 5.e→(yok, tümü görülmüş) | 6.f→ekle g | 7.g→(yok). Sağ alt kutu: her düğüm 1 kez işlenir → O(V); komşu gezmeleri = derece toplamı Σ deg = 2|E| = 16 → O(E); ⇒ BFS = O(V+E) (V=7, E=8). Doğrusal zaman = çizgeyi saklama YERİNDE doğrusal. Veri MOTORDAN: bfs_trace 7 adım, adım1 u=a added=[b,c], son delta a:0 b:1 c:1 d:2 e:2 f:3 g:4 (Solomon 51:40).

20.12 Kontrol Soruları

i Soru 1: Aynı çizge için komşuluk listesi mi yoksa komşuluk matrisi mi tercih edilir? Neye bağlı?

Cevap: Bağlıdır. **Komşuluk matrisi** kenar sorgusunu ($v \to w$ var mı?) $O(1)$ yapar ama $O(V^2)$ yer ister ve bir düğümün komşularını gezmek $O(V)$'dir (tüm satırı taramak gerekir). **Komşuluk listesi (+ hash)** hem kenar sorgusunu beklenen $O(1)$ hem de komşu gezmeyi $O(\text{derece})$ yapar ve yalnızca $O(V+E)$ yer kullanır. Seyrek çizgelerde ($E \ll V^2$) — ki çoğu gerçek çizge seyrektir — komşuluk listesi belirgin biçimde üstündür.

i Soru 2: En kısa yol ağacı neden tüm yolları saklamaktan (O(V^2)) daha az yer (O(V)) kullanır?

Cevap: Her düğüm için tam yolu (V'ye kadar düğüm) saklarsak $O(V^2)$ olur. Ama optimal alt yapı sayesinde, her düğümde yalnızca **öncülü** $P(v)$ (tek düğüm) tutmak yeterlidir — yolu $P(v), P(P(v)), \ldots$ ile geriye izleyerek yeniden kurarız. Her düğüm bir öncül tuttuğundan toplam $O(V)$ yerdir; yine de her yolu tam olarak verir.

i Soru 3: BFS'te bir düğüm neden yalnızca "henüz görülmemişse" eklenir? Görülmüşse ne olurdu?

Cevap: BFS, düğümleri kaynaktan *artan uzaklık* sırasında işler. Bir düğüm ilk kez L_i'de görüldüğünde, ona en kısa mesafe tam i'dir (daha erken bir seviyede görülmediğine göre). Onu daha sonraki bir L_j'ye ($j > i$) tekrar eklersek, yanlış (daha uzun) mesafe atamış oluruz. "Henüz görülmemiş" kontrolü, her düğümün *ilk* (= en kısa) uzaklığını sabitler ve algoritmanın doğruluğunu sağlar.

i Soru 4: BFS neden O(V + E)'dir, neden tek bir terim (örneğin O(V^2)) değil?

Cevap: İki ayrı maliyet vardır: V boyutlu mesafe/öncül dizilerini ayırmak $O(V)$; ve "her düğüm için her komşusunu gez" döngüsü, derecelerin toplamı $= O(E)$. Her düğüm tam bir kez işlendiğinden çift sayım yoktur. $O(V+E)$ yazmak $O(V^2)$'den daha bilgilendiricidir: kenarsız çizgede V baskın, yoğun çizgede E ($\leq V^2$) baskın olur. Tek terim, çizgenin seyrekliğini gizlerdi.

20.13 Egzersizler

Egzersiz 1. Verilen bir yönlü çizge için Adj$^+$ ve Adj$^-$ kümelerini, her düğümün in/out derecesini yaz; derecelerin toplamının $|E|$ (çıkış) olduğunu doğrula.

Egzersiz 2. Bir çizgeyi hem komşuluk listesi hem komşuluk matrisi olarak temsil et; "$v \to w$ var mı?" ve "u'nun komşuları" sorgularının her birinde maliyetini karşılaştır.

Egzersiz 3. Optimal alt yapıyı kanıtla: $a \to \ldots \to v$ en kısa yol ise, onun her öneki de bir en kısa yoldur. (İpucu: daha kısa bir önek olsaydı ana yolu kısaltırdın.)

Egzersiz 4. Python'da komşuluk listesi üzerinde BFS yaz (δ ve P döndür):

```python
from collections import deque

def bfs(adj, s):
    delta = {s: 0}
    parent = {s: None}
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v in adj[u]:
            if v not in delta:        # henuz gorulmemis
                delta[v] = delta[u] + 1
                parent[v] = u
                queue.append(v)
    return delta, parent
```

Egzersiz 5. BFS'in ürettiği `parent` (öncül) sözlüğünden, s'den belirli bir t'ye en kısa yolu (düğüm dizisi) yeniden kuran bir fonksiyon yaz. Süresi nedir?

20.14 Sonraki Ders İçin Hazırlık

Ders 15 (L10): Derinlemesine Arama (DFS) ve Topolojik Sıralama

Justin Solomon ile, BFS'in kardeşi **DFS (depth-first search)**'e geçiyoruz: çizgeyi "olabildiğince derine in, sonra geri çekil" mantığıyla gez. DFS, **topolojik sıralama** (bağımlılık sıralaması) ve **çevrim tespiti** için kullanılır. (Not: kitap düzeninde önce **Ders 14 (Quiz 1 Gözden Geçirme, Jason Ku)** araya girer; DFS bunun ardından gelir.)

Ders 15 Öncesi Yapılacak

- Bu dersin egzersizlerini, özellikle **Egzersiz 4**'ü (BFS) çöz.
- Üç çizge problemini (erişilebilirlik, tek-çift, tek-kaynak) ve indirgemeleri ezberden anlat.
- Ana cümleyi tekrar oku: *"BFS dalga dalga yayılarak ağırlıksız en kısa yolu $O(V+E)$'de bulur."*

20.15 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Çizge $G=(V,E)$	Düğümler + kenarlar; yönlü (sıralı) / yönsüz (sırasız)	Böl. 1
Basit çizge	Özdöngüsüz, çoklu-kenarsız; $\lvert E\rvert = O(V^2)$	Böl. 3
Derece toplamı	$2\lvert E\rvert$ (yönsüz) / $\lvert E\rvert$ (yönlü) → komşu döngüsü $O(E)$	Böl. 4

Kavram	Tanım	Sayfada
Komşuluk listesi	Düğüm → komşular (hash); kenar sorgusu $O(1)$	Böl. 5
Yol / en kısa yol	Ardışık-kenarlı dizi; uzunluk = kenar sayısı	Böl. 6
En kısa yol ağacı	Öncül $P(v)$; $O(V)$ yer; optimal alt yapı	Böl. 7
Seviye kümesi L_k	Kaynaktan k uzaklıktaki düğümler	Böl. 8
BFS	Katman katman; $\delta/P/L$ doldurur; $O(V+E)$	Böl. 8-9

20.16 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "bağlı her şeyin evrensel soyutlaması (çizge)" ve "katman katman keşif (BFS)" sezgisini kurar — köprülerin özeti:

1. **Çizge gösterimi** → her sistem tasarımı: sosyal graf, bağımlılık grafı, ağ topolojisi — komşuluk listesi varsayılan.
2. **BFS** → en kısa yol (ağırlıksız), web crawler (katman katman), ağ yayını, "kaç adım uzakta" (LinkedIn derece).
3. **Reduction** → OMSCS CS 6515: problemleri birbirine indirgemek, NP-tamlık dahil her yerde temel teknik.
4. **Optimal alt yapı** → dinamik programlama (DP) köprüsü: en kısa yol, DP'nin habercisidir (alt-problem çözümleri birleşir).
5. $O(V+E)$ **doğrusal** → seyreklik bilinci: gerçek çizgeler seyrektir; E-bağlı algoritma V^2-bağlıdan çok hızlıdır.
6. **Öncül ağacı** → yol yeniden kurma: routing tablosu, git geçmişi, bağımlılık çözümü — hep "geri izleme".

! Tek bir şey alıp gideceksen

Çizge, "bağlı her şeyin" evrensel soyutlamasıdır; onu komşuluk listesiyle saklayıp BFS ile kaynaktan dışa dalga dalga gezersek, ağırlıksız en kısa yolu $O(V+E)$'de buluruz. Ve tüm yolları saklamak yerine sadece "öncül"leri tutarak, $O(V)$ yerde her yolu yeniden kurabiliriz.

21 Quiz 1 Gözden Geçirme

Ders 1-12 toplu tekrar — büyük dört (çöz/kanıtla/verimli/anlat), iki çözüm yolu (sıfırdan vs İNDİRGE), üç problem tipi (white-box/black-box/modification), tuzaklar (rasyonel-radix-augmentation) ve iki gerçek sınav problemi (Criminal Seafood + Rainy Research)

ℹ Oturum bilgisi

- **Ku'nun videosu:** YouTube — Quiz 1 Review (≈85 dk — normal dersten uzun!)
- **OCW sayfası:** MIT 6.006 Quiz 1 Review
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 14 (Quiz 1 Review)
- **Hoca:** Jason Ku
- **Okuma süresi:** ≈28 dk

  Bu **normal bir ders değil** — Quiz 1 öncesi **toplu tekrar** oturumudur. Kursun ilk yarısını (Ders 1-12) tek çatı altında toparlar ve sınavda nasıl düşünüleceğini öğretir.

21.1 Bu Quiz Review Ne Hakkında?

Bu, Jason Ku ile **Quiz 1 öncesi toplu tekrar** oturumudur. Kursun **ilk yarısını** (Ders 1-12: hesaplama modeli, asimptotikler, özyineleme/master teoremi, sıralama, hashing, ikili ağaçlar/AVL, ikili yığın) tek çatı altında toparlar ve **sınavda nasıl düşünüleceğini** öğretir. İçerik üç eksende ilerler: (1) quiz neyi ölçer, (2) konu tekrarı (veri yapıları + sıralama bloğu), (3) iki gerçek sınav problemi çözümü.

> *"What are we trying to test you in this class? … Algorithms. … Also data structures."* — Ku, 0:40

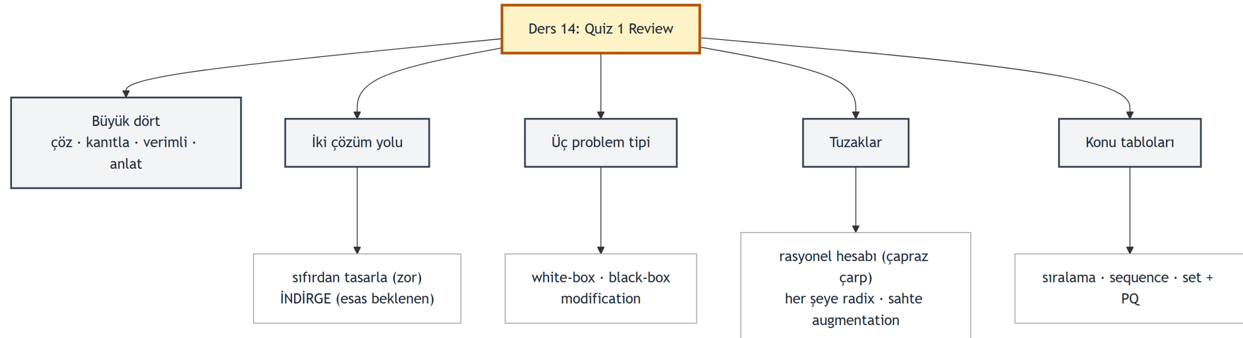
Ku'nun **"büyük dört"** hedefi (kursun ve quiz'in özü): bir hesaplama problemini (1) **çöz**, (2) **doğru** olduğunu kanıtla, (3) **verimli** olduğunu göster, (4) bunu başkalarına **anlat**.

> *"part of this class is about communication. If your thing is correct but we can't tell what you're saying, then it's not correct."* — Ku, 33:24

💡 Builder Notu — Bu = İlk-Yarı Midterm

Quiz 1, kursun ilk-yarı sınavıdır: veri yapıları + sıralama bloğunu kapsar. Çizge ve dinamik programlama, Quiz 2-3'e kalır.

- **Bu = midterm.** OMSCS CS 6515 (Graduate Algorithms) ekseninde Quiz 1, "veri yapısı + sıralama refleksi" yani lisansüstü dersin giriş varsayımıdır.



Şekil 21.1: Ders 14'ün (Quiz 1 Review) kavram haritası: kök = Quiz 1 Review. Beş dal — (1) büyük dört: çöz / kanıtla / verimli yap / anlat (iletişim de puana dahil). (2) iki çözüm yolu: sıfırdan tasarla (zor, nadiren istenir) vs bilinen kara kutuya İNDİRGE (esas beklenen). (3) üç problem tipi: white-box (içyapı) / black-box (reduction) / modification (augmentation). (4) tuzaklar: rasyonel hesabı (çapraz çarp), her şeye radix, subtree-property olmayan augmentation. (5) konu tabloları: sıralama (heap/radix özel), sequence (doubly-linked O(1) uç), set (augmented AVL ⊃ sorted array). Sonuç: Quiz 1 = icat değil SEÇİM sınavı.

- **Reduction = mühendisliğin kalbi.** "Bilinen bir kara kutuya indirge" disiplini, hem graduate algoritma dersinin hem de gerçek sistem tasarımının temel refleksidir.
- **İleriye → graduate algorithms:** burada öğrenilen "önce arayüz (correctness), sonra implementasyon (efficiency)" ayrımı, DP ve NP-tamlıkta birebir tekrar eder.

Tek cümle: *Quiz 1, "yeni algoritma icat et" demez; "doğru kara kutuyu seç, neye indirgediğini ve neyi sakladığını net söyle, sonra verimliliği optimize et" der.*

21.2 1. Büyük Dört ve İki Çözüm Yolu

Sınav, algoritma **icat etmeni** beklemez. Bir hesaplama problemini çözmenin iki yolu vardır:

1. **Sıfırdan tasarla** (zor yol): brute force veya böl-yönet gibi bir paradigmaya başvur. Bu, 6.046'nın işidir; 6.006'da nadiren istenir.
2. **Bilinen bir şeye indirge** (kolay yol, esas beklenen): derste öğretilen bir algoritmaya/arayüze **reduction**.

> *"reduce to a known thing — an algorithm that we've taught you."* — Ku, 5:25

Esas oyun: sana öğretilen sıralama ve sequence/set arayüzlerini **kara kutu** olarak kullanmak; mühendis olarak "ne zaman hangisini" seçeceğini söylemek. "Büyük dört" hedef bunun çerçevesidir: çöz, doğru olduğunu kanıtla, verimliliğini göster, **anlat** — son adım (iletişim) de puana dahildir.

21.3 2. Üç Problem Tipi: White-box / Black-box / Modification

Ku, quiz problemlerini üç sınıfa ayırır:

> *"there's three different types of problems."* — Ku, 7:02

- **(a) İçyapı / white-box:** veri yapısının içini bilmen gerekir — bir AVL ağacında rotasyon nasıl yapılır, bir max-yığında en üst k öğe nerede. Kara kutu **değil**; içine bakman şart.
- **(b) Reduction / black-box:** sadece API'yi bilmen yeter; çekirdek malzemeyi *uygula*. "Kütüphane import etmek" gibi — içine bakmazsın, sözleşmesine güvenirsin.
- **(c) Modification / augmentation:** hem API'yi hem içyapıyı bilmen gerekir — bir AVL'ye yeni bir özellik (augmentation) eklemek, dinamik diziyi ortadan büyütmek, böl-yönet'i uyarlamak. En zoru.

> *"Use as a black box… you import a library into your code… It's opaque to me."* — Ku, 6:17

Bir problemi bu üç tipten birine yerleştirebilmek, "ne kullanmalıyım?" sorusunu hızlandırır.

Şekil 21.2 üç tipi yan yana koyar: kapağı açık kutu (white-box, içyapı görünür), kapalı/opak kutu + API fişi (black-box, amber vurgu — quiz'in esas beklediği), yarı açık kutu + anahtar (modification, hem API hem içyapı).

21.4 3. Reduction Disiplini: Önce Arayüz, Sonra Verimlilik

Kilit teknik: bir algoritma/veri yapısı yerine bir **probleme veya arayüze** indirge.

> *"a lot of times it's useful to reduce it to a problem or an interface rather than an algorithm or a data structure."* — Ku, 9:43

Neden? "Sıralamaya indirgedim" dersen **doğruluğu** hemen savunursun (sıralama kara kutusu doğru); *hangi* sıralama algoritması olduğu yalnızca **verimliliği** etkiler. Böylece correctness ile efficiency'yi ayırırsın (decouple).

> *"approach these things by solving these problems just in terms of the interfaces first."* — Ku, 28:14

Veri yapıları probleminde altın kural: **ne sakladığını ve neye anahtarlandığını (keyed on)** söyle, sonra **değişmezleri (invariants)** belirt. Doğruluk kanıtı, "işlemden önce değişmezler geçerliyse, işlemden sonra da geçerli kalır" tümevarımına dayanır.

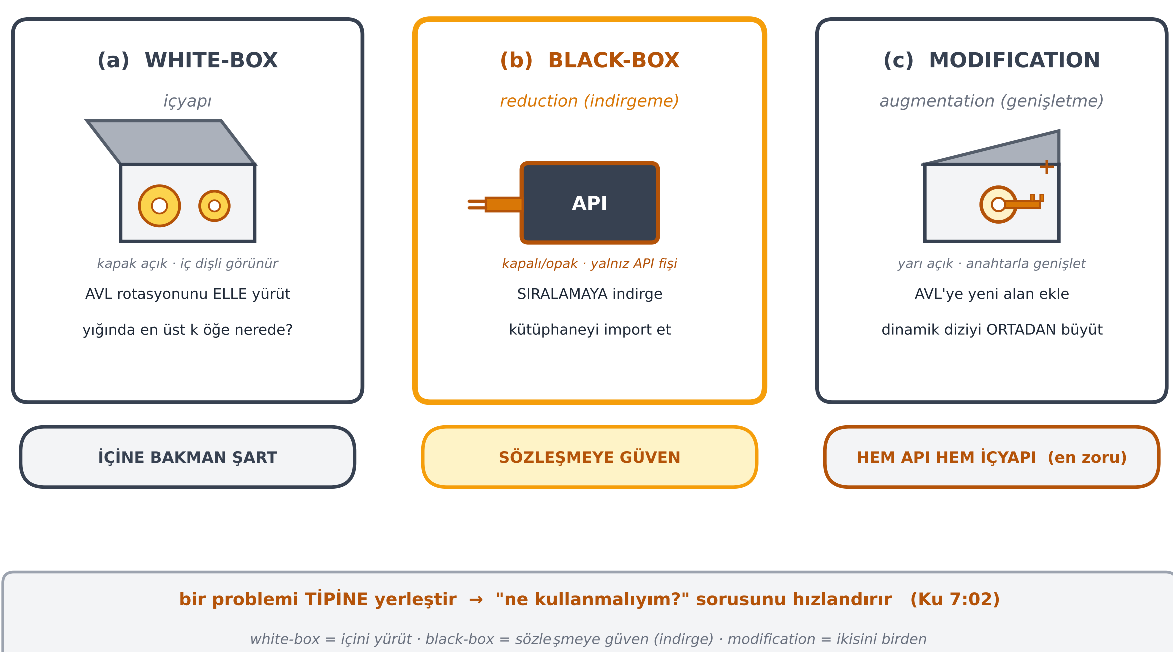
> *"based on the assumption that those invariants held before my operation… I can prove that an operation is correct."* — Ku, 31:56

21.5 4. Sınav Stratejisi ve Kısmi Puan

- **Önce tüm sınavı oku**, sana en kolay gelenleri seç, güven sırasına göre çöz.

> *"read through the entire exam before you start."* — Ku, 11:27

- Ortalama genelde **60-80** arası; problemlerin %50'sini iyi yapmak, hepsini yarım yapmaktan iyidir.
- **Kısmi puan:** doğru ama verimsiz bir algoritma puan alır. Exponential ise %10-20 ile sınırlı; log/linear faktör kadar yavaşsa daha çok puan. Ama verimsiz algoritmayı *hedef* karmaşıklıkmış gibi analiz etmek **iki kat** hata.



Şekil 21.2: Quiz 1'in üç problem tipi (QR1 §2, İMZA figür): bir problemle karşılaşınca onu üç tipten birine yerleştir — tip, 'ne kullanmalıyım?' sorusunu hızlandırır (Ku 7:02). (a) WHITE-BOX / içyapı — kapağı AÇIK kutu, iç dişli görünür: yapının içini elle yürütmen gerekir (AVL rotasyonunu elle yürüt; yığında en üst k öğe nerede). Rozet: İÇİNE BAKMAN ŞART. (b) BLACK-BOX / reduction (AMBER çerçeve = quiz'in ESAS beklediği, Ku 6:17) — kapalı/opak kutu + API fişi: içine bakmadan yalnız sözleşmeye güvenirsin (problemi SIRALAMAYA indirge, kütüphaneyi import et). Rozet: SÖZLEŞMEYE GÜVEN. (c) MODIFICATION / augmentation (en zoru) — yarı açık kutu + anahtar: var olan yapıyı değiştirip yeni alan/davranış eklersin (AVL'ye yeni alan ekle; dinamik diziyi ORTADAN büyüt). Rozet: HEM API HEM İÇYAPI. Motor tanığı: CrossLinkedWaitlist black-box örneği — add(Ali,Bora,Can) → seat() = Ali → [Bora, Can], içine bakmadan sözleşme.

- Çoklu-parça problemler **bağımsızdır**: A'yı çözmeden B çözülebilir.
- **Kod yazmazsın, kod okursun** (Python/pseudocode parçacıkları).

> "*If you're stuck, write down a correct algorithm that's inefficient.*" — Ku, 19:23

21.6 5. Kaçınılacak Tuzaklar (Downsides)

Üç refleks "yanlış yoldasın" işaretidir:

1. **Ondalık / rasyonel / reel sayı hesabı.** Bu kurs yalnızca **tamsayı** öğretti. İki kesri karşılaştırman gerekiyorsa bölme yapma — **çapraz çarpım** ile $O(1)$'de karşılaştır.
2. **Her şeye Radix sort.** Yalnızca tamsayılar polinom-sınırlıysa ($u = n^c$) doğrusal olur. Sınır yoksa exponential olabilir; comparison gereken yerde merge sort ($n \log n$) kullan.
3. **Subtree-property olmayan augmentation.** Bir düğümün augmentation'ı, **iki çocuğunun augmentation'larından $O(1)$'de** hesaplanabilmelidir. "Tüm ağaçtaki indeksim" veya "sol alt-ağacımın boyutu" doğrudan tutulamaz (rotasyonda logaritmik yürüyüş gerekir); doğrusu **alt-ağaç boyutunu** tutup soldakini çocuğundan okumaktır.

> "*if you're trying to augment a binary tree with something that's not a subtree property… you're doing something bad.*" — Ku, 24:32

Augmentation tanımlarken: formülü ver **ve** $O(1)$'de bakım yapıldığını göster.

> "*Max can be computed as the max between me and my left and right subtree.*" — Ku, 27:49

21.7 6. Konu Tekrarı — Sıralama Algoritmaları

Neden bu kadar çok sıralama algoritması? Çünkü farklı senaryolar farklı algoritma ister.

> "*Why do we show you so many sorting algorithms?*" — Ku, 36:45

- **Insertion sort:** k-yakın dizide (öğeler en fazla k uzak) $O(n \cdot k)$ — k küçükse iyi; ama ikili yığınla $O(n \log k)$'ya inilir.
- **Selection sort:** okuma ucuz, **yazma pahalıysa** iyi (doğrusal sayıda swap).
- **Merge sort / AVL sort:** $O(n \log n)$, asimptotik olarak denk; genel amaçlı.
- **Heap sort: worst-case** $O(n \log n)$ ve **yerinde (in-place)**. Anahtar: diziyi bir **tam ikili ağaç (complete binary tree)** olarak görmek — dizi $\leftrightarrow$ ağaç birebir eşleşir (tamlık benzersizliği verir).
- **Radix sort:** tamsayılar polinom-sınırlıysa ($u = n^c$) **doğrusal**; hatta $u = n^c \cdot \log \log n$ gibi durumda merge sort'tan hızlı. Sınır yoksa kötü.

> "*this correspondence between arrays and binary trees.*" — Ku, 40:17

21.8 7. Konu Tekrarı — Sequence Veri Yapıları

Üç temel: **bağlı liste (linked list)**, **dinamik dizi (dynamic array)**, **sequence AVL**.

- Derste sunulan **tekli bağlı liste (singly-linked)** — sonu bulmak/silmek $O(n)$.
- **Çiftli bağlı liste (doubly-linked):** önceki işaretçiyle, uçta ekleme/silme $O(1)$.
- Uçlarda (ön + arka) $O(1)$ için **üç** yol gösterildi: (1) çiftli bağlı liste, (2) sırt sırta iki dinamik dizi (double-ended queue), (3) **hash tablosu + en küçük indeksi saklama** (sequence'i set'le taklit etmek).
- **Sequence AVL:** ortaya $O(\log n)$ ekleme; pratikte nadir, ama teorik olarak dengeli sınırlar verir.

> *"I call this a linked data structure, because I'm linking between two data structures."* — Ku, 1:04:22

21.9 8. Konu Tekrarı — Set Veri Yapıları ve Öncelik Kuyruğu

- **Sıralı dizi (sorted array):** iyi find (ikili arama), ama dinamik değil.
- **Set AVL:** iyi find + dinamik; $O(n \log n)$ build (özünde sıralama).
- **Hash tablosu / direct access array:** sözlük işlemleri (find/insert/delete) çok hızlı; ama **sıra (order)** işlemleri kötü.
- Teori sorusunda set AVL'yi **alt-ağaç max/min** ile zenginleştirirsen, sıralı diziden **her açıdan üstün** olur.
- **Öncelik kuyruğu (priority queue):** ikili yığın; *veya* max/min augmentation'lı sequence AVL (amortizasyonsuz aynı sınırlar).

> *"augment by the max… I can make this one strictly better than this one."* — Ku, 53:38

21.10 Bu Quiz Review'in Özeti

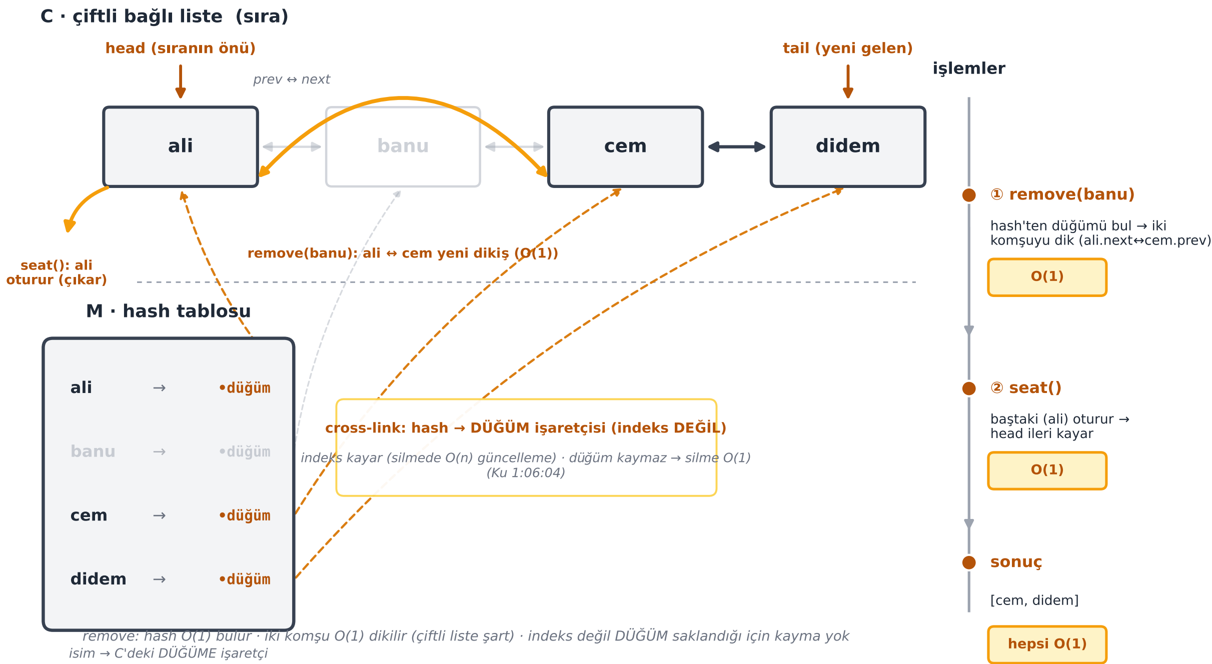
1. **Quiz = veri yapıları + sıralama bloğu** (Ders 1-12); "büyük dört": çöz, kanıtla, verimli yap, anlat.
2. **İki yol**: sıfırdan tasarla (zor) vs **bilinen kara kutuya indirge** (esas beklenen).
3. **Üç problem tipi**: white-box (içyapı) / black-box (reduction) / modification (augmentation).
4. **Reduction disiplini**: önce arayüz (correctness), sonra implementasyon (efficiency); ne sakladığını + neye anahtarlandığını + değişmezleri söyle.
5. **Strateji**: tüm sınavı oku, kolayı seç; sıkışınca doğru-ama-verimsiz yaz.
6. **Tuzaklar**: rasyonel hesabı (çapraz çarp), her şeye radix, subtree-property olmayan augmentation.
7. **Tablolar**: sorting (heap/radix özel), sequence (doubly-linked $O(1)$ uç), set (augmented AVL $\supset$ sorted array).

> **!** Tek Bir Cümle
>
> Quiz 1, icat değil **seçim** sınavıdır: doğru kara kutuyu seç, neye indirgediğini ve neyi sakladığını açıkça yaz, correctness'i invariant'larla kanıtla, sonra verimliliği optimize et.

21.11 Quiz-tarzı Problemler

Aşağıda dört quiz-tarzı problem var; her birinin çözümünü açmadan önce kendin dene. İlk ikisi motorla doğrulanmıştır (gerçek sınav problemleri); son ikisi mekanik kontrol soruları.

Şekil 21.3 Problem 1'in iki bileşenli yapısını gösterir: çiftli bağlı liste C (sıra) + hash tablosu M (isim → düğüm işaretçisi); `remove` ve `seat` her ikisi de $O(1)$.



Şekil 21.3: Criminal Seafood (QR1 Problem 1, Ku 1:06:04): çiftli bağlı liste + hash cross-link → add / remove / seat hepsi O(1). İki bileşen: C = çiftli bağlı liste (sıra ali ↔ banu ↔ cem ↔ didem, her düğüm prev+next), M = hash tablosu (isim → C'deki DÜĞÜME işaretçi — İNDEKS DEĞİL DÜĞÜM: indeks kayar, düğüm kaymaz). Cross-link neden düğüm işaretçisi: bir müşteri ortadan ayrılınca önündeki herkesin indeksi azalır (indeks saklamak O(n) güncelleme); düğüm işaretçisiyle silme O(1) (hash'ten düğümü bul, iki komşuyu dik). Zaman çizgisi MOTORDAN: add ali,banu,cem,didem → [ali,banu,cem,didem]; remove(banu) → ali↔cem dikilir → [ali,cem,didem]; seat() = ali (baş) → [cem,didem]. Tüm işlemler O(1) (hash beklenen, liste worst-case).

ℹ Problem 1 (Criminal Seafood — bekleme listesi): build/add/remove/seat işlemlerinin hepsi O(1). Hangi veri yapıları?

Çözüm.
İki ihtiyaç var: (i) **sıra (extrinsic order)** — önce gelen önce oturur; (ii) **isimle arama** — bir müşteriyi adından bul. Bu, "iki anahtarlı" bir problem → iki veri yapısı + **cross-linking**.

- **Sequence: çiftli bağlı liste (doubly-linked list)** C — müşterileri sırada tutar. (Çiftli, çünkü ortadan silerken iki komşuyu $O(1)$'de dikmek gerekir.)

- **Set: hash tablosu** M — ismi, o müşterinin C'deki **düğümüne işaretçisine** eşler.

Kritik incelik: indeks değil **işaretçi** sakla. İndeks saklarsan, biri oturunca tüm indeksler kayar ve M'yi baştan güncellemen gerekir ($O(n)$); düğüm işaretçisi ise öğe çıkana dek değişmez.

> *"store a pointer… to the node… because the node isn't changing."* — Ku, 1:06:04

İşlemler: **add x** → x'i C'nin sonuna ekle (düğüm v), M'ye x → v koy. **remove x** → M'den v'yi bul, C'den v'yi çıkar (çiftli bağlantıyla dik), M'den sil. **seat** → C'nin başını sil, o müşterinin adını M'den çıkar. Hepsi $O(1)$ (hash beklenen, liste worst-case).
Karmaşıklık. Tüm işlemler $O(1)$ (hash işlemleri *beklenen*; liste işlemleri *worst-case*).

Şekil 21.4 Problem 2'nin imza fikrini gösterir: alt-ağaç-max ile zenginleştirilmiş zaman-AVL'de tek taraflı aralık sorgusu — her düğümde bir taraf toptan halledilir, tek iniş yolu $O(\log n)$.

ℹ Problem 2 (Rainy Research — peak rainfall): record_data O(1)-benzeri ve peak_rainfall(l, t) worst-case O(log n). t zamanından beri l enlemindeki en yüksek yağış.

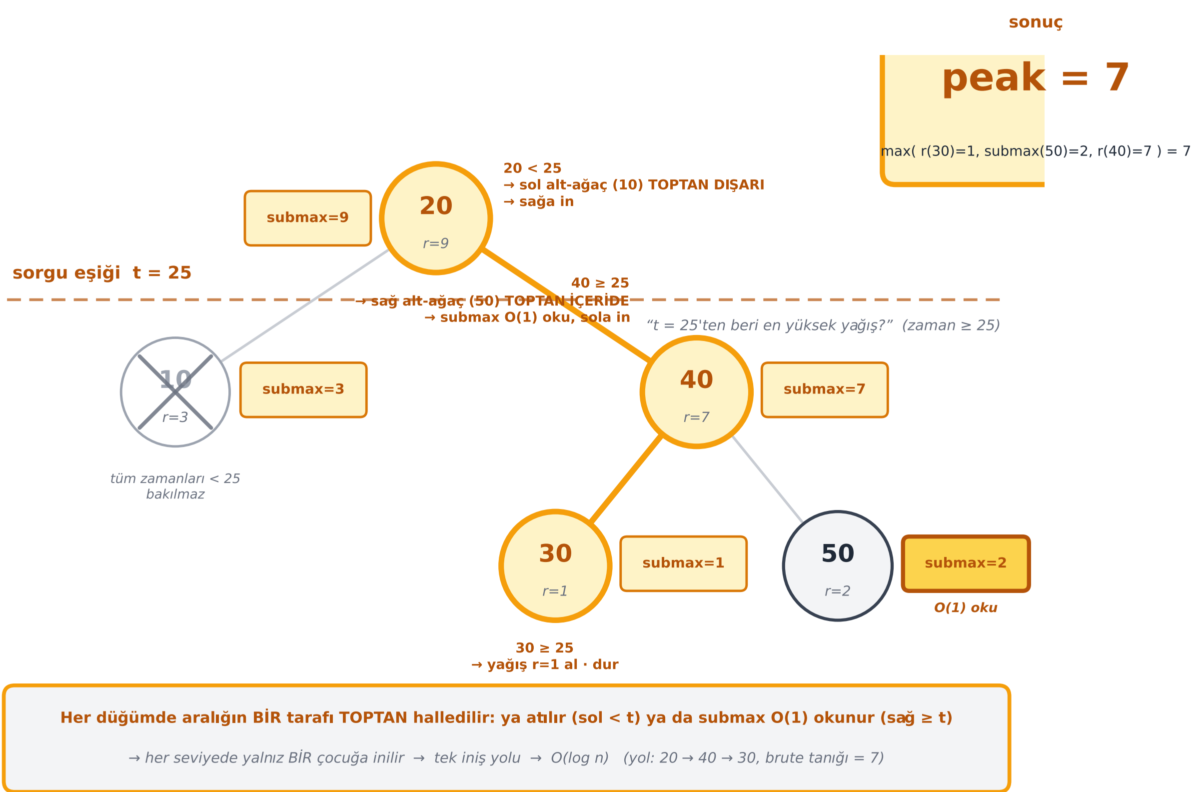
Çözüm.
Ölçümler (r yağış, l enlem, t zaman) üçlüsü; hepsi tamsayı, sınırsız → enlemin "küçük" olduğunu varsayma. Worst-case istendiği için **hash değil set AVL**.

- **Dış yapı:** enlem-anahtarlı **set AVL** L — her enlem l'yi bir "zaman veri yapısına" T(l) eşler.
- **İç yapı:** her T(l), **zaman-anahtarlı set AVL** — zamanı yağışa eşler.
- **Augmentation:** her düğümde **alt-ağaçtaki maksimum yağış** (alt-ağaç max r). Bu bir subtree-property'dir: düğüm = max(kendi r, sol alt-ağaç max, sağ alt-ağaç max), $O(1)$'de bakım.

One-sided range query (tek-taraflı aralık sorgusu): "t'den büyük-eşit tüm zamanlardaki max yağış"ı $O(\log n)$'de bul. Özyinelemeli, düğüm v'de iki durum:

```
def peak(v, t):              # v koku, t alt sinir
    if v is None:
        return 0
    if v.time < t:           # v + tum SOL alt-agac (< v.time < t) araligin DISINDA
        return peak(v.right, t)
    else:                    # v araliginda: SAG alt-agacin tamami da araliginda
        return max(v.r,
                   v.right.submax if v.right else 0,  # O(1): sag submax OKU
                   peak(v.left, t))                   # belirsiz SOLA in
```

[motor notu] — Kaynak sayfadaki pseudocode'un yönleri ters yazılmıştı (sol submax okuyup sağa iniyordu); bu, sol alt-ağaçtaki t'den küçük zamanları yanlışlıkla dahil eder. Bağımsız brute-force tanığı farkı doğruladı: ters yön 1000 sorgudan 193'ünde yanlış değer verdi, doğru yön (v.right.submax oku + peak(v.left)) 1000/1000 brute ile eşleşti. Statement ("t'den beri max") bağlayıcı; yukarıdaki kod doğru yönü uygular.
Püf nokta: aralık içindeyken **sağ alt-ağaca inmek yerine** onun augmented max'ını $O(1)$'de oku; yalnızca sola tek bir özyinelemeli iniş yap → ağaç boyunca tek yol → $O(\log n)$.



Şekil 21.4: Tek taraflı aralık sorgusu (QR1 Problem 2, İMZA figür, Ku 1:24:47): alt-ağaç-max zenginleştirme → peak_rainfall O(log n). Zaman-anahtarlı AVL (5 düğüm): kök 20, çocukları 10 ve 40, 40'ın çocukları 30 ve 50; her düğümde zaman + yağış r + amber submax kutusu (alt-ağaç max, MOTORDAN). Sorgu t=25 ('t'den beri en yüksek yağış', zaman ≥ 25). İmza fikir: her düğümde aralığın BİR tarafı TOPTAN halledilir → tek iniş yolu. 20 < 25 → SOL alt-ağaç (10) TOPTAN DIŞARI (soluk + X), sağa in; 40 ≥ 25 → SAĞ alt-ağaç (50) TOPTAN İÇERİDE → submax(50) O(1) OKU (inilmez), sola in; 30 ≥ 25 → r=1 al, dur. Sonuç MOTORDAN: peak = max(r(30)=1, submax(50)=2, r(40)=7) = 7 = brute tanığı; yol = [20, 40, 30]. Her seviyede yalnız BİR çocuğa inilir → O(log n).

> *“that’s what we call a one-sided range query.”* — Ku, 1:24:47

Karmaşıklık. record_data $O(\log n)$ (iki AVL’ye ekleme); peak_rainfall **worst-case** $O(\log n)$.

Problem 3 (mekanik — geçerli augmentation): Bir set AVL’yi ‘sol alt-ağacımdaki düğüm sayısı’ ile zenginleştirmek geçerli mi?

Cevap:
Doğrudan **hayır** — ama dolaylı **evet**. “Sol alt-ağaç boyutu”, rotasyon/güncellemede iki çocuğun augmentation’ından $O(1)$’de türetilemez (logaritmik yürüyüş gerekir), yani tek başına bir subtree-property değildir. Doğru yol: her düğümde **kendi alt-ağaç boyutunu** (`size = 1 + sol_size + sağ_size`, $O(1)$ bakım) tut; “sol alt-ağaç boyutu” gerektiğinde sol çocuğun size’ından **okunur**. Genel kural: depolanan augmentation daima bir subtree-property olmalı; türev bilgiler ondan $O(1)$’de hesaplanır.

> *“just augment the thing itself and just look at your left subtree and look at its augmentation.”*
> — Ku, 25:42

Bu `size` augmentation’ı, **rank sorgusu** (k. en küçük, tek-taraflı aralık) verir — sequence AVL’nin `subtree_at`’inin set karşılığı.

Problem 4 (mekanik — worst case vs expected): Bir hash tablosunda O(1) beklenen arama yapıp ardından bir AVL’de O(log n) predecessor sorgusu yapıyorum. Bu zincirin worst-case ve beklenen süresi?

Cevap:

- **Worst-case:** $O(n)$**.** Hash tablosunun worst-case araması $O(n)$’dir (tüm anahtarlar aynı kovaya düşebilir); AVL $O(\log n)$. Toplam worst-case $O(n + \log n) = O(n)$.
- **Beklenen:** $O(\log n)$**.** Hash beklenen $O(1)$ + AVL $O(\log n) = O(\log n)$.

Ders: “beklenen” ve “worst-case” farklı sözleşmelerdir. Sınav “worst-case istiyorum” derse hash tablosu kullanma (set AVL’ye geç); “expected/amortized serbest” derse hangi sınıfı elde ettiğini **etiketle**.

> *“It’s possible that in the worst case it could be higher.”* — Ku, 30:15

21.12 Quiz Hazırlığı Egzersizleri

Egzersiz 1. Sıralama tablosunu boş bir kâğıda ezberden doldur: her algoritma için worst-case süre, kararlı mı (stable), yerinde mi (in-place), hangi özel durumda tercih edilir.

Egzersiz 2. Sequence, set ve priority queue arayüzlerinin işlem-süre tablolarını ezberden çıkar; her hücreyi “hangi implementasyon bunu verir?” ile eşle.

Egzersiz 3. Üç master-teoremi durumunu üç farklı özyinelemeye uygula (örn. $T(n) = 2T(n/2) + O(n)$, $T(n) = 2T(n/2) + O(n^2)$, $T(n) = 4T(n/2) + O(n)$).

Egzersiz 4. Bir set AVL için yeni bir augmentation tasarla (örn. alt-ağaç toplamı), formülünü yaz ve rotasyonda $O(1)$'de korunduğunu kanıtla.

Egzersiz 5. İki anahtarlı (örn. isim + öncelik) bir veri yapısı problemini cross-linking ile çöz: hangi veri yapılarını, neye anahtarlanmış, hangi değişmezlerle kurarsın?

21.13 Quiz 2 Öncesi Kapsam Genişlemesi

Quiz 1 buraya kadar; sıradaki blok (**Quiz 2** kapsamı) **çizge algoritmalarına** geçer:

- **Ders 13 ve 15 (L9-L10):** çizge temeli, **BFS** (ağırlıksız en kısa yol, $O(V+E)$), **DFS** (topolojik sıralama, çevrim).
- **Ders 16, 18-19 (L11-L13; araya Ders 17 PS5 girer): ağırlıklı** en kısa yollar — Bellman-Ford (negatif kenar), Dijkstra (öncelik kuyruğu).
- **Ders 21 (L14):** tüm-çiftler en kısa yollar (Johnson).

Bağ: bu blokta da aynı disiplin sürer — çizgeyi komşuluk listesiyle sakla (bir veri yapısı seçimi), problemi BFS/DFS/Dijkstra **kara kutusuna indirge**, sonra verimliliği analiz et.

21.14 Ders 1-12 Toplu Cheat Sheet

Konu	Özü	Kaynak (L/PS)
Hesaplama modeli	Word-RAM; $O(1)$ işlemler; tamsayı bir kelimeye sığar	L1
Asimptotikler	O, Θ, Ω; karmaşıklık mertebeleri	L1-L2
Master teoremi	$T(n) = a \cdot T(n/b) + f(n)$; 3 durum	PS2
Sıralama	merge/AVL ($n \log n$), heap (worst $n \log n$, in-place), radix ($n^c \to$ linear)	L3, L5
Sequence DS	doubly-linked ($O(1)$ uç), dynamic array, sequence AVL ($O(\log n)$ orta)	L2, L7
Set DS	sorted array (statik), set AVL (dinamik), hash (sözlük hızlı / sıra kötü)	L3
Hashing	beklenen $O(1)$; zincirleme; universal hashing	L4
İkili ağaç / AVL	$O(h)$; denge $\to h = O(\log n)$; rotasyon; augmentation	L6-L7
İkili yığın	complete tree $\leftrightarrow$ array; build $O(n)$, delete_max $O(\log n)$	L8

Konu	Özü	Kaynak (L/PS)
Öncelik kuyruğu	binary heap veya max/min-augmented sequence AVL	L8

Sonraki Ders

Ders 15 (L10): Derinlemesine Arama (DFS)
Justin Solomon ile, BFS'in kardeşi **DFS (depth-first search)**'e geçiyoruz: çizgeyi "olabildiğince derine in, sonra geri çekil" mantığıyla gez. DFS, **topolojik sıralama** (bağımlılık sıralaması) ve **çevrim tespiti** için kullanılır.

21.15 Builder ve OMSCS Bağlantıları

6 köprü

Bu tekrar oturumu, "doğru kara kutuyu seç, indirge, sakladığını ve anahtarladığını yaz" disiplinini kurar — köprülerin özeti:

1. **Quiz 1 = ilk-yarı midterm** → OMSCS CS 6515: veri yapıları + sıralama refleksleri, graduate dersin giriş varsayımıdır.
2. **Reduction → her şey** → "kara kutuya indirge" disiplini, DP ve NP-tamlıkta (problem A'yı B'ye indirgemek) birebir döner.
3. **Correctness/efficiency ayrımı** → mühendislikte "önce çalışsın, sonra hızlansın"; profilleme öncesi doğruluk.
4. **Cross-linking** → gerçek sistemler: aynı veriyi iki indeksle (örn. ID + zaman) tutup işaretçiyle bağlamak — veritabanı ikincil indeksi.
5. **Augmentation = subtree-property** → aralık sorguları (segment tree, order-statistics tree) — finans/analitik altyapısının çekirdeği.
6. **İletişim puana dahil** → kod review, tasarım dokümanı: "doğru ama anlatılamayan = yanlış".

! Tek bir şey alıp gideceksen

Quiz 1 senden algoritma icat etmeni değil, **doğru kara kutuyu seçmeni** ister. Problemi bir arayüze indirge, ne sakladığını ve neye anahtarlandığını açıkça yaz, doğruluğu değişmezlerle kanıtla — verimlilik en son gelir. "Doğru ama anlatılamayan çözüm, yanlış çözümdür."

22 Derinlemesine Arama (DFS)

DFS = özyinelemeli derine in, geri çekil; erişilebilirlik O(E) (ebeveyn ağacı, en kısa değil); full DFS → bağlı bileşenler O(V+E); DAG'da ters bitiş sırası = topolojik sıra (u→v ⇒ f(u)<f(v), benzersiz değil); geri kenar (v → atası) = çevrim; bağımlılık çözen her sistem (make/npm/pip) içten içe bunu çalıştırır

> Bölüm bilgisi
>
> - **Solomon'un videosu:** YouTube — Lecture 10: Depth-First Search (≈52 dk)
> - **OCW sayfası:** MIT 6.006 Lecture 10: Depth-First Search
> - **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 15 (L10)
> - **Hoca:** Justin Solomon (Demaine/Ku değil)
> - **Okuma süresi:** ≈25 dk
>
> Bir önceki ders BFS'i (enine arama) verdi; bu ders onun kardeşi **DFS**'i çözer ve üç güçlü uygulamayı açar: erişilebilirlik, bağlı bileşenler, topolojik sıralama (+ çevrim tespiti).

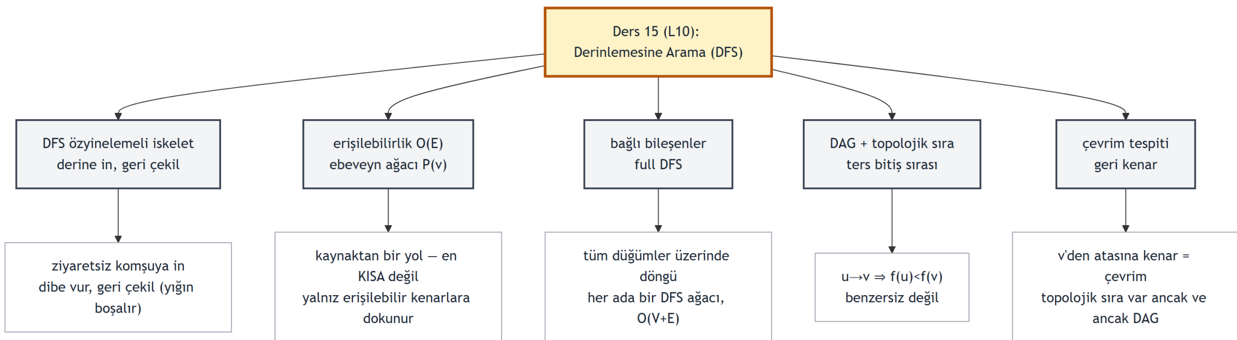
22.1 Bu Derste Ne Var?

Ders 13 (L9) BFS'i (enine arama) verdi; Ders 14 (Quiz 1 Gözden Geçirme, Jason Ku) araya girdi. Bugün yine Justin Solomon ile BFS'in kardeşi **DFS (depth-first search, derinlemesine arama)** ile devam ediyoruz. DFS, çizgeyi "olabildiğince derine in, sonra geri çekil" mantığıyla gezer ve üç güçlü uygulamayı açar: **erişilebilirlik**, **bağlı bileşenler**, **topolojik sıralama** (+ çevrim tespiti).

> *"in breadth-first search, we're like, drawing concentric circles. In depth-first search… we're like, shooting outward until we reach the outer boundary."* — Solomon, 6:55

Üç ana fikir bu derste yan yana gelir:

1. **DFS = özyinelemeli gezinme** — bir düğümden komşularına in, her komşu kendi komşularına insin; erişilebilirliği $O(E)$'de çöz.
2. **Bağlı bileşenler** — full DFS (her düğüm üzerinde döngü) ile $O(V+E)$'de tüm "kümeleri" bul.
3. **Topolojik sıralama** — bir DAG'da, DFS bitiş sırasının **tersi** geçerli bir bağımlılık sıralaması verir; aynı teknik **çevrim tespiti** yapar.



Şekil 22.1: Ders 15'in (L10) kavram haritası: kök = derinlemesine arama (DFS). Beş dal — (1) DFS özyinelemeli iskelet: bir düğümden ziyaretsiz komşuya in, dibe vur, geri çekil (özyineleme çözülür). (2) erişilebilirlik O(E): ebeveyn ağacı P(v) ile kaynaktan bir yol — en kısa DEĞİL; DFS yalnız erişilebilir kenarlara dokunur. (3) bağlı bileşenler: full DFS tüm düğümler üzerinde döngü → her ada bir DFS ağacıyla O(V+E)'de bulunur. (4) DAG + topolojik sıra: yönlü çevrimsiz çizgede ters bitiş sırası = topolojik sıralama (u→v ⇒ f(u)<f(v)), benzersiz değil. (5) çevrim tespiti: geri kenar (v'den atasına kenar) bir çevrimi tamamlar; topolojik sıra var ancak ve ancak DAG. Sonuç: tek özyinelemeli iskelet, dört güç.

Builder Notu — Tek İskelet, Dört Güç

DFS, "derine in, geri çekil" diyen tek bir özyinelemeli iskelettir — ama o tek iskeletten dört ayrı güç çıkar. Önce nasıl gezdiğini (özyineleme), sonra ne sorabildiğini (erişilebilirlik, bileşenler, sıra, çevrim) sorarız.

- **İleriye → topolojik sıralama her yerde:** `make`, paket yöneticileri (npm/pip bağımlılık çözümü), build sistemleri, görev zamanlayıcılar — hepsi DAG'da topolojik sıralama çalıştırır.
- **İleriye → çevrim tespiti:** deadlock tespiti, döngüsel bağımlılık uyarısı (import cycle), elektronik tablo formül döngüsü.
- **Geriye → BFS (Ders 13):** ikisi de $O(V+E)$ çizge gezme iskeleti; BFS en kısa yol (ağırlıksız) verir, DFS yapısal sorular (DAG mı? bileşenler? sıra?) için.
- **İleriye → Dijkstra (Ders 16-19 (L11-L13)):** DFS/BFS ağırlıksız; ağırlık girince öncelik kuyruğu gelir.

Tek cümle: *DFS, çizgeyi özyinelemeli olarak derinlemesine gezer; bu tek iskeletten erişilebilirlik ($O(E)$), bağlı bileşenler ($O(V+E)$) ve topolojik sıralama/çevrim tespiti çıkar.*

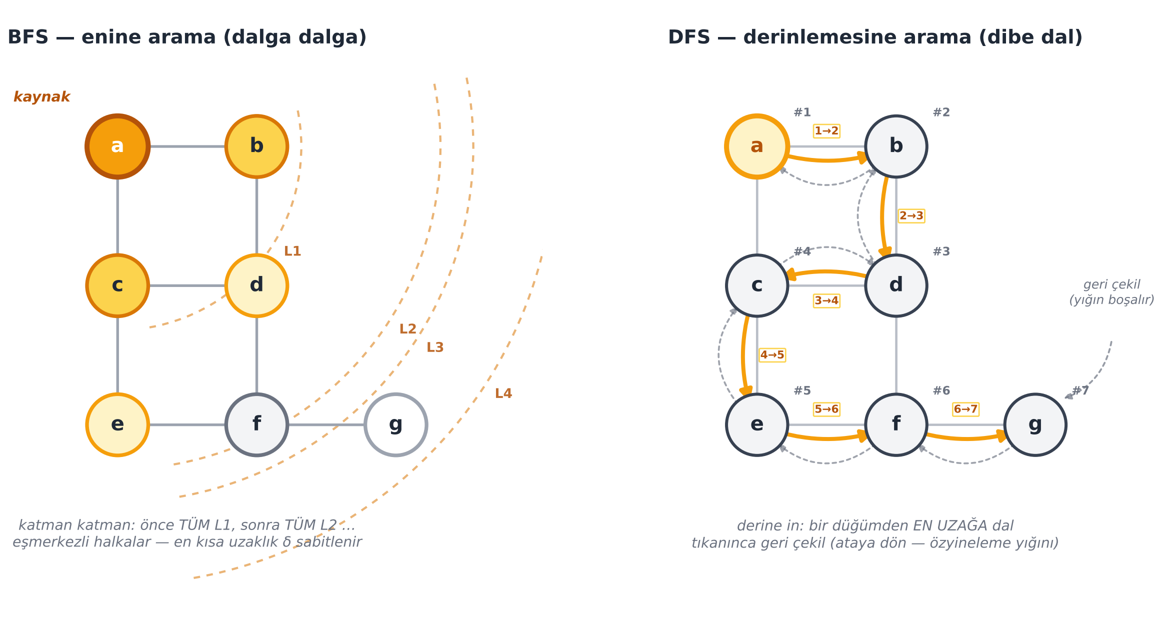
22.2 1. BFS'ten DFS'e: İki Arama Stratejisi

İki temel çizge gezme tekniği vardır. **BFS** kaynaktan dışa **eşmerkezli daireler** çizer: önce L_1 (uzaklık 1), tüm L_1 bitmeden L_2'ye geçilmez. **DFS** ise tam tersi: ilk düğümden başlar, gidebildiği kadar **derine** koşar, tıkanınca **geri çekilir (backtrack)**.

> *"shooting outward until we reach the outer boundary, and then exploring the graph that way."* — Solomon, 6:55

İkisi de aynı çizge terminolojisi üzerine kurulu: $G = (V, E)$, $\mathrm{Adj}^+(u)$ çıkış komşuları, yol (path), basit yol (simple path — aynı düğüm iki kez yok). "Doğrusal zaman" çizgede $O(V + E)$ demektir (girdiyi saklamak için gereken yer).

Şekil 22.2 aynı 7 düğümlü çizge üzerinde iki stratejiyi yan yana koyar: solda BFS dalga dalga (seviye tonları + eşmerkezli halkalar), sağda DFS tek bir kıvrımlı izle dibe dalar (ziyaret sırası $a \to b \to d \to c \to e \to f \to g$) ve tıkanınca geri çekilir.



Şekil 22.2: İki arama stratejisi — AYNI çizge: BFS dalga dalga · DFS dibe dalar (L10 §1, İMZA figür). SOL panel BFS = kaynaktan eşmerkezli halkalar, katman katman (önce TÜM L1={b,c}, sonra TÜM L2={d,e}…); her düğüm seviye tonuyla (L0 a amber kaynak → L4 g beyaz+slate). SAĞ panel DFS = aynı çizge, tek AMBER kıvrımlı ok zinciri dibe dalar (ziyaret sırası a→b→d→c→e→f→g; her ok üstünde sıra numarası); sona varınca geri-çekilme (kesikli gri geri-oklar, özyineleme yığını boşalır). Kontrast: BFS b ve c'yi BİRLİKTE (aynı katman) görür; DFS b'den HEMEN d'ye dalar, c'ye ancak geri çekilince 4. sırada varır. BFS = ağırlıksız en kısa yol; DFS = yapısal sorular (DAG mı? bileşenler? topolojik sıra? çevrim?). Veri MOTORDAN: bfs_levels(build_example_graph(),'a')=[[a],[b,c],[d,e],[f],[g]]; dfs_visit_order(…,'a')['order']=[a,b,d,c,e,f,g] (Solomon 6:55).

22.3 2. Erişilebilirlik Problemi ve Ebeveyn Ağacı

Erişilebilirlik (reachability): bir kaynak s'den yönlü kenarları izleyerek **hangi düğümlere ulaşılır?**

> *"the reachability problem is just asking, which nodes can I reach from a given source?"* — Solomon, 8:20

BFS ile çözülebilir (erişilemeyen düğümün mesafesi ∞), ama daha doğrudan bir yol var. “Kanıt” istersek (sadece “ulaşılır” değil, “nasıl”), her düğümü **öncülü (predecessor)** $P(v)$ ile etiketleriz — kaynaktan o düğüme giden *bir* yolda önceki düğüm. Yolu, P’yi geriye izleyip listeyi ters çevirerek kurarız.

Önemli fark (Ders 13’ün en kısa yol ağacından): buradaki **ebeveyn ağacı** en kısa yolu garanti etmez — erişilebilirlikte yolun kısalığı umurumuzda değil, sadece varlığı. Yine de bir ağaçtır (çevrim olamaz).

22.4 3. DFS Algoritması

DFS özyinelemeli bir `visit` fonksiyonudur: kaynağı işaretle; her komşu v için, **henüz ziyaret edilmemişse** ($P(v) =$ None) ebeveynini “ben” yap ve özyinelemeli olarak ona in.

Çalışılan Örnek — gezinme sırası. Kaynak 1’den başla. 1’in tek komşusu 2 → visit(2). 2’nin komşuları 3 ve 5; önce 3 → visit(3) → visit(4). Özyineleme dibe vurdu, geri çekil. 2 sonraki komşusu 5’e iner → visit(5). Sıra: $1 \to 2 \to 3 \to 4 \to (\text{geri}) \to 5$. Dikkat: bu **seviye kümelerini izlemez** — DFS dibe (4) gidip sonra 2’ye geri sarıp 5’i çağırdı. “Geri çekilme” = özyinelemenin çözülmesi.

```
def dfs(adj, s, parent=None):
    if parent is None:
        parent = {s: None}        # kaynagi ziyaret et
    for v in adj[s]:
        if v not in parent:       # henuz ziyaret edilmemis
            parent[v] = s         # ebeveyni = ben
            dfs(adj, v, parent)   # ozyinele
    return parent
```

Şekil 22.3 bu çalışılan örneği adım adım izler: üstte yönlü çizge ($1 \to 2 \to 3 \to 4$ dibe in, sonra geri çekilip 5’e dal), altta olay şeridi — in/out olaylarının sırası, out(4), out(3) olaylarının in(5)’ten **önce** gelmesi (dibe vur, geri sar, sonra 5).

22.5 4. DFS Doğruluğu

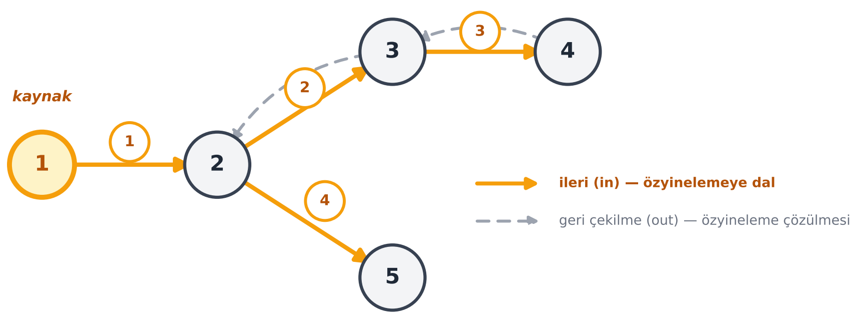
İddia: DFS, erişilebilir her v’yi ziyaret eder ve ebeveynini doğru atar.

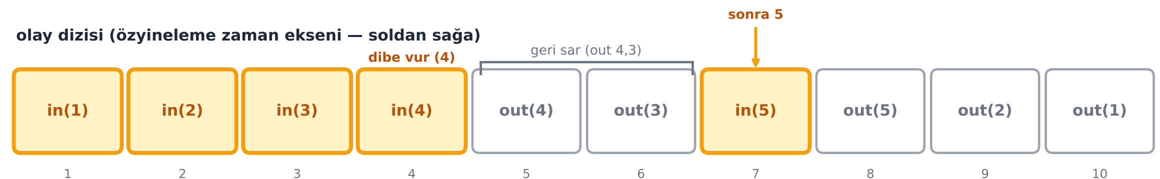
Çalışılan Örnek — kaynağa uzaklık üzerinden tümevarım. Kaynağa uzaklık k üzerinden tümevarım.

- **Temel ($k = 0$):** uzaklık 0 olan tek düğüm kaynağın kendisidir; algoritma ilk satırda onu doğru kurar. ✓
- **Adım ($k \to k + 1$):** v, kaynaktan uzaklık $k + 1$ olsun. En kısa yolda v’den bir önceki düğüm u’nun uzaklığı k’dır → tümevarım varsayımıyla u doğru ziyaret edilir. visit(u) çağrıldığında, $v \in \text{Adj}^+(u)$ olduğundan DFS v’yi değerlendirir. İki durum: ya $P(v) \neq$ None (zaten uygun bir ebeveyn bulunmuş) ya da $P(v) =$ None (bir sonraki satır ebeveyni doğru atar). Her iki durumda da v’nin ebeveyni doğru. ✓

Sezgisel olarak basit; biçimsel tümevarım totoloji gibi hissettirebilir ama değil.

DFS çalışılan örnek: 1→2→3→4 dibe in, sonra geri çekilip 5'e dal (Solomon L10 §3)





seviye kümeleri İZLENMEZ — dibe vur (4), geri sar, sonra 5 · geri çekilme = özyinelemenin çözülmesi (out, in'lerin TERSİ sırada kapanır)

Şekil 22.3: DFS çalışılan örnek: 1→2→3→4 dibe in, sonra geri çekilip 5'e dal (L10 §3, İMZA figür). ÜST: yönlü çizge — kenarlar 1→2, 2→3, 2→5, 3→4. Ziyaret (ileri) okları AMBER + sıra numarası: 1→2 [1], 2→3 [2], 3→4 [3], sonra 2→5 [4]. Geri-çekilme okları KESİKLİ GRİ: 4→3, 3→2 (özyineleme çözülmesi; 5 ziyaretinden ÖNCE). ALT: olay şeridi — in(1) in(2) in(3) in(4) out(4) out(3) in(5) out(5) out(2) out(1); 'in' amber dolu, 'out' soluk çerçeve. out(4) ve out(3), in(5)'ten ÖNCE: dibe vur (4), geri sar (out 4,3), sonra 5. Seviye kümeleri İZLENMEZ — geri çekilme = özyinelemenin çözülmesi (out'lar in'lerin TERSİ sırada kapanır). Veri MOTORDAN: dfs_visit_order(build_dfs_example(),1)['order']=[1,2,3,4,5]; events=[in1,in2,in3,in4,out4,out3,in5,out5,out2,out1] (Solomon L10 §3).

22.6 5. DFS Çalışma Süresi O(E)

Her düğüm en fazla bir kez ziyaret edilir (visit her düğüm için bir kez çağrılır); her kenar en fazla bir kez gezilir (kenarın "çıkış" düğümü bir kez ziyaret edildiğinden). Önemli ayrım: BFS, başta $O(V)$ yer ayırdığı için $O(V + E)$'dir; DFS ise yalnızca **erişilebilir** kenarlara dokunur, hiç erişilemeyen düğümü görmez → $O(E)$.

> *"depth-first search… it just takes order e time."* — Solomon, 22:09

(Kenarsız bir çizgede BFS yine $O(V)$, DFS ise $O(E) = O(0)$ işi yapar; sınırlar birebir aynı değildir.)

22.7 6. DFS En Kısa Yol Vermez

DFS'in ürettiği yol **en kısa olmak zorunda değildir**.

> *"there's no reason why the path that we get should be the shortest."* — Solomon, 24:01

Uç örnek: bir **çevrim çizgesi** (büyük halka). Kaynaktan başlayıp komşu komşuyu özyinelemeli çağırırsak, son düğüme 4 düğümlük bir zincirin arkasından varırız — oysa tek bir kenarla doğrudan gidilebilirdi. DFS o kenarı *seçmedi*. Bu yüzden ağırlıksız en kısa yol için **BFS** kullanılır; DFS yapısal sorular içindir.

22.8 7. Bağlılık ve Bağlı Bileşenler

Bir çizge **bağlı (connected)** ise her düğümden her düğüme bir yol vardır.

> *"a graph is connected if there's a path getting from every vertex to every other vertex."* — Solomon, 25:14

Yönlü çizgede bağlılık tuhaftır ($u \to v$ var ama $v \to u$ yok olabilir), bu yüzden çoğunlukla **yönsüz** çizgelerde konuşulur: düğümler "kümeler" hâlinde gelir — bir **bağlı bileşen (connected component)**.

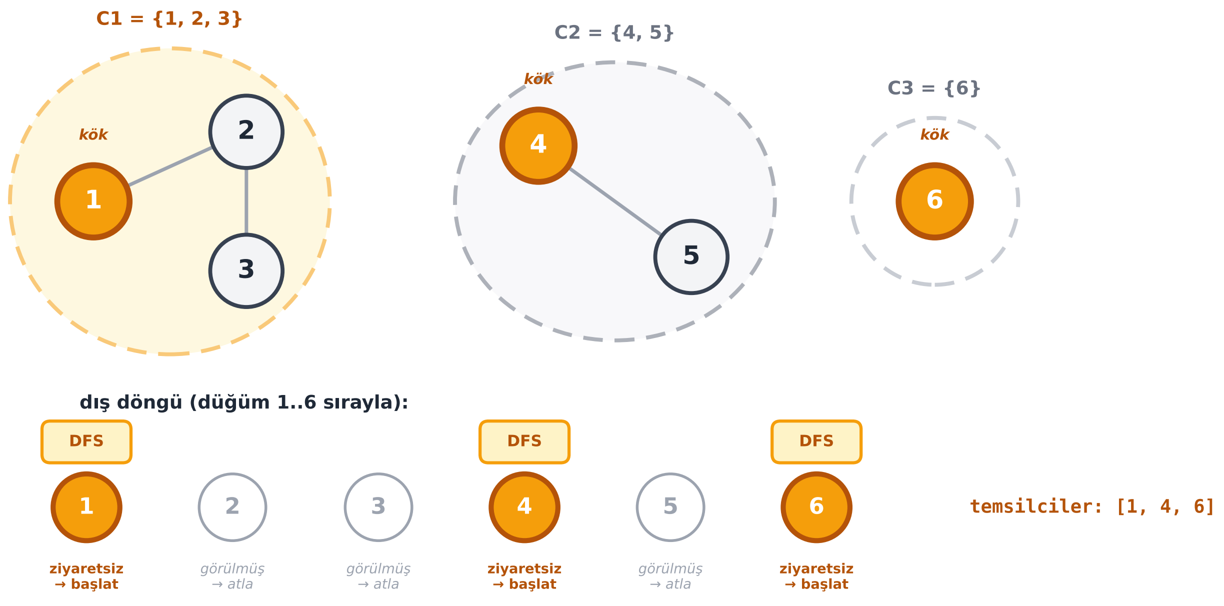
Bağlı bileşenler problemi: tüm kümeleri listele. Çözüm **full DFS**: tüm düğümler üzerinde bir döngü; düğüm henüz ziyaret edilmemişse ondan DFS başlat (o bileşeni "flood fill" eder), topla, devam et. Her kenar en fazla bir kez gezilir (bir düğüm iki bileşende olamaz) → $O(V + E)$.

```
def full_dfs(adj, V):
    parent = {}
    components = []
    for s in V:                   # tum dugumler uzerinde dongu
        if s not in parent:       # yeni bilesen
            parent[s] = None
            dfs(adj, s, parent)
            components.append(s)   # bu bilesenin temsilcisi
    return components, parent
```

Şekil 22.4 full DFS'i "ada keşfi" olarak gösterir: üç bağlı bileşen ($\{1, 2, 3\}$, $\{4, 5\}$, $\{6\}$), her birinin temsilci kökü $(1, 4, 6)$, ve altta dış-döngünün düğümleri sırayla tarayışı (ziyaretsiz köke yeni DFS başlat, görülmüşü atla).

Full DFS = bağlı bileşenler: ziyaretsiz her köke yeni DFS başlat (ada ada)

FULL DFS → bağlı bileşenler: her ada bir DFS ağacıyla taranır



dış döngü her düğüme 1 kez bakar + her kenar en fazla 1 kez gezilir → O(V + E) · (Solomon L10 §7)

Şekil 22.4: Full DFS = bağlı bileşenler: ziyaretsiz her köke yeni DFS başlat (ada ada) (L10 §7, İMZA figür). ÜST: 3 ada çizgesi — küme 1 {1,2,3} (üçgen-vari, amber halo, bileşen C_1), küme 2 {4,5} (çift, slate halo, C_2), küme 3 {6} (tek düğüm, beyaz halo, C_3); her ada altında bileşen etiketi. Temsilci kökler (her bileşenin ilk ziyaret edilen düğümü) amber dolgulu + 'kök' etiketi: 1, 4, 6. ALT: full DFS dış-döngü şeridi — düğümler 1..6 SIRAYLA taranır; 1 ziyaretsiz → DFS başlat (ada 1'i sular), 2,3 görülmüş → atla, 4 ziyaretsiz → DFS başlat (ada 2), 5 atla, 6 ziyaretsiz → DFS başlat (ada 3); amber 'DFS' rozeti her başlatmada. Dış döngü her düğüme 1 kez bakar + her kenar en fazla 1 kez gezilir → O(V+E). Veri MOTORDAN: connected_components=[[1,2,3],[4,5],[6]]; full_dfs temsilcileri=[1,4,6] (Solomon L10 §7).

22.9 8. Yönlü Çevrimsiz Çizge (DAG) ve Topolojik Sıralama

Bir **DAG (Directed Acyclic Graph)**: yönlü kenarlı, **çevrimsiz** çizge. (Yönlendirilmiş bir ağaç bir DAG örneğidir.)

> *"a DAG is a Directed Acyclic Graph."* — Solomon, 32:28

Topolojik sıralama: her düğüme bir sıra indeksi f atayan, şu özelliği sağlayan bir düzen: her $u \to v$ kenarı için $f(u) < f(v)$ (u, v'den önce gelir).

> *"if I have a directed edge from u to v, then f of u is less than f of v."* — Solomon, 34:46

Topolojik sıralama **benzersiz değildir** — $A \to B$, $A \to C$, $B \to D$, $C \to D$ çizgesinde hem "A C B D" hem "A B C D" geçerlidir. Bu "özgürlük", algoritmanın çoktan birini bulmasına izin verir.

22.10 9. Bitiş Sırası → Topolojik Sıralama

Bitiş sırası (finishing order): full DFS çalıştır; bir düğümün visit çağrısı **tamamlandığında** (tüm komşuları gezildiğinde) onu sıraya ekle. **İddia:** bir DAG'da, **bitiş sırasının tersi** geçerli bir topolojik sıralamadır.

> *"the reverse of the finishing order is a topological order."* — Solomon, 37:54

Çalışılan Örnek — kanıt (iki durum). Bir $u \to v$ kenarı al; ters bitiş sırasında u'nun v'den önce geldiğini göstermeliyiz.

- **Durum 1 — u, v'den önce ziyaret edildi:** visit(u), v'yi (henüz ziyaretsizse) çağırır; visit(v), visit(u)'dan **önce tamamlanır**. Ters sırada bu, u'yu v'nin önüne koyar. ✓
- **Durum 2 — v, u'dan önce ziyaret edildi:** çizge çevrimsiz olduğundan v'den u'ya yol **yoktur** (olsaydı çevrim olurdu). O hâlde visit(v), u'yu hiç görmeden tamamlanır → Durum 1'le aynı sonuç: ters sırada u, v'den önce. ✓

Şekil 22.5 bu mekanizmayı gösterir: örnek DAG, full DFS'in **bitiş sırası** $[D, B, C, A]$ (D yaprak, ilk biter; A kök, son biter), ve onun **tersi** = topolojik sıra $[A, C, B, D]$; her DAG kenarı için $f(u) < f(v)$ onayı yeşil rozetle, ve "benzersiz değil" notu ($[A, B, C, D]$ de geçerli).

22.11 10. Çevrim Tespiti

DAG değilsek topolojik sıralama elde edemeyiz. Yani: ters bitiş sırasını hesapla, sonra her kenarın $f(u) < f(v)$ ilişkisine uyup uymadığını kontrol et. **Uymayan bir kenar bulursak çizge DAG değildir → bir çevrim vardır.** (Aslında: topolojik sıralama var $\iff$ çizge DAG.)

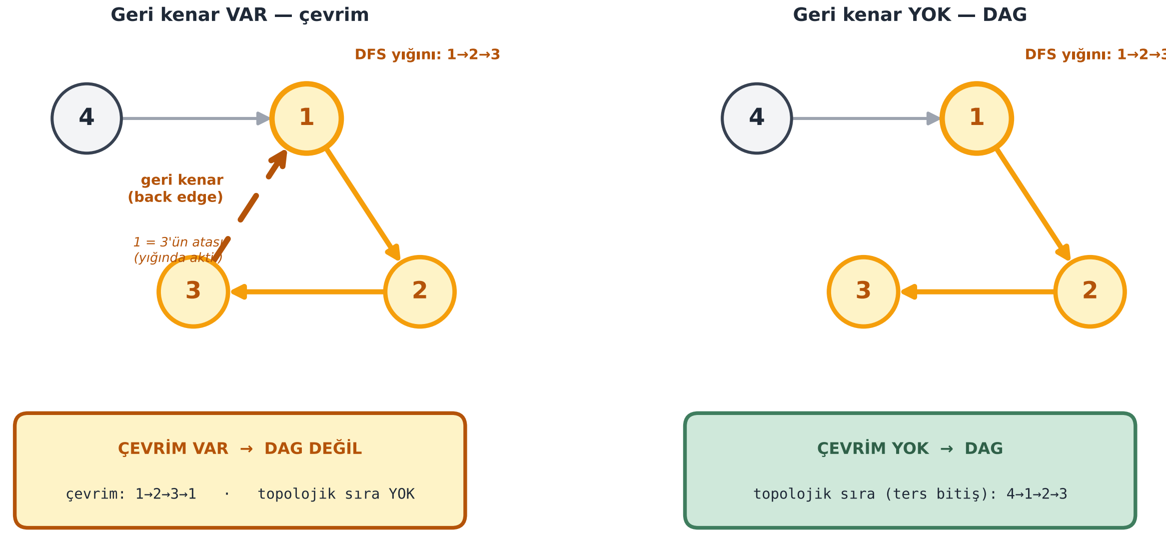
Çevrimi yalnızca tespit değil **bulmak** için: G çevrim içeriyorsa, full DFS bir v düğümünden onun bir **atasına (ancestor)** giden bir kenar gezer (atası = özyineleme ağacında v'den önce gelen düğüm).

> *"full DFS will traverse an edge from a vertex v to some ancestor of v."* — Solomon, 48:20

Kanıt taslağı: çevrim $v_0 \to v_1 \to \ldots \to v_k \to v_0$ olsun; v_0'ı DFS'in ilk gördüğü düğüm seç. v_0'ın özyineleme ağacı tamamlanmadan, ondan erişilebilen $v_1, \ldots, v_k$ de tamamlanır; v_k komşularını gezerken v_0 kenarını görür — bu, v_k'den atası v_0'a giden **geri kenardır (back edge)**. DFS sırasında bu geri kenarı yakaladığın an çevrimi bulmuş olursun.

Şekil 22.6 bu ayrımı iki panelle gösterir: solda çevrimli çizge ($1 \to 2 \to 3 \to 1$ + $4 \to 1$), DFS yığını $1 \to 2 \to 3$ aktifken $3 \to 1$ bir **geri kenardır** (1, 3'ün atası) → çevrim $[1, 2, 3, 1]$, DAG değil; sağda $3 \to 1$ kaldırılmış, geri kenar yok → DAG, topolojik sıra $[4, 1, 2, 3]$.

Şekil 22.5: DAG'da topolojik sıra = TERS bitiş sırası (tek DFS geçişi, O(V+E)) (L10 §8-9, İMZA figür). ÜST-SOL: örnek DAG — A üstte, B ve C ortada yan yana, D altta (elmas); yönlü oklar A→B, A→C, B→D, C→D. ORTA serit: BİTİŞ SIRASI (visit tamamlanınca out() ile eklenir) [D, B, C, A]; D ilk biter (yaprak), A son biter (kök). 'TERSE ÇEVİR' dönüş oku. ALT serit: TOPOLOJİK SIRA = ters bitiş = [A, C, B, D] (amber kutular); her DAG kenarı için f(u)<f(v) onayı yeşil ✓ rozeti (A<C, A<B, B<D, C<D); verify_topological=True. YAN NOT: topolojik sıra BENZERSİZ DEĞİL — [A, C, B, D] ve [A, B, C, D] ikisi de geçerli (B-C arası kenar yok); ters bitiş bunlardan BİRİNİ bulur. Kanıt iki durum (Solomon 37:54). Veri MOTORDAN: build_dag_example={A:[B,C],B:[D],C:[D],D:[]}; finishing_order=[D,B,C,A]; topological_order=[A,C,B,D]; verify=True.

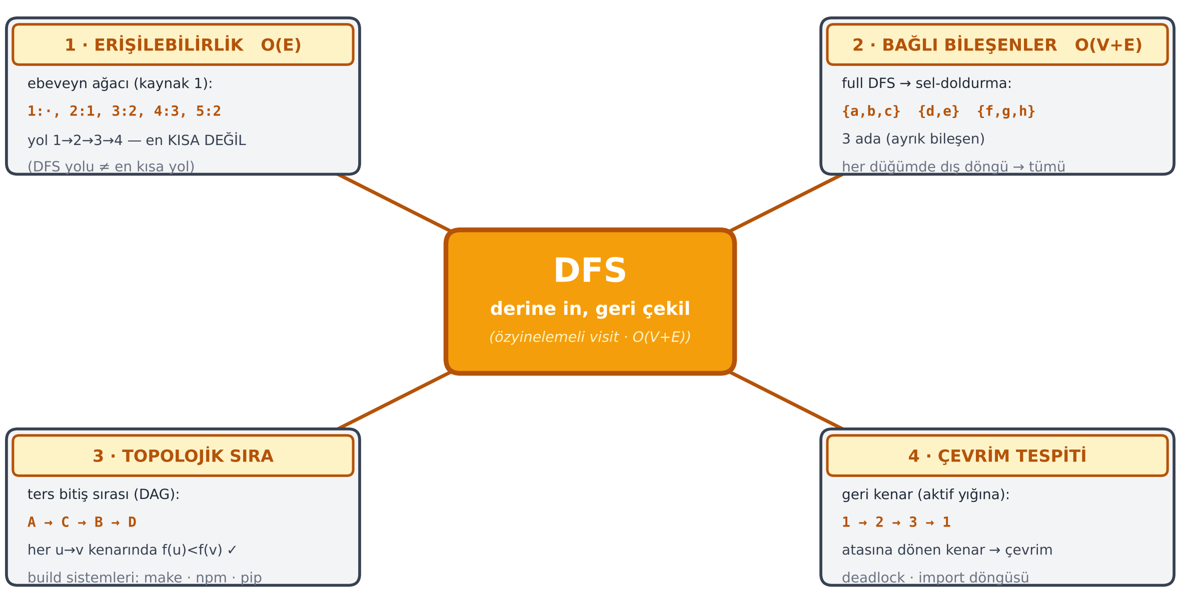


Şekil 22.6: Çevrim tespiti: geri kenar (aktif yığındaki ataya kenar) = çevrim (L10 §10, İMZA figür). SOL panel — çevrimli çizge: 1→2→3→1 üçgeni + 4→1 dış kenarı. DFS özyineleme yolu 1→2→3 amber (aktif yığın). 3'ten 1'e giden kenar KALIN AMBER KESİKLİ + 'geri kenar (back edge)' etiketi — 1, 3'ün ATASIDIR (özyineleme yığınında hâlâ aktif). Sonuç: ÇEVRİM VAR → DAG DEĞİL; çevrim [1,2,3,1], topolojik sıra YOK. SAĞ panel — aynı yapı ama 3→1 kenarı KALDIRILMIŞ (3 çıkışsız): DFS temiz biter, geri kenar yok, çevrim yok → DAG; ters bitiş sırası = topolojik sıra [4,1,2,3]. Full DFS çevrim varsa bir v düğümünden atasına giden kenarı gezer. Veri MOTORDAN: find_cycle({1:[2],2:[3],3:[1],4:[1]})=[1,2,3,1]; find_cycle(3→1 kaldırılmış)=None; topological_order=[4,1,2,3]; verify=True (Solomon 48:20).

22.12 Bu Dersin Özeti

1. **DFS** = özyinelemeli "derine in, geri çekil"; BFS eşmerkezli dairelerken DFS dışa fırlar.
2. **Erişilebilirlik**: ebeveyn ağacı $P(v)$ ile *bir* yol (en kısa değil) ver; ağaç (çevrimsiz).
3. **DFS doğruluğu**: uzaklık k üzerinden tümevarım; u (k) doğruysa v ($k+1$) de doğru.
4. **Çalışma süresi**: DFS $O(E)$ (yalnız erişilebilir kenarlar); BFS $O(V+E)$ (başta $O(V)$ yer).
5. **Bağlı bileşenler**: full DFS, tüm düğümler üzerinde döngü → $O(V+E)$.
6. **DAG + topolojik sıralama**: $f(u) < f(v)$; benzersiz değil; **ters bitiş sırası** = topolojik sıra.
7. **Çevrim tespiti**: topolojik sıra $\iff$ DAG; geri kenar ($v \to$ atası) çevrimi verir.

Şekil 22.7 dersi tek bir sentez figüründe toplar: ortada **DFS** ($O(V+E)$ özyinelemeli iskelet), dört köşede dört güç — erişilebilirlik, bağlı bileşenler, topolojik sıra, çevrim tespiti — her biri motordan gelen bir mini sonuçla.



Şekil 22.7: DFS = tek özyinelemeli iskelet → dört güç (L10 sentezi, İMZA figür). ORTADA büyük amber kutu: DFS — derine in, geri çekil (özyinelemeli visit, O(V+E)). Dört köşede uydu kutusu, merkezden oklarla bağlı, her biri motordan mini sonuç: (1) ERİŞİLEBİLİRLİK O(E) — ebeveyn ağacı (kaynak 1) {1:·, 2:1, 3:2, 4:3, 5:2}, yol 1→2→3→4 en KISA DEĞİL (DFS yolu ≠ en kısa yol). (2) BAĞLI BİLEŞENLER O(V+E) — full DFS sel-doldurma → {a,b,c} {d,e} {f,g,h}, 3 ada. (3) TOPOLOJİK SIRA — ters bitiş sırası (DAG) A→C→B→D, her u→v'de f(u)<f(v) ✓ (build sistemleri make·npm·pip). (4) ÇEVRİM TESPİTİ — geri kenar (aktif yığına) 1→2→3→1, atasına dönen kenar → çevrim (deadlock·import döngüsü). Bağımlılık çözen her sistem içten içe bunu çalıştırır. Veri MOTORDAN: dfs(build_dfs_example(),1)={1:None,2:1,3:2,4:3,5:2}; connected_components=[[a,b,c],[d,e],[f,g,h]]; topological_order(build_dag_example())=[A,C,B,D]; find_cycle({1:[2],2:[3],3:[1]})=[1,2,3,1].

> **!** Tek Bir Cümle
>
> DFS, çizgeyi özyinelemeli olarak derinlemesine gezen tek bir iskelettir; ondan erişilebilirlik ($O(E)$), bağlı bileşenler ve topolojik sıralama/çevrim tespiti — hepsi $O(V+E)$ içinde — doğal olarak türer.

22.13 Kontrol Soruları

> **i** Soru 1: DFS ile BFS arasındaki temel davranış farkı nedir, ve neden DFS O(E) iken BFS O(V+E)?
>
> **Cevap:** BFS, kaynaktan dışa **katman katman** (seviye kümeleri) ilerler — bir seviye bitmeden sonrakine geçmez. DFS, bir dalda **olabildiğince derine** iner, tıkanınca geri çekilir. Süre farkı uygulamadan gelir: bu dersteki BFS başta $O(V)$ yer ayırdığı için (erişilemeyenler dahil) $O(V+E)$'dir; DFS yalnızca kaynaktan **erişilebilen** kenarlara dokunur, hiç erişilemeyen düğümü görmez, dolayısıyla $O(E)$. (full DFS ise tüm düğümler üzerinde döngü kurduğundan yine $O(V+E)$.)

> **i** Soru 2: DFS'in ürettiği ebeveyn ağacı neden en kısa yolu vermez? Bir örnek ver.
>
> **Cevap:** DFS, bir komşuya "ilk denk geldiği" sırada iner — bu sıra en kısa yolu izlemez. Çevrim çizgesi (halka) uç örnektir: kaynaktan komşu komşuyu özyinelemeli çağırınca son düğüme uzun bir zincirin arkasından varılır, oysa tek kenarla doğrudan gidilebilirdi. DFS o kısa kenarı seçmez. Bu yüzden ağırlıksız en kısa yol için **BFS** kullanılır; DFS yapısal sorular (DAG mı, bileşenler, sıra) içindir.

> **i** Soru 3: "Ters bitiş sırası = topolojik sıralama" iddiasında, v'nin u'dan önce ziyaret edildiği durum neden çevrimsizliğe dayanır?
>
> **Cevap:** $u \to v$ kenarı var. v önce ziyaret edildiyse, visit(v) tamamlanmadan u'yu görür mü? Görmesi için v'den u'ya bir yol gerekir — ama $u \to v$ zaten var, $v \to \ldots \to u$ olsaydı bir **çevrim** oluşurdu. Çizge DAG (çevrimsiz) olduğundan bu imkânsız. O hâlde visit(v), u'yu hiç görmeden tamamlanır; ters bitiş sırasında u, v'den önce gelir (Durum 1 ile aynı sonuç). Çevrimsizlik olmasaydı iddia çökerdi.

> **i** Soru 4: Bir çizgenin DAG olup olmadığını ve (varsa) bir çevrimini nasıl bulursun?
>
> **Cevap:** full DFS ile ters bitiş sırasını hesapla; her kenar $f(u) < f(v)$ ilişkisine uyuyorsa çizge bir **DAG**'dır (topolojik sıralama var $\iff$ DAG). Bir kenar bu ilişkiyi bozuyorsa çizge DAG değildir → çevrim vardır. Çevrimi *bulmak* için DFS sırasında bir düğümden **atasına** giden bir kenar (geri kenar, back edge) ara; o geri kenar bulunduğu an, atadan o düğüme inen özyineleme zinciri + geri kenar bir çevrim oluşturur.

22.14 Egzersizler

Egzersiz 1. Verilen yönlü bir çizgede, belirli bir kaynaktan DFS gezinme sırasını ve ebeveyn ağacı P'yi elle çıkar; aynı çizgede BFS sırasıyla karşılaştır.

Egzersiz 2. Python'da özyinelemeli DFS'i (yukarıdaki `dfs`) yığın (stack) tabanlı **özyinelemesiz** bir sürüme dönüştür; çıktının ebeveyn ağacının aynı kaldığını doğrula.

Egzersiz 3. full DFS ile bir yönsüz çizgenin bağlı bileşen sayısını hesaplayan bir fonksiyon yaz; süresinin $O(V + E)$ olduğunu argümanla göster.

Egzersiz 4. Küçük bir DAG için tüm geçerli topolojik sıralamaları elle listele; ters bitiş sırasının bunlardan biri olduğunu doğrula.

Egzersiz 5. DFS'i, bir geri kenar (atası ziyarette olan bir komşu) bulduğunda çevrimi düğüm listesi olarak döndürecek şekilde genişlet.

22.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 16 (L11) — Ağırlıklı En Kısa Yollar

Ders 16 (L11): Ağırlıklı En Kısa Yollar — Jason Ku ile, kenarlara **ağırlık** ekliyoruz: artık "en az kenar" değil, "en az toplam ağırlık" arıyoruz. BFS/DFS yetmez; ağırlıklar (hatta negatif ağırlıklar) yeni problemler doğurur. Bu, Bellman-Ford ve Dijkstra'ya giden kapının açılışıdır.
Ders 16 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 3 (bağlı bileşenler) ve 4 (topolojik sıralama) çöz.
- DFS doğruluk kanıtını (uzaklık k tümevarımı) ezberden anlat.
- Ana cümleyi tekrar oku: *"DFS tek iskelet; erişilebilirlik, bileşenler, topolojik sıra ondan türer."*

22.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
DFS	Özyinelemeli "derine in, geri çekil"; ebeveyn ağacı $P(v)$	Böl. 1, 3
Erişilebilirlik	s'den ulaşılan düğümler; P ile bir yol (en kısa değil)	Böl. 2
DFS süresi	$O(E)$ (yalnız erişilebilir kenarlar); full DFS $O(V + E)$	Böl. 5, 7
Bağlı bileşen	Yönsüz çizgede birbirinden erişilebilir küme; full DFS	Böl. 7
DAG	Yönlü + çevrimsiz çizge	Böl. 8
Topolojik sıra	$u \to v \Rightarrow f(u) < f(v)$; benzersiz değil	Böl. 8
Bitiş sırası	visit tamamlanınca ekle; tersi = topolojik sıra	Böl. 9
Çevrim tespiti	geri kenar ($v \to$ atası); topolojik sıra $\iff$ DAG	Böl. 10

22.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, “tek özyinelemeli iskelet (DFS), dört güç” sezgisini kurar — köprülerin özeti:

1. **Topolojik sıralama** → build sistemleri (`make`, Bazel), paket bağımlılığı (npm/pip), görev zamanlama: DAG’da çalıştırma sırası.
2. **Çevrim tespiti** → deadlock, import döngüsü, elektronik tablo formül döngüsü, derleyici bağımlılık grafı.
3. **Bağlı bileşenler** → kümeleme, sosyal ağ grupları, görüntü “flood fill”, birlik-bul (union-find) alternatifi.
4. **DFS özyineleme** → ağaç/çizge gezme, geri izleme (backtracking) algoritmaları, ifade ağacı değerlendirme.
5. **Erişilebilirlik** → ulaşılabilir kod analizi (derleyici), çöp toplama (mark-and-sweep), bağımlılık kapanışı.
6. $O(V + E)$ **doğrusal** → seyreklik bilinci: gerçek çizgeler seyrek; bir gezinmede her kenara bir kez dokun.

! Tek bir şey alıp gideceksen

DFS, “derine in, geri çekil” diyen tek bir özyinelemeli iskelettir — ama o tek iskeletten erişilebilirlik ($O(E)$), bağlı bileşenler ($O(V + E)$), topolojik sıralama ve çevrim tespiti çıkar. Bağımlılık çözen her sistem (make, paket yöneticisi, derleyici) içten içe bunu çalıştırır.

23 Ağırlıklı En Kısa Yollar

Kenarlara tamsayı ağırlık w: E → ℤ verince en kısa yol = toplam ağırlığın minimumu; iki yeni tuzak (yol yok → +∞, negatif çevrim erişilir → −∞); mesafe yeter (ebeveyn işaretçileri δ'dan O(V+E)'de kurulur); bir DAG'da SSSP'yi ∞'dan başlayan tahminleri topolojik sırada üçgen eşitsizliğine göre güvenle gevşeterek O(V+E)'de çöz; algoritma haritası DAG / Bellman-Ford / Dijkstra

i Bölüm bilgisi

- **Ku'nun videosu:** YouTube — Lecture 11: Weighted Shortest Paths (≈58 dk)
- **OCW sayfası:** MIT 6.006 Lecture 11: Weighted Shortest Paths
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 16 (L11)
- **Hoca:** Jason Ku (ağırlıklı en kısa yollar ünitesinin açılışı)
- **Okuma süresi:** ≈26 dk

Bir önceki ders BFS/DFS ile uzaklığı **kenar sayısıyla** ölçtü; bu ders her kenara bir tamsayı **ağırlık** verir ve "en az kenar" yerine "en az toplam ağırlık" arar. Bu, sonraki dört dersin (DAG relaxation, Bellman-Ford, Dijkstra, Johnson) temelidir.

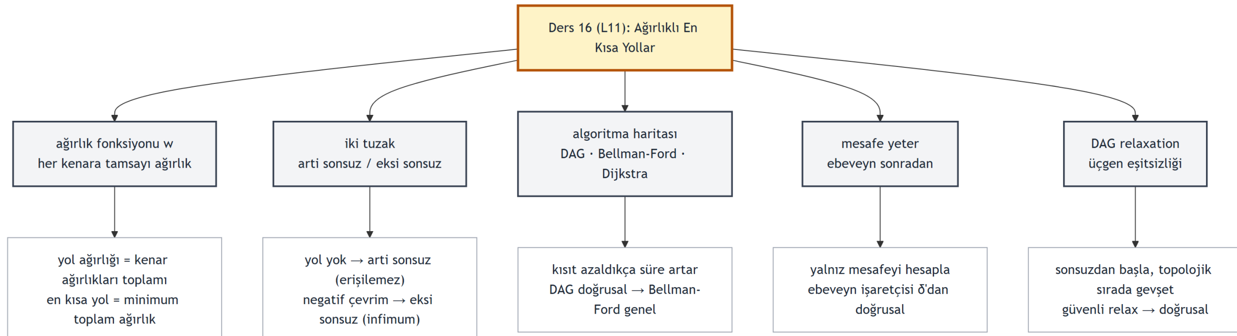
23.1 Bu Derste Ne Var?

Şimdiye dek uzaklığı **kenar sayısıyla** ölçtük (BFS/DFS). Bugün Jason Ku ile **ağırlıklı en kısa yollar** ünitesini açıyoruz: her kenara bir tamsayı **ağırlık** veririz ve iki düğüm arasında "en az kenar" yerine "en az toplam ağırlık" ararız. Bu genelleştirme, sonraki dört dersin (DAG relaxation, Bellman-Ford, Dijkstra, Johnson) temelidir.

> *"instead of… the number of edges in a path… we're going to count an integer associated with that edge. It's going to be called a weight."* — Ku, 3:33

Üç ana fikir bu derste yan yana gelir:

1. **Ağırlıklı en kısa yol** $\delta(s,t)$ — yol ağırlıklarının minimumu; iki yeni tuzak: yol yoksa $+\infty$, **negatif ağırlıklı çevrim** erişilirse $-\infty$.
2. **Mesafe yeter** — yalnızca δ mesafelerini hesapla; ebeveyn işaretçilerini sonradan doğrusal zamanda kur.
3. **DAG relaxation** — bir DAG'da, topolojik sırada **kenar gevşetme (relax)** ile SSSP'yi $O(V+E)$'de çöz.



Şekil 23.1: Ders 16'nın (L11) kavram haritası: kök = ağırlıklı en kısa yollar (Ku). Beş dal — (1) ağırlık fonksiyonu w: her kenara tamsayı ağırlık (pozitif, negatif ya da sıfır); yol ağırlığı = kenar ağırlıklarının toplamı; en kısa yol = minimum toplam ağırlık. (2) iki tuzak: yol yoksa arti sonsuz (erişilemez); negatif ağırlıklı çevrim erişilirse eksi sonsuz (minimum yok, infimum). (3) algoritma haritası: kısıt azaldıkça süre doğrusaldan uzaklaşır — DAG relaxation doğrusal, Bellman-Ford genel, Dijkstra negatif yok. (4) mesafe yeter: yalnız mesafeyi hesapla; ebeveyn işaretçileri sonradan doğrusal zamanda kurulur. (5) DAG relaxation: tahminleri sonsuzdan başlat, topolojik sırada üçgen eşitsizliğine göre güvenle gevşet, doğrusalda bitir. Sonuç: tek gevşetme tekniği, dört dersin temeli.

Builder Notu — Ağırlık Girince Dünya Değişir

BFS/DFS, "en az kenar" sorusunu doğrusal zamanda çözdü. Kenarlara ağırlık verince soru değişir: artık kenar saymıyoruz, **toplam ağırlığı** topluyoruz — ve iki yeni tuzak doğuyor (yol yok, negatif çevrim). Önce neyi sorduğumuzu (ağırlıklı en kısa yol), sonra nasıl çözdüğümüzü (DAG relaxation) ele alırız.

- **İleriye → GPS/yönlendirme:** ağırlıklı en kısa yol, Google Maps'in özüdür (kenar = yol, ağırlık = süre/mesafe); ağ gecikmesi (latency) yönlendirmesi de aynı.
- **İleriye → kritik yol/zamanlama:** DAG relaxation, proje zamanlama (PERT/CPM) ve build sistemlerinde "en uzun/en kısa yol" hesabıdır.
- **İleriye → sezgisel arama:** üçgen eşitsizliği (triangle inequality), sezgisel-yönlü `A*` aramasının teorik dayanağıdır.
- **Geriye → BFS (Ders 13):** ağırlıksız en kısa yolu BFS verdi; bu ders onu *genelleştirir* — BFS, tüm ağırlıklar eşit olan özel durumdur.

Tek cümle: *Kenarlara ağırlık verince "en kısa yol" toplam ağırlığın minimumu olur; bir DAG'da bunu, mesafe tahminlerini topolojik sırada üçgen eşitsizliğine göre gevşeterek $O(V+E)$'de çözeriz.*

23.2 1. Ağırlıksızdan Ağırlığa: Neden ve Nasıl

Son iki ders BFS/DFS ile ağırlıksız problemleri (SSSP, erişilebilirlik, bağlı bileşenler, topolojik sıralama) doğrusal zamanda çözdü. Şimdi genelleştiriyoruz: her kenara bir **tamsayı ağırlık** atayan bir **ağırlık fonksiyonu** $w : E \to \mathbb{Z}$. Ağırlıklar pozitif, negatif veya sıfır olabilir.

Neden? Gerçek uygulamalar: yol ağında mesafe/süre (Vassar Street ile Amherst arası kısa, köprü uzun), ağ

gecikmesi, sosyal ağda ilişki gücü (hatta "frenemy" için negatif).

23.3 2. Ağırlık Gösterimi

Çizgeyi komşuluk listesiyle saklıyorduk; ağırlığı iki yolla ekleriz:

- **Komşulukla birlikte:** her komşuyu ağırlığıyla bir **ikili (tuple)** olarak sakla.
- **Ayrı sözlük:** kenarları ağırlıklarına eşleyen ayrı bir hash tablosu (edge → weight).

Tek varsayım: bir kenarın ağırlığı $O(1)$'de sorgulanabilir (hash tablosu veya hash-tablosu-of-hash-tablosu).

23.4 3. Ağırlıklı Yol ve En Kısa Yol

Bir **yolun ağırlığı** $w(\pi)$, yoldaki kenarların ağırlıklarının **toplamıdır**.

> *"the weight of a path… is just going to be the sum of the weights in the edges in the path."* — Ku, 11:08

Örneğin §4'te kullanacağımız çizgede $a \to b \to f \to g = -5 + (-4) + 2 = -7$. **Ağırlıklı en kısa yol**, iki düğüm arasında **minimum ağırlıklı** yoldur.

> *"a shortest path… is a minimum weight path from s to t."* — Ku, 13:29

Mesafeyi $\delta(s, t)$ ile gösteririz: s'den t'ye tüm yolların ağırlık minimumu. (Uyarı: ağırlıklı yol bir kenarı birden çok kez kullanabilir — *basit yol* değilse; en kısa yolların bunu yapmadığını sonra göreceğiz.)

Şekil 23.2 ağırlıklı bir DAG üzerinde yol ağırlığı fikrini gösterir: her kenarda bir ağırlık rozeti var (negatif $g \to d = -2$ amber vurgulu), vurgulu yol $\pi = e \to f \to g \to d$ amber kalın, altta toplam kutusu $w(\pi) = 3 + 2 + (-2) = 3$ — kenar ağırlıklarının toplamı (kenar sayısı değil).

23.5 4. İki Tuzak: +∞ ve −∞

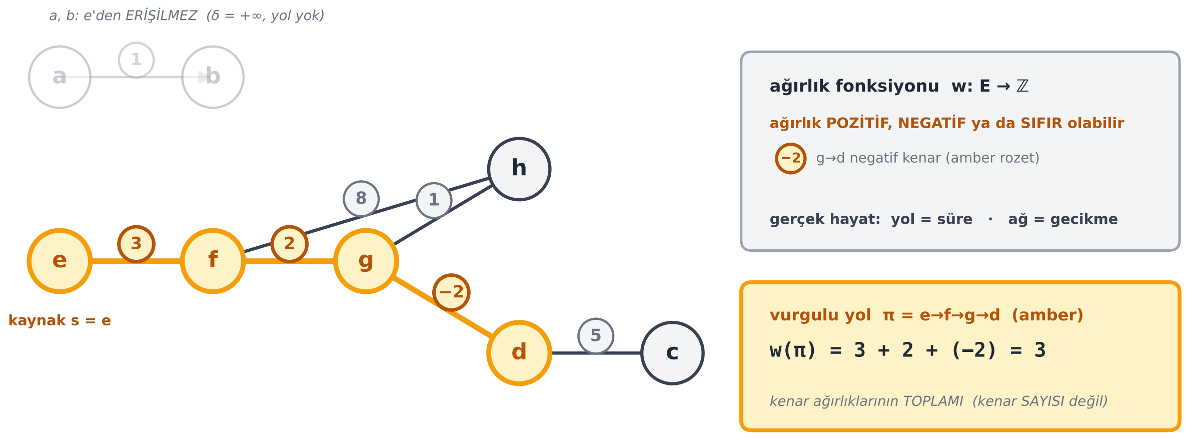
δ tanımında iki şey ters gidebilir:

1. **Yol yoksa:** $\delta(s, t) = +\infty$ (uzlaşım, BFS'teki gibi).
2. **Sonlu en kısa yol yoksa:** kenarları döne döne **giderek daha kısa** bir yol bulunabiliyorsa δ tanımsızdır; bu durumda $\delta = -\infty$ (minimum değil, *infimum*).

> *"It's possible that a finite length shortest path doesn't exist."* — Ku, 15:26

Çalışılan Örnek — negatif ağırlıklı çevrim. Bu, **negatif ağırlıklı çevrim** olduğunda olur. Örnek çizgede $b \to f \to g \to c \to b = (-4) + 2 + 1 + (-1) = -2$. b'ye -5 ağırlıklı kenarla varıp bu çevrimde her tur -2 kazanırsak, sonsuza dek küçülürüz → sonlu en kısa yol yok.

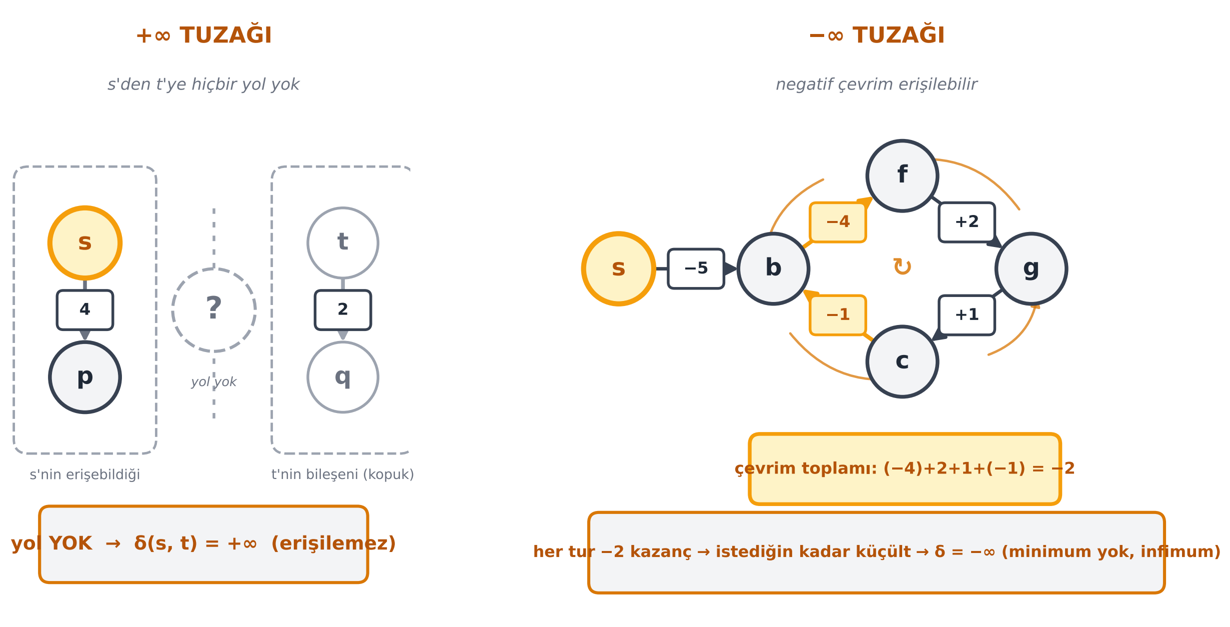
> *"we could have negative weight cycles."* — Ku, 18:46



Şekil 23.2: Ağırlıklı çizge: w: E → ℤ ve yol ağırlığı = kenar ağırlıklarının TOPLAMI (L11 §1-3). Örnek ağırlıklı DAG (kaynak e): kenarlar e→f(3), f→h(8), f→g(2), g→h(1), g→d(−2 AMBER negatif), d→c(5); a, b ayrık üst köşe (e'den ERİŞİLMEZ → δ=+∞, soluk çizilir). Vurgulu yol π = e→f→g→d amber kalın; sağ alt toplam kutusu w(π) = 3 + 2 + (−2) = 3 (kenar ağırlıklarının TOPLAMI, kenar SAYISI değil). Yan kutu: ağırlık fonksiyonu w: E → ℤ — pozitif, negatif ya da sıfır olabilir (gerçek hayat: yol = süre, ağ = gecikme). Alt not: en kısa yol δ(s,t) = MİNİMUM toplam ağırlıklı yol (artık en az kenar değil; negatif kenar daha uzun bir yolu kısaltabilir). Veri MOTORDAN: path_weight(w, [e,f,g,d]) = 3 + 2 + (−2) = 3 (Ku 11:08, 13:29).

Kural: s'den v'ye giden bir yol bir **negatif ağırlıklı çevrim üzerindeki** düğümden geçiyorsa, $\delta(s, v) = -\infty$. Bu durumda ebeveyn işaretçisiyle uğraşmayız (sonlu yol yok); bunun yerine bir negatif çevrim döndürmek isteyebiliriz (bu, sonraki dersin Bellman-Ford konusudur).

Şekil 23.3 iki tuzağı yan yana koyar: solda $+\infty$ (iki ayrık küme, geçiş yok), sağda $-\infty$ (negatif çevrim $b \to f \to g \to c \to b$ erişilir, her tur -2 kazanç).



Şekil 23.3: Ağırlıklı en kısa yolun iki tuzağı: +∞ (erişilemez) ve –∞ (negatif çevrim) (L11 §4, İMZA figür). SOL panel +∞ TUZAĞI: iki ayrık küme — kaynak s'nin erişebildiği küme (s→p, ağırlık 4) ve t'nin kopuk bileşeni (t→q); aralarında kesikli '?' engeli, geçiş yok → δ(s,t) = +∞ (erişilemez). SAĞ panel –∞ TUZAĞI: kaynak s'den çevrime –5 ile giriş (b'ye); çevrim b→f→g→c→b elmas yerleşimle, kenar ağırlıkları MOTORDAN: b→f(–4 amber), f→g(2), g→c(1), c→b(–1 amber); çevrim çevresinde dönen kavisli oklar. Çevrim toplamı kutusu (–4)+2+1+(–1) = –2 (AMBER): her tur –2 kazanç → istediğin kadar küçült → δ = –∞ (minimum yok, infimum). Alt şerit: +∞ = erişilemez, –∞ = negatif çevrim erişilir (sonlu en kısa yol YOK). Veri MOTORDAN: build_negative_cycle_example çevrim [b,f,g,c,b], path_weight = –2 (Ku 15:26, 18:46).

23.6 5. BFS'e İndirgenebilen Özel Durumlar

Ağırlıklı SSSP'nin bazı özel halleri **BFS**'e indirgenir:

- **Tüm ağırlıklar = 1:** zaten ağırlıksız çizge — BFS doğrudan çözer.
- **Tüm ağırlıklar = aynı pozitif değer c:** c'ye böl → ağırlıksız çöz → sonuçları c ile çarp.
- **Pozitif ağırlıkları seri kenarlara aç:** ağırlık 4'lük bir kenar, 4 ardışık (seri) ağırlık-1 kenara dönüştürülür. Tüm ağırlıklar toplamı asimptotik olarak $V + E$'den küçükse, bu dönüşüm + BFS yine **doğrusal**.

Genel ağırlıklı SSSP'yi doğrusal zamanda çözmek **açık problemdir** (bilinmiyor).

23.7 6. Genel Manzara: DAG / Bellman-Ford / Dijkstra

Üç algoritma, üç kısıt seviyesi:

- **DAG relaxation** (bu ders): çizge bir **DAG** ise — ağırlıklar *herhangi bir* tamsayı olabilir (negatif dahil, çevrim olmadığından sorun yok) — $O(V + E)$ doğrusal.
- **Bellman-Ford** (Ders 18, L12): *herhangi* çizge (çevrim, negatif çevrim dahil); $\sim O(V \cdot E)$ (karesel benzeri); pratik ve yaygın.
- **Dijkstra** (Ders 19, L13): ağırlıklar **negatif değilse**; $\sim O(V \log V + E)$ (doğrusala yakın, V'de bir log faktörü). Çoğu gerçek problem (yol ağı) negatif olmadığından Dijkstra en sık kullanılandır.

Şekil 23.4 bu üç algoritmayı kısıt seviyesine göre dizer ve çalışılan DAG örneğinde δ değerlerini gösterir (motor kanıtı: $e = 0, f = 3, g = 5, h = 6, d = 3, c = 8$).

23.8 7. En Kısa Yol Ağacı: Mesafeden Ebeveyn

BFS gibi iki şey döndürürüz: mesafeler δ ve ebeveyn işaretçileri (en kısa yol ağacı). Ama ikisini de her algoritmada taşımak angaryadır. **İddia:** yalnızca mesafeler verilirse, ebeveyn işaretçileri **doğrusal zamanda** geri kurulabilir.

> *"distances are actually sufficient for us to reconstruct parent pointers if we need them later."* — Ku, 28:57

Algoritma (yalnız δ sonlu olan v'ler için): her u için, her $v \in \text{Adj}^+(u)$: v'ye henüz ebeveyn atanmamışsa **ve** $\delta(s, v) = \delta(s, u) + w(u, v)$ ise (bu kenar bir en kısa yolun parçası), $P(v) = u$ ata. Tüm düğümler + komşular üzerinde bir geçiş → $O(V + E)$. Bu yüzden bundan sonra yalnızca **mesafeleri** hesaplamaya odaklanırız.

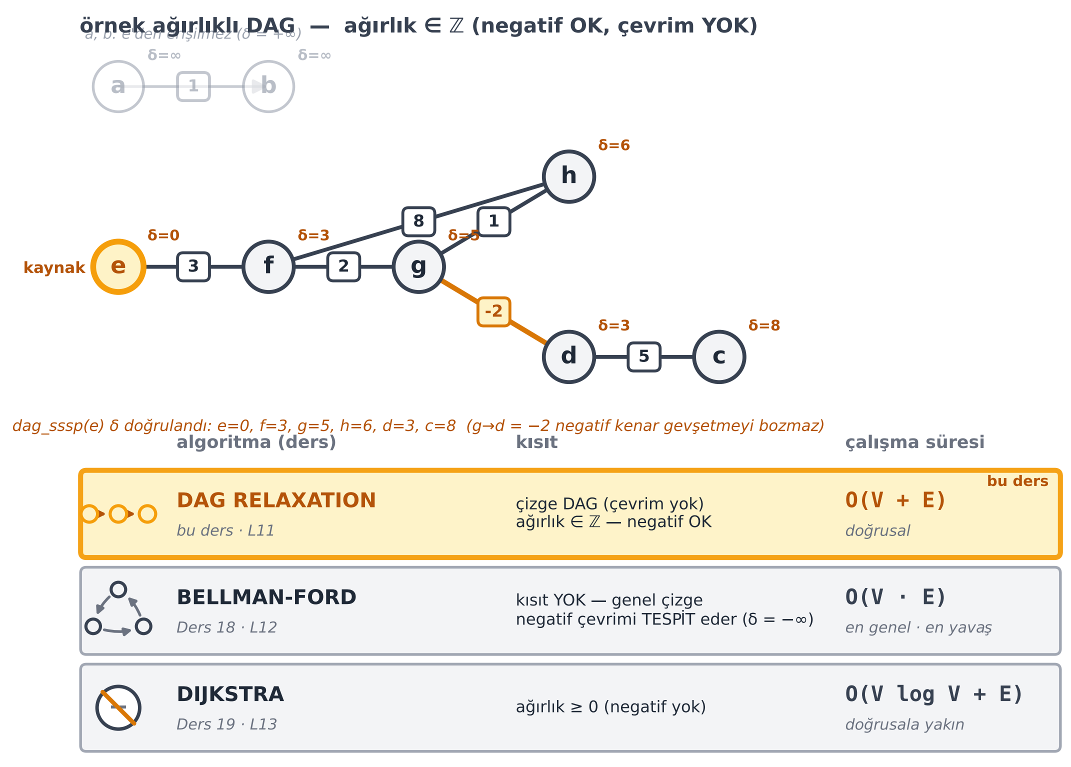
Şekil 23.5 bu fikri üç blokta gösterir: solda yalnız δ tablosu (figürün tek girdisi), ortada kontrol kuralı ($\delta(h) = 6 = \delta(g) + 1$ ✓, $\delta(h) = 6 \neq \delta(f) + 8 = 11$ ×), sağda kurulan en kısa yol ağacı (ebeveyn kenarları kalın).

23.9 8. DAG Relaxation: Mesafe Tahminleri ve Üçgen Eşitsizliği

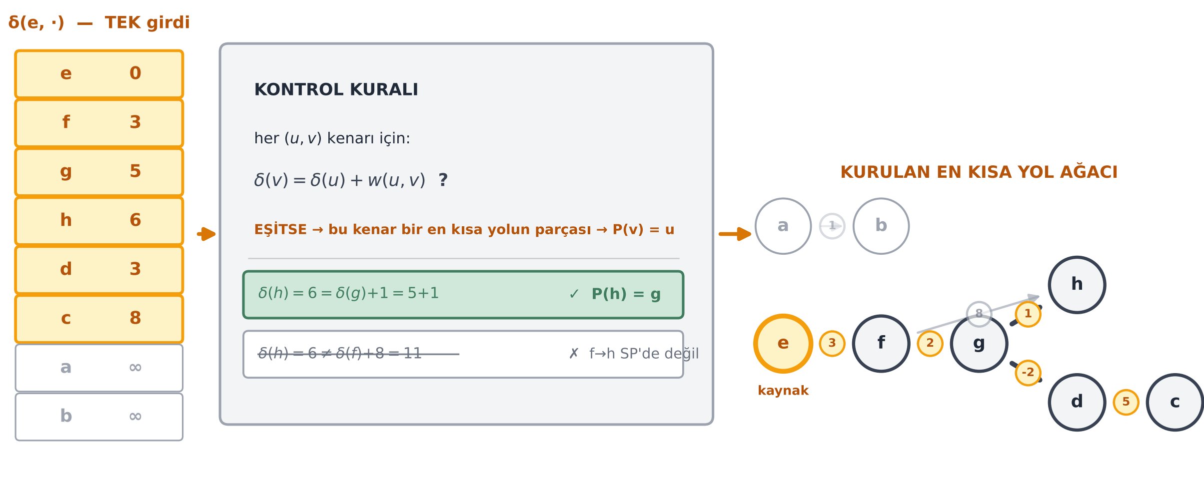
Fikir: her v için bir **mesafe tahmini** $d(s, v)$ tut; başta $+\infty$ (kaynak için 0). Değişmez: tahmin daima δ'yı **yukarıdan sınırlar** (üst sınır); bilgi geldikçe **kademeli olarak düşür**.

> *"we're going to maintain that they upper-bound this thing and gradually lower until they're equal."* — Ku, 38:20

Ne zaman düşürürüz? Tahmin **üçgen eşitsizliğini (triangle inequality)** ihlal ettiğinde. Üçgen eşitsizliği: herhangi bir x için $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$ (yolları x'ten geçenlerle kısıtlamak minimumu küçültemez).



Şekil 23.4: Ağırlıklı SSSP: üç algoritma, üç kısıt seviyesi — kısıt azaldıkça süre doğrusaldan uzaklaşır (L11 §6, İMZA figür). ÜST: örnek ağırlıklı DAG (kaynak e); her düğümün yanında motordan δ değeri (e=0, f=3, g=5, h=6, d=3, c=8; a,b = +∞ erişilmez, soluk). g→d = −2 negatif kenar AMBER (gevşetmeyi bozmaz). ALT: 3 satırlık karşılaştırma panosu, kısıt azaldıkça süre artar. Satır 1 DAG RELAXATION (bu ders L11, AMBER çerçeve): kısıt çizge DAG + ağırlık ∈ ℤ negatif OK → O(V+E) doğrusal. Satır 2 BELLMAN-FORD (Ders 18 L12): kısıt YOK, genel çizge, negatif çevrimi TESPİT eder (δ=−∞) → O(V·E) en genel/en yavaş. Satır 3 DIJKSTRA (Ders 19 L13): kısıt ağırlık ≥ 0 → O(V log V + E) doğrusala yakın. Alt not: genel SSSP'yi doğrusal zamanda çözmek hâlâ AÇIK PROBLEM; çoğu gerçek problem (yol ağı, mesafe) negatif içermez → Dijkstra en yaygın. Veri MOTORDAN: dag_sssp(e) = {e:0,f:3,g:5,h:6,d:3,c:8,a:∞,b:∞}.



Şekil 23.5: Mesafe YETER: en kısa yol ağacı yalnızca δ tablosundan kurulur (L11 §7, İMZA figür). SOL blok: yalnız δ(e,·) tablosu (figürün TEK girdisi) — e:0, f:3, g:5, h:6, d:3, c:8; a, b ∞ (erişilmez köşeler soluk). ORTA blok: kontrol kuralı — her (u,v) kenarı için δ(v) = δ(u)+w(u,v) mi? EŞİTSE bu kenar bir en kısa yolun parçası → P(v)=u. İki örnek: δ(h)=6 = δ(g)+1 = 5+1 ✓ (P(h)=g, yeşil onay); δ(h)=6 ≠ δ(f)+8 = 11 × (f→h en kısa yolda DEĞİL, üstü çizili). SAĞ blok: kurulan en kısa yol ağacı — ebeveyn kenarları KALIN slate ok (f←e, g←f, h←g, d←g, c←d); en kısa yola katılmayan kenar (f→h) soluk; erişilmez a→b soluk. Alt not: tek geçiş O(V+E) — bu yüzden TÜM en kısa yol algoritmaları yalnız MESAFEYE odaklanır; ebeveynler δ'dan ücretsizce kurulur. Veri MOTORDAN: parents_from_distances → f←e, g←f, h←g, d←g, c←d (Ku 28:57).

> *"the shortest path distance from u to v can't be bigger than… from u to x plus… from x to v."* — Ku, 41:07

Kenar gevşetme (relax). Bir (u, v) kenarı için $d(s, v) > d(s, u) + w(u, v)$ ise (yani u üzerinden v'ye gitmek mevcut tahminden iyiyse), tahmini düşür: $d(s, v) \leftarrow d(s, u) + w(u, v)$. "Relax" tarihsel bir terimdir; ihlal edilen bir kısıtı *yerel olarak* çözer (başka yerde yeni ihlal doğabilir, ama bu kısıt artık sağlanır).

23.10 9. Relax Güvenlidir

Gevşetme **güvenlidir (safe):** her mesafe tahmini $d(s, v)$ daima ya $+\infty$'dur ya da s'den v'ye **gerçek bir yolun ağırlığıdır** — asla "uydurma" bir değer değil.

> *"each distance estimate s, v is weight of some path from s to v or infinite."* — Ku, 45:17

Kanıt (kısa): (u, v)'yi gevşetirken $d(s, v) \leftarrow d(s, u) + w(u, v)$. Varsayımla $d(s, u)$ zaten s'den u'ya bir yolun ağırlığıydı; ona $u \to v$ kenarını eklersek sonuç yine s'den v'ye bir yolun ağırlığıdır. Değişmez korunur. Bu yüzden tahmin hiçbir zaman gerçek en kısa mesafenin altına inmez — yalnızca ona yaklaşır.

Şekil 23.6 bu iki temeli yan yana koyar: solda üçgen eşitsizliği $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$, sağda relax öncesi/sonrası — $d(v) = 12$ üstü çizili, $d(u) + w = 5 + 4 = 9$ amber yeni değer.

23.11 10. DAG Relaxation Algoritması ve Doğruluğu

Algoritma: tüm $d(s, v) = +\infty$; $d(s, s) = 0$; sonra düğümleri **topolojik sırada** işle, her u için tüm çıkış komşularını gevşet.

> *"we're going to process each vertex u in a topological sort order."* — Ku, 48:30

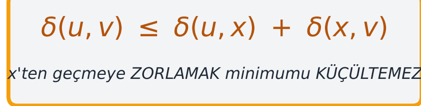
```python
def dag_relaxation(adj, weight, s, topo_order):
    d = {v: float('inf') for v in adj}   # tahminler
    d[s] = 0
    for u in topo_order:                 # topolojik sirada
        for v in adj[u]:                 # cikis komsulari
            if d[u] + weight[(u, v)] < d[v]:   # ucgen esitsizligi ihlali
                d[v] = d[u] + weight[(u, v)]    # relax (guvenli)
    return d
```

Çalışılan Örnek — e'den en kısa yollar. Topolojik sıra a, b, e, f, g, h, d, c. a ve b kaynaktan (e) önce geldiğinden ∞ kalır (e'den onlara yol yok). $e = 0$. $e \to f$ kenarı (ağırlık 3): $d(f) = 3$. f'yi işle: $3 + 8 = 11$, $3 + 2 = 5$. g'yi işle: $5 + 1 = 6$ (11'i 6 ile değiştir), $5 + (-2) = 3$. Devam… sonuçta δ: $a = b = +\infty$, $e = 0$, $f = 3$, $g = 5$, $h = 6$, $d = 3$, $c = 8$.

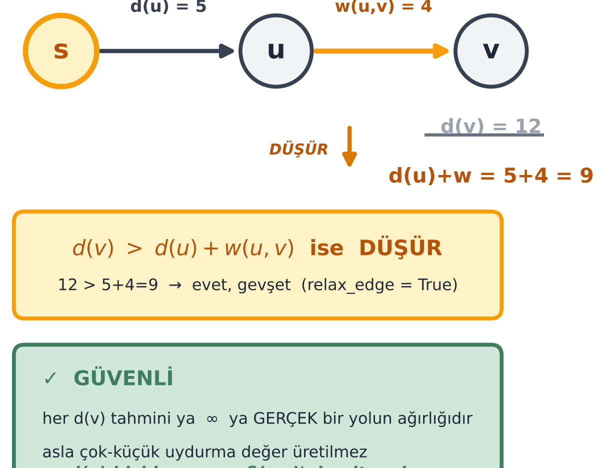
Doğruluk (topolojik tümevarım). Topolojik sıradaki k. düğüm v; ondan öncekiler doğru (tümevarım varsayımı). s'den v'ye en kısa yolda v'den hemen önceki u, topolojik sırada v'den önce gelmek zorundadır (yoksa DAG değil) → u doğru hesaplanmıştır; u işlenirken (u, v) kenarı gevşetildiğinden $d(v)$ doğru en kısa mesafeye iner. Her düğüm + komşuları bir kez → $O(V + E)$ doğrusal.

Gevşetme (relax) GÜVENLİDİR — üçgen eşitsizliği + tahmin asla δ'nın altına inmez

ÜÇGEN EŞİTSİZLİĞİ (§8)



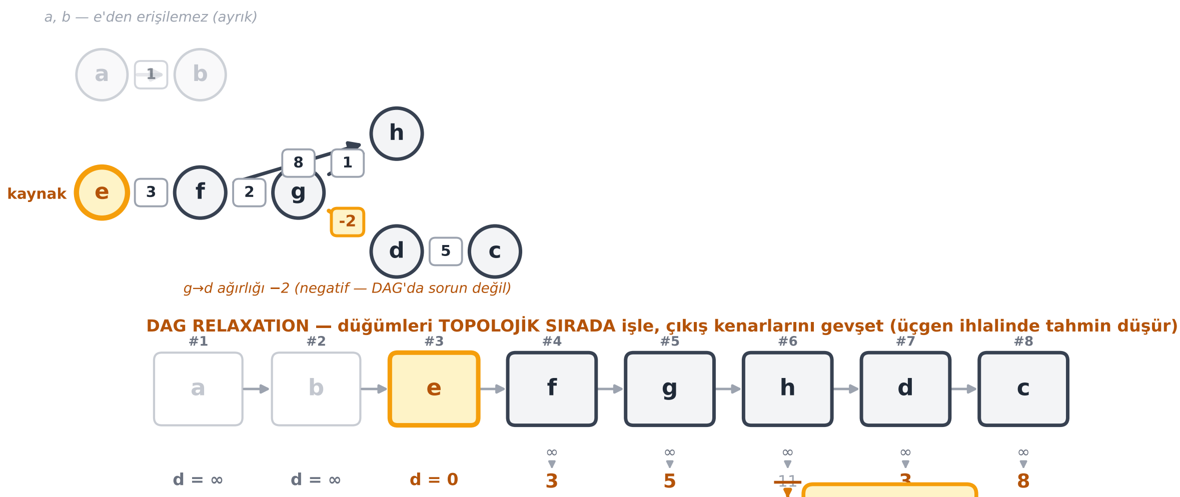
RELAX GÜVENLİ (§9)



"relax" tarihsel terim: ihlal edilen kısıtı yerel olarak çözer (gergin bir tahmini gerçeğe doğru gevşetir) · (Ku 41:07 + 45:17)

Şekil 23.6: Gevşetme (relax) GÜVENLİDİR — üçgen eşitsizliği + tahmin asla δ'nın altına inmez (L11 §8-9, İMZA figür). SOL panel ÜÇGEN EŞİTSİZLİĞİ (§8): üç düğüm u, x, v üçgen yerleşim; alt düz kenar = doğrudan en kısa yol δ(u,v), üstte iki kenarlı kırık yol δ(u,x) + δ(x,v) kesikli. Eşitsizlik kutusu δ(u,v) ≤ δ(u,x) + δ(x,v); not: x'ten geçmeye ZORLAMAK minimumu KÜÇÜLTEMEZ. SAĞ panel RELAX (§9, öncesi/sonrası): mini durum s →(d(u)=5) u →(w(u,v)=4) v; eski d(v)=12 üstü çizili (soluk), yeni d(u)+w = 5+4 = 9 amber. Koşul kutusu d(v) > d(u)+w(u,v) ise DÜŞÜR (12 > 9 → evet, relax_edge = True). GÜVENLİ rozeti (yeşil): her d(v) tahmini ya ∞ ya GERÇEK bir yolun ağırlığıdır → asla çok-küçük uydurma değer üretilmez → d(v) hiçbir zaman δ(s,v)'nin altına inmez. Alt not: 'relax' tarihsel terim, ihlal edilen kısıtı yerel çözer. Veri MOTORDAN: relax_edge(d={s:0,u:5,v:12}, w(u,v)=4) → d(v) 12→9 (5+4) True; ikinci çağrı False (Ku 41:07 + 45:17).

Şekil 23.7 algoritmayı tek bir akışta gösterir: üstte ağırlıklı DAG, altta topolojik sıra şeridi (a, b, e, f, g, h, d, c) ve her düğümün altında d-tahmini evrimi — h'nin $\infty \to 11 \to 6$ geçişi imza an (g işlenirken daha iyi yol bulunur).



Şekil 23.7: DAG relaxation: ∞'dan başla, topolojik sırada gevşet — h ∞→11→6 (g daha iyi yol bulur) (L11 §10, İMZA figür). ÜST: örnek ağırlıklı DAG (kaynak e); kenar ağırlıkları rozette, g→d(−2) negatif amber (DAG'da sorun değil); a, b ayrık üst köşe (e'den erişilmez, soluk). ALT: topolojik sıra işleme şeridi a, b, e, f, g, h, d, c (soldan sağa, #1..#8 rozeti). a, b soluk + 'd = ∞ kaldı' (erişilmez). e = 0 (kaynak). Her düğümün altında d-tahmini evrimi mini sütun: f ∞→3; g ∞→5; h ∞→11→6 (11 üstü çizili, 6 amber — KLİMAKS: g işlenirken daha iyi yol! d[g]+w(g,h)=5+1=6 < 11); d ∞→3; c ∞→8. Alt not: topolojik sıra garantisi — bir düğüm işlenirken TÜM öncülleri zaten DOĞRU (tümevarım), tek geçiş yeter → O(V+E); negatif ağırlık (g→d=−2) sorun değil, çevrim olmadığından her kenar tek kez gevşetilir, −∞ tuzağı yok. Veri MOTORDAN: topo = [a,b,e,f,g,h,d,c]; dag_relaxation(e) = {a:∞,b:∞,e:0,f:3,g:5,h:6,d:3,c:8}; g adımı relaxed = [(h,11,6),(d,∞,3)] (Ku 48:30).

23.12 Bu Dersin Özeti

1. **Ağırlık** $w : E \to \mathbb{Z}$; **yol ağırlığı** = kenar ağırlıkları toplamı; **en kısa yol** = minimum ağırlık.
2. **İki tuzak**: yol yoksa $\delta = +\infty$; **negatif ağırlıklı çevrim** erişiliyorsa $\delta = -\infty$.
3. **BFS özel durumları**: ağırlık 1 / aynı pozitif / küçük-toplamlı seri açma → doğrusal.

4. **Algoritma haritası**: DAG → $O(V+E)$; Bellman-Ford → $O(V \cdot E)$; Dijkstra (negatif yok) → $O(V \log V + E)$.
5. **Mesafe yeter**: ebeveyn işaretçileri δ'dan $O(V+E)$'de kurulur.
6. **DAG relaxation**: d tahmini ∞'dan başlar, üçgen eşitsizliği ihlalinde gevşet; relax **güvenli**.
7. **Doğruluk**: topolojik sırada işle; tümevarımla tüm $d = \delta$; $O(V+E)$.

❗ Tek Bir Cümle

Ağırlıklı en kısa yol, toplam kenar ağırlığının minimumudur; bir DAG'da bunu, ∞'dan başlayan mesafe tahminlerini topolojik sırada üçgen eşitsizliğine göre güvenle gevşeterek $O(V+E)$'de tam olarak çözeriz.

23.13 Kontrol Soruları

ℹ Soru 1: Negatif ağırlıklı çevrim neden en kısa yolu tanımsız (−∞) yapar? +∞ durumundan farkı ne?

Cevap: $+\infty$, s'den t'ye **hiç yol olmadığı** durumdur (erişilemez). $-\infty$ ise yol *vardır* ama **sonlu en kısa yol yoktur**: s'den v'ye giden yol negatif ağırlıklı bir çevrim üzerindeki bir düğümden geçiyorsa, o çevrimi her dolaştığımızda ağırlık azalır (örn. tur başına -2), bu yüzden istediğimiz kadar küçük bir değere inebiliriz. Minimum erişilemez → infimum $-\infty$. Bu yüzden negatif çevrim erişilen düğümler için ebeveyn ağacı kurmayız.

ℹ Soru 2: Neden yalnızca mesafeleri (δ) hesaplayıp ebeveyn işaretçilerini sonradan kuruyoruz?

Cevap: Çünkü her algoritmada ikisini birden taşımak gereksiz angaryadır ve mesafeler ebeveyni *belirlemeye yeter*. δ verilince, her u için her çıkış komşusu v'yi tara; $\delta(s,v) = \delta(s,u) + w(u,v)$ ise (u,v) bir en kısa yolun son kenarıdır, $P(v) = u$ ata. Tüm düğüm + komşu üzerinde bir geçiş $O(V+E)$ — algoritmanın kendi alt sınırı zaten doğrusal olduğundan ek maliyet yok. Böylece tüm en kısa yol algoritmaları tek bir işe (mesafe) odaklanır.

ℹ Soru 3: "Kenar gevşetme güvenlidir" ne demek ve neden önemli?

Cevap: Güvenli (safe) = her mesafe tahmini $d(s,v)$ daima ya $+\infty$'dur ya da s'den v'ye **gerçek bir yolun** ağırlığıdır; asla hiçbir yola karşılık gelmeyen bir sayı olamaz. Gevşetmede $d(s,v) \leftarrow d(s,u) + w(u,v)$ yaparız; $d(s,u)$ zaten bir yolun ağırlığıysa, $u \to v$ kenarını ekleyince yine bir yolun ağırlığı olur. Önemi: tahmin **hiçbir zaman gerçek en kısa mesafenin altına inemez** (çünkü bir yol var, en kısa yol ondan kısa olamaz) — yani algoritma yukarıdan δ'ya doğru iner ve doğru değerde durur.

ℹ Soru 4: DAG relaxation neden topolojik sıra gerektirir? Sıra olmasaydı ne bozulurdu?

Cevap: Doğruluk, "bir düğümü işlerken tüm geçerli öncüllerinin (incoming neighbors) zaten doğru hesaplanmış olması"na dayanır. Topolojik sıra tam bunu garanti eder: $s \to v$ en kısa yolunda v'den önceki u, topolojik sırada v'den **önce** gelir (DAG olduğundan), dolayısıyla u işlenirken $d(u) = \delta(s,u)$ olmuştur ve (u,v) gevşetilince $d(v)$ doğru değere iner. Rastgele sırada bir düğümü öncülü hesaplanmadan işlersek,

> onu eksik bilgiyle bırakırız; ayrıca çevrim olsaydı topolojik sıra zaten var olmazdı (bu yüzden DAG şart).

23.14 Egzersizler

Egzersiz 1. Verilen küçük ağırlıklı çizgede, belirtilen kaynaktan tüm δ değerlerini elle DAG relaxation ile hesapla; topolojik sıra seçimini de göster.

Egzersiz 2. Bir negatif ağırlıklı çevrim içeren çizge çiz; hangi düğümlerin $\delta = -\infty$, hangilerinin sonlu olduğunu işaretle.

Egzersiz 3. Pozitif ağırlıkları seri kenarlara açma dönüşümünü küçük bir çizgede uygula; toplam ağırlık $V + E$'yi aşmazsa BFS'in doğrusal kaldığını argümanla göster.

Egzersiz 4. Yalnızca δ mesafeleri verilen bir çizgede ebeveyn işaretçilerini kuran fonksiyonu Python'da yaz; süresinin $O(V + E)$ olduğunu doğrula.

Egzersiz 5. DAG relaxation'ın üçgen eşitsizliği koşulunu, "en uzun yol" (longest path) bulacak şekilde değiştir; bunun neden yalnızca DAG'da anlamlı olduğunu açıkla.

23.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 18 (L12) — Bellman-Ford (araya Ders 17 = PS5 girer)

Ders 18 (L12): Bellman-Ford (araya **Ders 17 = PS5** problem oturumu girer) — Jason Ku ile, DAG kısıtını kaldırıyoruz: *herhangi* bir çizgede (çevrim, hatta negatif ağırlıklı çevrim dahil) SSSP. Aynı "relax" tekniğini kullanır ama topolojik sıra olmadığından kenarları **tekrar tekrar** gevşetir; karesel-benzeri $O(V \cdot E)$ ama pratik. Negatif çevrimleri de tespit eder.
Ders 18 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (DAG relaxation) ve 2 (negatif çevrim) çöz.
- Üçgen eşitsizliği + relax-güvenli değişmezini ezberden anlat.
- Ana cümleyi tekrar oku: *"DAG relaxation: ∞'dan başla, topolojik sırada güvenle gevşet, $O(V + E)$'de bitir."*

23.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Ağırlık fonksiyonu	$w : E \to \mathbb{Z}$; $O(1)$ sorgu (tuple veya hash)	Böl. 1-2
Yol ağırlığı $w(\pi)$	Yoldaki kenar ağırlıklarının toplamı	Böl. 3

Kavram	Tanım	Sayfada
En kısa yol $\delta(s,t)$	Minimum ağırlıklı yol; yok $\rightarrow +\infty$	Böl. 3-4
Negatif çevrim	Erişilen düğüm için $\delta = -\infty$ (infimum)	Böl. 4
Algoritma haritası	DAG $O(V+E)$ / Bellman-Ford $O(V \cdot E)$ / Dijkstra $O(V \log V + E)$	Böl. 6
Üçgen eşitsizliği	$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$	Böl. 8
Relax (u,v)	$d(s,v) > d(s,u) + w(u,v)$ ise düşür; güvenli	Böl. 8-9
DAG relaxation	Topolojik sırada relax; $O(V+E)$	Böl. 10

23.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "ağırlık girince en kısa yol = toplam ağırlığın minimumu" sezgisini kurar ve DAG relaxation ile gevşetme tekniğini açar — köprülerin özeti:

1. **Ağırlıklı en kısa yol** → GPS/yönlendirme (Google Maps, Waze), ağ yönlendirme (OSPF latency), oyun yol bulma.
2. **DAG relaxation** → proje zamanlama (kritik yol, PERT/CPM), build sistemi süre tahmini, elektronik tablo hesap sırası.
3. **Üçgen eşitsizliği** → `A*` sezgisel araması, metrik uzaylar, yer gömme (embedding) kalitesi.
4. **Relax (gevşetme)** → kısıt çözme (constraint relaxation), iteratif optimizasyon, label-correcting algoritmalar.
5. **Negatif çevrim tespiti** → arbitraj (döviz çevrim grafı), kâr döngüsü, tutarsızlık tespiti.
6. **Topolojik sıra + DP** → DAG üzerinde dinamik programlama; en kısa/en uzun yol DP'nin habercisidir (Ders 23-26, L15-L18).

! Tek bir şey alıp gideceksen

Kenarlara ağırlık verince "en kısa yol" toplam ağırlığın minimumu olur ve iki yeni tuzak doğar ($+\infty$ yol yok, $-\infty$ negatif çevrim). Bir DAG'da bu problemi, ∞'dan başlayan mesafe tahminlerini topolojik sırada üçgen eşitsizliğine göre **güvenle gevşeterek** $O(V+E)$'de çözeriz — ve mesafeler, ebeveyn işaretçilerini sonradan kurmaya yeter.

24 Problem Oturumu 5

Ders 13-15'in uygulaması: beş çizge problemi BFS/DFS'e iner — yarıçap ve 2-yaklaşım, süpernode router gecikmesi, büyülü kapılar bileşen sayımı, sabitten yararlanan 3-şehir turu, durum-uzayı ve ortada buluşma

ℹ Oturum bilgisi

- **Solomon'un videosu:** YouTube — Problem Session 5 (≈88 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 5
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 17 (Problem Oturumu 5)
- **Hoca:** Justin Solomon
- **Okuma süresi:** ≈24 dk

24.1 Bu Problem Oturumu Ne Hakkında?

Beşinci problem oturumu (Justin Solomon) **BFS ve DFS** üzerine kurulu beş çizge problemini çözer (Şekil 39.1). Hepsinin ortak teması şudur: problem ilk bakışta "ağırlıklı en kısa yol" veya "gezgin satıcı" gibi *zor* görünür, ama doğru **indirgeme** ile ağırlıksız BFS/DFS'e veya basit bir sayıma iner.

> *"we're going to cover some problems in graph theory related to depth-first search and breadth-first search."* — Solomon, 0:20

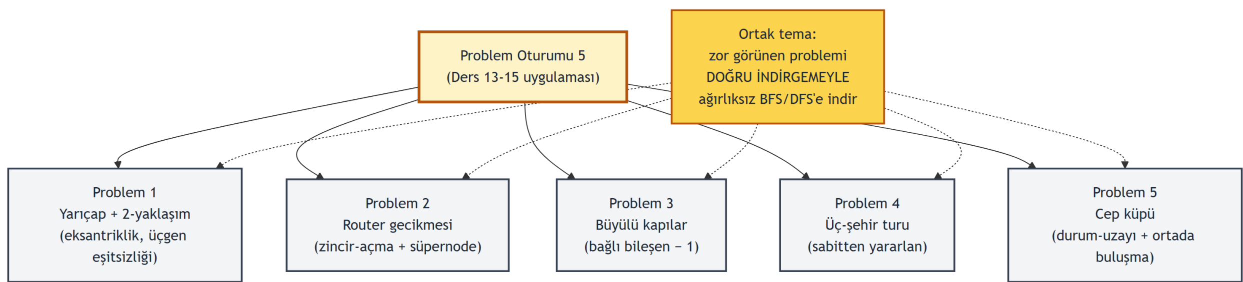
Solomon'un tekrarladığı meta-ders üç adımda toplanır:

- **Süslemeyi soy.** 6.006 problemleri basit hesaplama problemlerini bol metinle süsler; ilk iş paragrafı madde madde ayıklamaktır.
- **Asıl soruyu çıkar.** "Asıl ne soruluyor?" — en kısa yol mu, sadece bir sayım mı, yoksa bir min-maks mı?
- **Verilen sabitleri çıkar.** "Verilen sabitler neler?" — "tam 3 şehir", "tel ≤ 100r", "derece ≤ 4" gibi sabitler indirgemenin anahtarıdır.

Her problem bu ayıklamadan sonra "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir.

Yaklaşım — ortak strateji: zoru kolaya indirgemek

Beş problemin tamamı aynı refleksle başlar: önce metni soyup "asıl ne soruluyor" ve "hangi sabitler verildi" diye çıkar; sonra problemi **bildiğin bir araca** (ağırlıksız BFS, full DFS/bağlı bileşenler, bir sayım) indirge. Bu oturumun beş indirgeme tekniği — eksantriklik + üçgen eşitsizliği, zincir-açma + süpernode, bedava-kapı bileşenleri, sabit-sayıda BFS, ortada buluşma — algoritma tasarımının "tanıdık



Şekil 24.1: Problem Oturumu 5'in kavram haritası: kök (PS5) beş probleme dallanır ve ortadaki ortak tema düğümü beşini birden yönlendirir. Problem 1 yarıçabı eksantriklikle hesaplar, sonra üçgen eşitsizliğiyle tek BFS'lik 2-yaklaşıma iner; Problem 2 ağırlıklı router hattını zincir-açmayla ağırlıksızlaştırıp süpernode ile tek BFS'e indirger; Problem 3 büyülü kapı problemini bedava-kapı çizgesinin bağlı bileşen sayısı eksi bir olarak çözer; Problem 4 üç-şehir turunu sabit sayıda BFS ve permütasyona indirir; Problem 5 cep küpünü durum-uzayı çizgesi olarak modelleyip ortada buluşmayla arar. Ortak tema — zor görünen problemi doğru indirgemeyle ağırlıksız BFS/DFS'e veya basit sayıma çevir — Solomon'un her probleme aynı kapıdan girmesini sağlar.

alt-yapıya yönlendir" kasını çalıştırır.

24.2 Problem 1: Çizge Yarıçapı ve Eksantriklik

İfade. Bağlı bir yönsüz çizge G verilir. Bir düğümün **eksantrikliği** $\varepsilon(v) = \max_w \mathrm{dist}(v, w)$ (en uzaktaki düğüme mesafe); çizgenin **yarıçapı** $R(G) = \min_u \varepsilon(u)$. (a) $R(G)$'yi $O(V \cdot E)$'de hesapla. (b) R'yi daha hızlı ($O(E)$) **2-yaklaşımla** tahmin et: $R \leq R^* \leq 2R$.

Yaklaşım — min-maks: merkezi bul, sonra üçgen eşitsizliğiyle gevşet

Bu bir **min-maks** problemidir: metrik geometride bir dairenin merkezini bulmak gibi — her noktanın en uzak noktaya mesafesi minimumda merkezde gerçekleşir. (a) şıkkında tanımı doğrudan kodla: her düğümden BFS, en uzak mesafe = eksantriklik, bunların minimumu = yarıçap. (b) şıkkında ise *tek bir* düğümün eksantrikliğinin yeterli bir tahmin olduğunu **üçgen eşitsizliğiyle** göster — kesin merkezi bulmak zorunda kalmadan.

> *"the radius… is the min over all of the different vertices, u, of the eccentricity of u."* — Solomon, 3:25

Çözüm.

(a) Kesin yarıçap. Her v için BFS ile tüm mesafeleri bul, maksimumunu al (= $\varepsilon(v)$), bunların minimumunu tut.

```
def radius_exact(adj):                      # (a) kesin R — O(V·E)
    best, center = None, None
    for u in adj:                           # V kez BFS
        e = max(bfs(adj, u)[0].values())    # ε(u) = en uzak mesafe
```

```
        if best is None or e < best:
            best, center = e, u
    return best, center                         # (R, merkez)
```

(b) 2-yaklaşım. *Herhangi* bir u seç, $R^* = \varepsilon(u)$ döndür (tek BFS). İki sınır:

- **Alt sınır.** $R = \min_u \varepsilon(u) \le \varepsilon(u) = R^*$ (minimum, herhangi bir değerden küçük-eşittir).
- **Üst sınır.** $u_0 = \arg\min \varepsilon$ gerçek merkez, $\bar{v}$ ise u'dan en uzak düğüm olsun. Üçgen eşitsizliği: $R^* = \text{dist}(u, \bar{v}) \le \text{dist}(u, u_0) + \text{dist}(u_0, \bar{v}) \le R + R = 2R$.

```
def radius_2approx(adj, u=None):                # (b) 2-yaklaşım — O(E)
    if u is None:
        u = next(iter(adj))                     # HERHANGİ bir düğüm
    return max(bfs(adj, u)[0].values())         # R* = ε(u);  R ≤ R* ≤ 2R
```

> *"Justin's favorite inequality is the triangle inequality."* — Solomon, 24:43

Şekil 24.2 bu sınırı bir **yol çizgesi** 0-1-2-3-4 üzerinde motordan **gerçek** verilerle gösterir: kesin merkez düğüm 2'dir ve $\varepsilon(2) = R = 2$ (üst panel); keyfi $u = 0$ seçilince $R^* = \varepsilon(0) = 4 = 2R$ olur (alt panel) — bu çizge tam olarak sınırın doyduğu kötü durumdur. Üçgen eşitsizliği $R \le R^* \le 2R$ tek BFS'in kesin yarıçabın en çok iki katı olduğunu garanti eder.

Karmaşıklık. (a) V kez BFS = $O(V \cdot (V + E))$; **bağlı** çizgede $V = O(E)$ olduğundan $V + E = O(E)$ → $O(V \cdot E)$. (b) tek BFS → $O(E)$ (V faktörü düşer).

24.3 Problem 2: Router Gecikmesi ve Süpernode

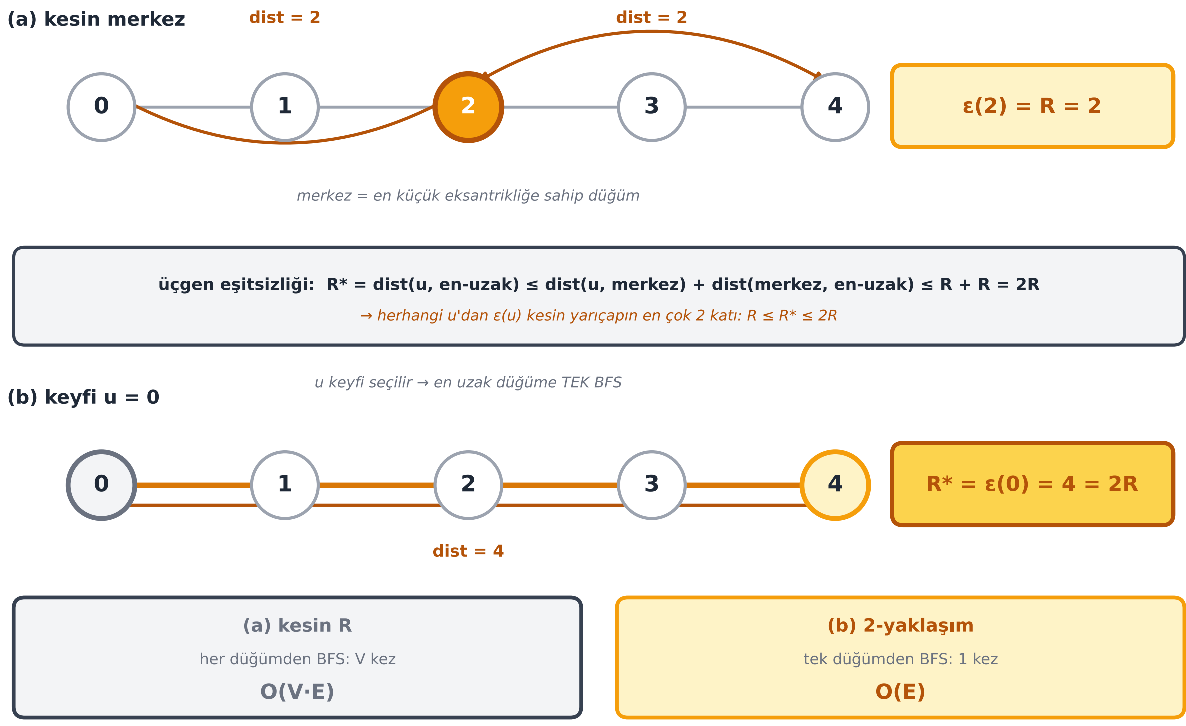
İfade. r router, bazıları **giriş noktası (entry point)**; çift yönlü teller, her biri pozitif tamsayı uzunluk l_i. Bir router'ın **gecikmesi** = en yakın giriş noktasına en kısa yol. Toplam tel $\le 100r$ ve her router bir giriş noktasına ulaşır. Tüm router'ların gecikme toplamını $O(r)$'de hesapla.

> Yaklaşım — iki klasik hile: zincir-açma (ağırlıksızlaştır) + süpernode (tek kaynak)
>
> İlk bakışta ağırlıklı en kısa yol gibi görünür ama değil. İki hile birleştirilir: (1) **Zincir-açma** — toplam tel $\le 100r$ olduğundan, ağırlık-l kenarı l tane ağırlık-1 kenardan oluşan bir zincire açılır → ağırlıksız çizge, BFS geçerli. (2) **Süpernode** — her router için her giriş noktasına ayrı ayrı BFS yapmak iç içe bir döngüdür; bunun yerine tüm giriş noktalarına bağlı *tek* sanal düğüm s eklenir ve s'ten tek BFS tüm gecikmeleri bir dalgada verir.

> *"he is a supernode, which is a term of art."* — Solomon, 40:18

Çözüm. Her router bir düğüm; her tel l_i kenarlık zincire açılır (yardımcı "aux" düğümlerle), böylece $V = O(r)$ ve $E \le 100r = O(r)$. Tüm giriş noktalarına bağlı **tek bir süpernode** s ekle. Üçgen eşitsizliğiyle, s'ten bir router'a en kısa yol önce s'ten en yakın giriş noktasına ($+1$ kenar) sonra oradan router'a gider — yani

$$\text{gecikme}(i) = \text{dist}(s, i) - 1.$$



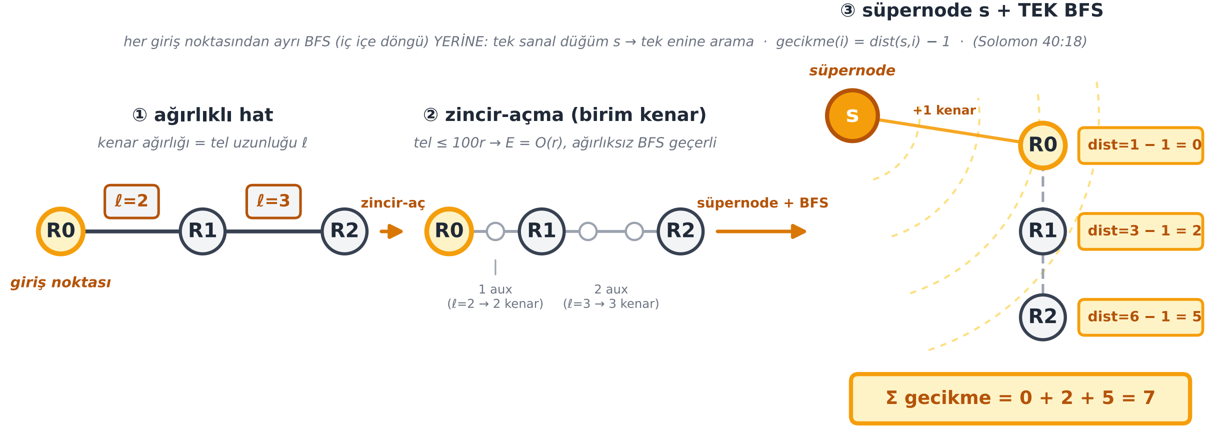
Şekil 24.2: Yarıçap ve 2-yaklaşım — tek BFS ε(u) kesin R'nin en çok 2 katı (R ≤ R* ≤ 2R) — Problem 1 İMZA. Yol çizgesi 0-1-2-3-4 (motordan GERÇEK). ÜST panel (a) KESİN MERKEZ: gerçek merkez düğüm 2 amber dolgu; iki uca mesafe yayları (her ikisi 2) → ε(2) = R = 2 rozeti; merkez = en küçük eksantrikliğe sahip düğüm. ALT panel (b) KEYFİ u=0: keyfi seçilen u=0 (slate dolgu); en uzak düğüm 4'e tek BFS yolu vurgulu → R* = ε(0) = 4 = 2R rozeti — bu çizge sınırın DOYDUĞU kötü durum. ORTADA üçgen eşitsizliği kutusu (Solomon 24:43): R* = dist(u, en-uzak) ≤ dist(u, merkez) + dist(merkez, en-uzak) ≤ R + R = 2R. ALT şerit maliyet: (a) kesin R = V kez BFS → O(V·E) | (b) 2-yaklaşım = TEK BFS → O(E), V faktörü düşer.

Tek BFS (kaynak s) ile tüm $\mathrm{dist}(s,i)$ bulunur; cevap $\sum_i(\mathrm{dist}(s,i)-1)$.

```
def total_latency_supernode(n_nodes, wires, entries):   # O(r)
    adj = expand_weighted_edges(n_nodes, wires)   # zincir-açma: ağırlıklı → ağırlıksız
    s = ("super", 0)                              # tek sanal süpernode
    adj[s] = []
    for e in entries:                             # s → her giriş noktası (+1 kenar)
        adj[s].append(e); adj[e].append(s)
    delta, _ = bfs(adj, s)                        # TEK BFS (iç içe döngü YOK)
    return sum(delta[i] - 1 for i in range(n_nodes))   # gecikme = dist(s,i) - 1
```

Şekil 24.3 üç panelde tüm hileyi motordan **gerçek** bir örnekle gösterir: 3 router, teller $(0\text{-}1,\ l{=}2)$ ve $(1\text{-}2,\ l{=}3)$, giriş noktası kümesi $\{0\}$. Zincir-açma $l{=}2$ teli için 1, $l{=}3$ teli için 2 aux düğüm üretir. Süpernode BFS'i $\mathrm{dist}(s,\cdot)=(1,3,6)$ verir → gecikmeler $0,2,5$ ve toplam $0+2+5=7$.

Karmaşıklık. Süpernode tek düğüm + (giriş sayısı kadar) kenar ekler (asimptotik değişmez). Tek BFS, $V+E=O(r)\rightarrow O(r)$.



Şekil 24.3: Süpernode hilesi — iç içe döngü YERİNE tek BFS dalgası — Problem 2 İMZA. Üç panel (motordan GERÇEK: 3 router, teller (0-1,ℓ=2),(1-2,ℓ=3), giriş {0}). AĞIRLIKLI HAT: router R0–R1–R2; kenar ağırlığı = tel uzunluğu ℓ; giriş noktası R0 amber çerçeve. ZİNCİR-AÇMA: her ağırlık-ℓ kenarı ℓ birim-kenara açılır, araya aux düğümler (ℓ=2→1 aux/2 kenar, ℓ=3→2 aux/3 kenar); tel ≤ 100r → E = O(r), ağırlıksız BFS geçerli. SÜPERNODE s + TEK BFS: s büyük amber düğüm girişe +1 kenarla bağlanır; eş-merkezli BFS dalga yayları; her router gecikme rozeti dist(s,i) − 1 = (0, 2, 5); toplam kutusu Σ gecikme = 0 + 2 + 5 = 7 (Solomon 40:18).

24.4 Problem 3: Potry Harter ve Büyülü Kapılar

İfade. n odalı bir labirent, her oda ≤ 4 kapı (yani düğüm derecesi ≤ 4). Kapılar kapalı; bazıları **büyülü (enchanted)** — açmak maliyetli, diğerleri bedava. Tüm odaları ziyaret etmek için **açılması gereken minimum**

büyülü kapı sayısını $O(n)$'de bul.

Yaklaşım — TSP tuzağı: aslında bedava-kapı bileşenlerini say

Tuzak: gezgin satıcı (TSP) gibi görünür ama **değil**. İki gözlem onu basitleştirir: (1) bir büyülü kapı açıldıktan sonra açık kalır — ileri-geri geçiş artık bedava; (2) en kısa yol *önemsizdir*, yalnız açılan kapı **sayısı** sorulur. Anahtar: bedava kapılarla birbirine bağlı odalar tek bir "kümedir" (bedava gezilebilen öbek). Geriye kalan, bu öbekleri birbirine bağlamak için kaç büyülü kapı gerektiğidir — bu da bir yayılma ağacı sayımıdır.

> *"we're actually going to remove the enchanted doors."* — Solomon, 51:17

Çözüm. Çizge kur: düğüm = oda, kenar = **yalnız büyülü-olmayan (bedava) kapılar**. Bu çizgenin **bağlı bileşenlerini** (full BFS/DFS) hesapla — her bileşen, bedava gezilebilen bir oda öbeğidir. Cevap: **(bağlı bileşen sayısı)** $- 1$.

Gerekçe: bileşenleri tek düğüm sayan bir meta-çizge düşün; hepsini birbirine bağlayan bir **yayılma ağacının** (spanning tree) kenar sayısı tam olarak (düğüm sayısı $- 1$) = (bileşen sayısı $- 1$)'dir, ve her böyle kenar bir büyülü kapıya karşılık gelir.

```
def min_enchanted_doors(n_rooms, free_doors):           # O(n)
    adj = make_undirected(free_doors) if free_doors else {}
    for r in range(n_rooms):
        adj.setdefault(r, [])                           # izole oda = kendi bileşeni
    return len(connected_components(adj)) - 1           # bileşen − 1 (yayılma ağacı)
```

> *"the number of edges in a spanning tree of my graph is exactly the number of vertices in my graph minus 1."* — Solomon, 58:08

Karmaşıklık. Derece $\leq 4 \rightarrow V = O(n)$ ve $E = O(n)$; bağlı bileşenler $O(V + E) = $ $O(n)$.

24.5 Problem 4: Purity Atlantic ve Sabitten Yararlanma

İfade. Bir havayolu; ev şehri + ziyaret edilecek **tam 3 şehir**; toplam **aktarma (connection)** sayısını en aza indiren rotayı bul. c şehir, f uçuş; hedef $O(c + f)$.

Yaklaşım — verilen sabiti sömür: 3 şehir → O(1) permütasyon ve çift

"Tüm-çiftler en kısa yol" gibi görünür ama yalnızca **ilgilenilen 4 şehir** (ev + 3) önemlidir. Burada asıl numara verilen **sabiti** sömürmektir: ziyaret edilecek şehir sayısı tam 3 olduğundan, permütasyon sayısı $3! = 6$ (sabit), gerekli şehir-çifti sayısı $2 \cdot \binom{4}{2} = 12$ (sabit). Bu sabitler patlamadığı için problem doğrusal kalır.

> *"this is one of these problems where you're really taking advantage of the constants that we gave you."* — Solomon, 1:09:06

Çözüm. Çizge: düğüm = şehir, kenar = uçuş. 4 ilgili şehrin her çifti (12 yönlü çift) için BFS ile en kısa yol uzunluğunu hesapla (aktarma sayısı = yol uzunluğu $- 1$). Sonra 6 permütasyonu (ev $\to 1 \to 2 \to 3 \to$ ev) gez, her birinin toplam maliyetini bul, minimumu döndür.

```
def best_three_city_tour(adj, home, cities):          # O(c + f)
    pts = [home] + list(cities)                       # 4 ilgili şehir
    dist = {}
    for p in pts:                                     # 4 BFS (sabit sayıda)
        delta, _ = bfs(adj, p)
        for q in pts:
            dist[(p, q)] = delta.get(q)
    best, order = None, None
    for perm in permutations(cities):                 # 3! = 6 permütasyon
        legs = [(home, perm[0]), (perm[0], perm[1]),
                (perm[1], perm[2]), (perm[2], home)]
        total = sum(dist[l] - 1 for l in legs)        # aktarma = yol - 1
        if best is None or total < best:
            best, order = total, perm
    return best, order
```

Karmaşıklık. $12\times$ BFS $= 12 \cdot O(c+f) = O(c+f)$; 6 permütasyon $= O(1)$. Toplam $O(c+f)$. (Eğer "m şehir" deselerdi $m!$ patlardı — sabit olması kritik.)

24.6 Problem 5: Cep Küpü — Durum Çizgesi ve Ortada Buluşma

İfade. 2×2 Rubik küpü (pocket cube). Hamle = (yüz f_j, yön s). (a) Farklı konfigürasyon sayısının 12 milyondan az olduğunu göster. (b) Her düğümün derecesini bul. (c) Bir küpü en kısa hamle dizisiyle çözen hızlı algoritma.

Yaklaşım — durum-uzayı çizgesi + ortada buluşma (çift-yönlü BFS)

Klasik **durum-uzayı çizgesi**: düğüm = küpün bir konfigürasyonu (renk durumu), kenar = bir hamle; çözüm = "düz" küpe en kısa yoldur. (a) ve (b) doğrudan sayımla çözülür. (c) için tek-yönlü BFS yavaştır (orta seviyelerde milyonlarca düğüm); **ortada buluşma (meet-in-the-middle)** ile kaynaktan *ve* hedeften *paralel* BFS yürütülür — iki dalga yarı-derinlikte buluşur.

> *"think of every vertex of my graph as being the state of some system and every edge as being a transition."* — Solomon, 1:14:18

Çözüm.

(a) Konfigürasyon üst sınırı. Bir köşeyi sabitlersek geriye 7 köşe kalır; bunlar $\le 7!$ farklı düzende sıralanabilir ve her köşe 3 yönde dönebilir (3^7). Üst sınır:

$$3^7 \cdot 7! = 11\,022\,480 < 12 \text{ milyon.}$$

(b) Derece. 3 döndürülebilir yüz $\times$ 2 yön = **derece 6** (her düğümde sabit).

(c) Ortada buluşma. Tek-yönlü BFS, çözülebilir tüm konfigürasyonları gezer (~3 milyon; durum çizgesi 3 bağlı bileşenli, çap 14). Bunun yerine kaynaktan *ve* hedeften **paralel** BFS yap; seviye kümeleri ortada kesişir, hiçbir seviye yol uzunluğunun yarısından ($\lceil w/2 \rceil$) büyük olmaz.

```
def bidirectional_bfs(adj, s, t):                # ortada buluşma — Solomon 1:27:08
    ds, dt = {s: 0}, {t: 0}                       # iki ayrı mesafe haritası
    qs, qt = deque([s]), deque([t])              # iki ayrı kuyruk (paralel BFS)
    while qs and qt:
        # küçük cepheyi genişlet; komşu DİĞER taraftaysa → buluşma, dur
        ...                                       # kesişim: ds[u] + 1 + dt[v]
```

> *"I'm going to run BFS sort of in parallel for two different vertices… The source and the target."*
> — Solomon, 1:27:08

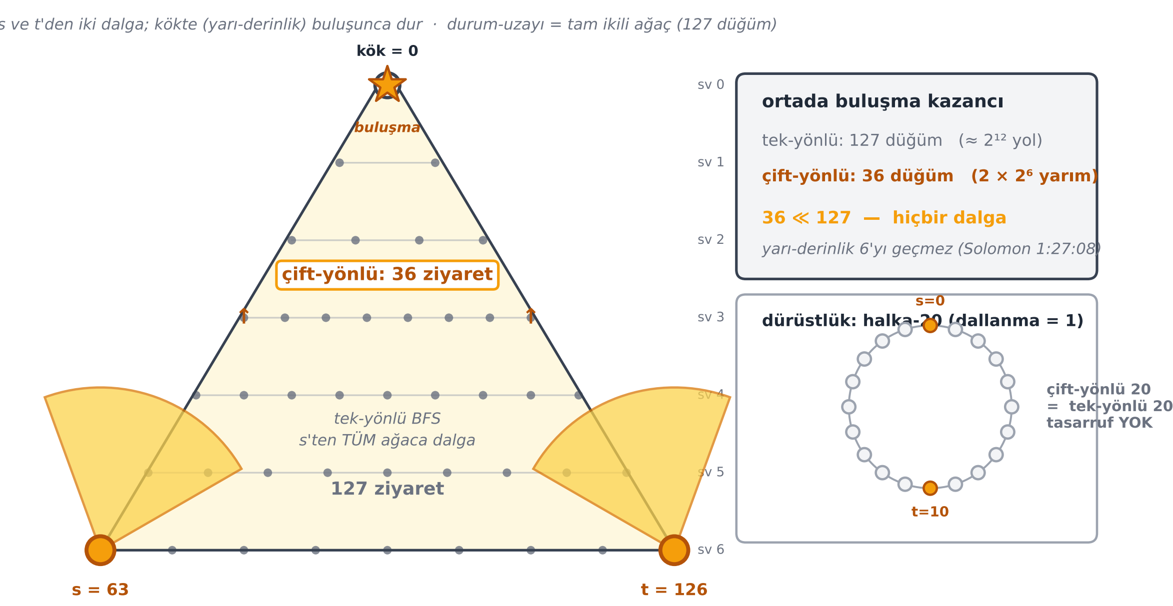
Şekil 24.4 bu kazancı bir durum-uzayı maketi olan **tam ikili ağaç (127 düğüm)** üzerinde motordan **gerçek** çalıştırarak gösterir: iki uç yaprak $s = 63$ ve $t = 126$ arasındaki en kısa yol köke çıkıp inen 12 kenardır. Tek-yönlü BFS tüm ağacı tarar → **127 ziyaret**; çift-yönlü BFS iki küçük dalgayı yarı-derinlikte (6) buluşturup yalnız **36 ziyaret** eder.

> ⚠ Dürüstlük notu — kazanç ÜSTEL dallanmadan gelir, çizgilerden değil
>
> Notion'un "üstel büyüme yarıya iner" iddiası **yalnızca üstel dallanan** durum-uzaylarında doğrudur. Figürdeki yan panel bunu bir karşı-örnekle dürüstçe gösterir: dallanma çarpanı 1 olan bir **halka-20** çizgesinde (s=0, t=10) çift-yönlü BFS de tek-yönlü BFS de **20 düğüm** ziyaret eder — tasarruf YOKtur. Ortada buluşmanın gerçek kazancı, N^w büyüyen bir uzayı $2 \cdot N^{\lceil w/2 \rceil}$'ye indirmesinden, yani üstel dallanmayı yarı-derinliğe çekmesinden gelir; cep küpü bu üstel rejimde olduğu için kazanç gerçektir.

Karmaşıklık. Tek-yönlü BFS, çözülebilir tüm konfigürasyonları (~3 milyon) gezer. Ortada buluşma, gezilen düğüm sayısını ~$2 \cdot N^{\lceil w/2 \rceil}$'ye indirir ($N_i$ = i hamlede erişilen konfigürasyon sayısı).

24.7 Ne Öğrendik?

> ❗ Yedi Taşınabilir Araç
>
> Bu oturum, Ders 13-15'in BFS/DFS teorisini beş somut problemde uyguladı ve yedi taşınabilir araç kazandırdı:
>
> 1. **Süpernode:** birden çok "hedef"i tek düğüme bağlayıp tek BFS ile hepsine mesafeyi çöz (iç içe döngüden kurtul).
> 2. **Ağırlıklı → ağırlıksız:** küçük-toplamlı tamsayı ağırlıkları kenar zincirine açıp BFS'in doğrusallığını koru.
> 3. **Bağlı bileşenler:** "öbek" problemleri (büyülü kapılar) full BFS/DFS + (#bileşen − 1) ile çözülür; yayılma ağacı argümanı.
> 4. **Sabitten yararlan:** "tam 3 şehir" gibi sabitler permütasyon/çift sayısını $O(1)$ yapar — m olsaydı patlardı.



Şekil 24.4: Ortada buluşma — çift-yönlü BFS üstel büyümeyi yarıya iner — Problem 5 İMZA. Durum-uzayı maketi: tam ikili ağaç 127 düğüm (motordan GERÇEK). İki uç yaprak s=63 (sol-alt) ve t=126 (sağ-alt) amber işaretli; en kısa yol köke çıkıp inen 12 kenar. Tek-yönlü BFS: s'ten TÜM ağaca dev dalga → 127 ziyaret (soluk üçgen). Çift-yönlü: s ve t'den iki küçük amber kama; kökte (yarı-derinlik 6) buluşma yıldızı → 36 ziyaret. Karşılaştırma kutusu: 36   127, hiçbir dalga yarı-derinlik 6'yı geçmez (Solomon 1:27:08). YAN panel DÜRÜSTLÜK: halka-20 (dallanma=1) — çift-yönlü 20 = tek-yönlü 20, TASARRUF YOK; kazanç üstel dallanmadan gelir. Alt şerit: pocket cube durum-uzayı $3^7 \times 7! = 11.022.480 < 12$ milyon; düğüm derecesi = 3 yüz × 2 yön = 6.

5. **Durum-uzayı çizgesi:** düğüm = sistem durumu, kenar = geçiş; bulmaca çözmek = en kısa yol (Rubik, satranç, AI arama).
6. **Ortada buluşma:** kaynak + hedeften paralel BFS, üstel arama uzayını yarı-derinliğe indirir (ama yalnız üstel dallanmada kazanç verir).
7. **Üçgen eşitsizliği + arg min/arg max:** yaklaşım sınırlarını $(2R)$ ve süpernode mesafelerini kanıtlamanın aracı.

24.8 Sonraki

Ders 18 (L12) — Bellman-Ford

Sırada **Ders 18 (L12): Bellman-Ford** var — Jason Ku ile, DAG kısıtını kaldırıyoruz: *herhangi* bir ağırlıklı çizgede (çevrim, hatta negatif ağırlıklı çevrim dahil) tek-kaynak en kısa yol. DAG relaxation'ın "kenar gevşetme" tekniği aynı kalır ama topolojik sıra olmadığından kenarlar tekrar tekrar gevşetilir; negatif ağırlıklı çevrimler de tespit edilir.

25 Bellman-Ford

DAG kısıtını kaldırırız: herhangi bir ağırlıklı çizgede (çevrim ve negatif ağırlık dahil) tek-kaynak en kısa yol; sonlu en kısa yol V−1 kenardan uzun olamaz (basittir), çizgeyi V+1 seviyeye çoğaltıp DAG yaparak DAG relaxation çağırırız, V kenarda hâlâ kısalan tanıkları bulup onlardan eksi sonsuz yayarız — hepsi O(V·E)

> **i Bölüm bilgisi**
>
> - **Ku'nun videosu:** YouTube — Lecture 12: Bellman-Ford (~57 dk)
> - **OCW sayfası:** MIT 6.006 Lecture 12: Bellman-Ford
> - **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 18 (L12)
> - **Hoca:** Jason Ku (ağırlıklı en kısa yollar ünitesinin en genel algoritması)
> - **Okuma süresi:** ~27 dk
>
> Bir önceki ünite dersinde (Ders 16) DAG relaxation, ağırlıklı SSSP'yi yalnızca **çevrimsiz** çizgelerde $O(V + E)$'de çözdü. Bu ders DAG kısıtını kaldırır: Bellman-Ford *herhangi* bir çizgede çalışır, negatif ağırlıklı çevrimleri tespit eder ve onlardan erişilen düğümlere $-\infty$ atar.

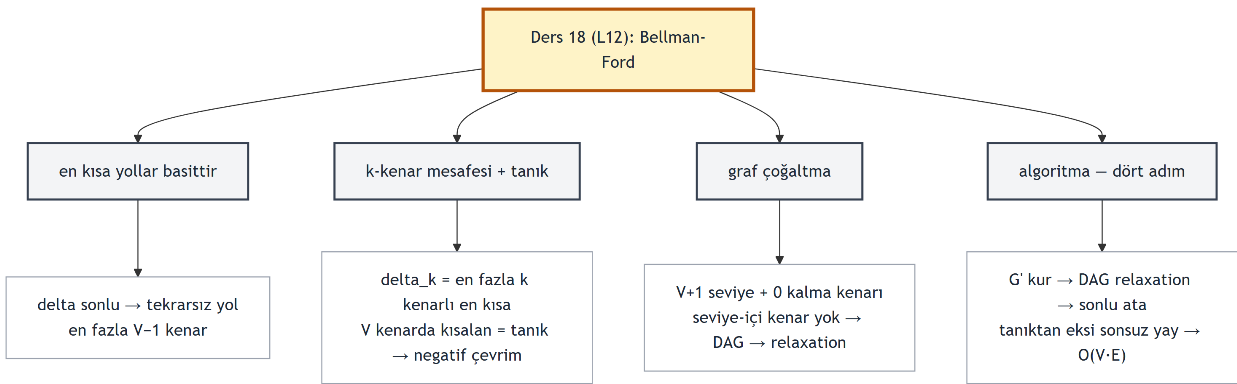
25.1 Bu Derste Ne Var?

DAG relaxation (Ders 16) yalnızca çevrimsiz çizgelerde çalışıyordu. Bu ders (Jason Ku), **en genel** tek-kaynak en kısa yol algoritmasını verir: **Bellman-Ford**. *Herhangi* bir ağırlıklı çizgede — çevrimler ve negatif ağırlıklar dahil — çalışır; negatif ağırlıklı çevrimleri tespit eder ve onlardan erişilen düğümlere $-\infty$ atar.

> *"if a negative weight cycle is reachable from our source, then the vertices in that cycle and anything reachable from that cycle will potentially have an unbounded number of edges."* — Ku, 2:22

Üç ana fikir:

1. **En kısa yollar basittir** — δ sonluysa düğüm tekrarsız (simple) bir en kısa yol vardır; bu da en fazla $V - 1$ **kenar** demektir.
2. **k-kenar mesafesi + tanık** — "en fazla k kenarlı" mesafeyi izleyerek, V kenarda hâlâ kısalan düğüm bir **tanıktır** (witness) → negatif çevrim kanıtı.
3. **Graf çoğaltma** — çizgeyi $V + 1$ seviyeye kopyalayıp DAG'a çevir; DAG relaxation'ı çağır → $O(V \cdot E)$.



Şekil 25.1: Ders 18'in (L12) kavram haritası: kök = Bellman-Ford (Ku) — herhangi ağırlıklı çizgede en genel SSSP. Dört dal — (1) en kısa yollar basittir: delta sonlu ise düğüm tekrarsız yol var → en fazla V−1 kenar; bu, sonsuz yol kümesini sonlandırır. (2) k-kenar mesafesi delta_k = en fazla k kenarlı en kısa mesafe; V kenarda hâlâ kısalan düğüm bir tanıktır → negatif çevrim kanıtı; delta = eksi sonsuz ise düğüm bir tanıktan erişilir. (3) graf çoğaltma: çizgeyi V+1 seviyeye kopyala, kenarlar yalnız bir üst seviyeye gider + 0 ağırlıklı kalma kenarı → seviye-içi kenar yok → DAG; şimdi DAG relaxation kara kutusu çağrılır. (4) algoritma dört adım: G' kur → DAG relaxation → sonlu mesafeleri ata → tanıktan eksi sonsuz yay; toplam O(V·E). Sonuç: çevrimli/negatif problemi, çözmeyi bildiğimiz çevrimsiz bir probleme indirger; bedeli yalnız bir V çarpanı.

Builder Notu — DAG Kısıtı Kalkınca

DAG relaxation (Ders 16) topolojik sıraya dayanıyordu, çevrim olduğunda topolojik sıra yoktur. Bellman-Ford aynı "relax" tekniğini korur ama çevrime ve negatif ağırlığa izin verir; karşılığında bir V çarpanı öder. Püf noktası tek bir gözlemdir: sonlu bir en kısa yol $V-1$ kenardan uzun olamaz.

- **İleriye → yönlendirme:** Bellman-Ford, distance-vector yönlendirme protokollerinin (RIP) temelidir — her router komşularından mesafe alıp gevşetir.
- **İleriye → arbitraj:** negatif ağırlıklı çevrim tespiti, döviz çevrim grafında **kâr döngüsü** (arbitraj) bulmaktır.
- **İleriye → graf çoğaltma:** "düğümü duruma göre çoğalt" tekniği, durum-augmentasyonlu birçok problemde (zaman, yakıt, mod) tekrar eder.
- **Geriye → DAG relaxation (Ders 16):** Bellman-Ford, problemi bir DAG'a indirgeyip o algoritmayı kara kutu olarak kullanır.

Tek cümle: *En kısa yol sonluysa $V-1$ kenardan uzun olamaz; çizgeyi $V+1$ seviyeye çoğaltıp DAG yaparak DAG relaxation çağırırız, V kenarda hâlâ kısalan düğümleri tanık olarak işaretleyip $-\infty$ yayarız — hepsi $O(V \cdot E)$.*

25.2 1. Bellman-Ford'un Hedefi

Çizge çevrim ve negatif ağırlık içerebilir. Hedef: her düğüm için $\delta(s, v)$ hesapla — erişilemez ise $+\infty$, **negatif ağırlıklı çevrimden** erişilen ise $-\infty$, diğerleri sonlu. Ek olarak, varsa bir negatif çevrim döndür. Çalışma süresi hedefi $O(V \cdot E)$.

25.3 2. Isınma: Yönsüz Çevrim ve İndirgeme

İki kısa alıştırma:

- **Yönsüz çizgede negatif çevrim ⇔ negatif kenar.** Yönsüz bir negatif kenar üzerinde ileri-geri gidersek (uzunluk-2 çevrim) negatif çevrim olur. Yani yönsüz hâl ilginç değil; bu ders **yönlü** çizgelere odaklanır.
- $O(V \cdot (V+E)) \to O(V \cdot E)$ **indirgemesi.** Bir algoritma SSSP'yi $O(V \cdot (V+E))$'de çözüyorsa: önce BFS/DFS ile s'den **erişilebilenleri** bul, gerisini at. Kalan çizgede $V = O(E)$ (bağlı bileşen en fazla $E+1$ düğüm), dolayısıyla $V + E = O(E)$ → toplam $O(V \cdot E)$. (Bu yüzden $V \cdot (V+E)$ hedefiyle $V \cdot E$ aynı.)

25.4 3. En Kısa Yollar Basittir

Çalışılan Örnek — Claim 1. $\delta(s, v)$ **sonlu** ise, s'den v'ye **basit** (simple, düğüm tekrarsız) bir en kısa yol vardır.

> *"if my shortest path distance from S to some vertex is finite… there exists a shortest S to V path that is simple."* — Ku, 11:06

Kanıt: bir en kısa yol bir çevrim C içerse, δ sonlu olduğundan $w(C)$ negatif **olamaz** (negatif olsaydı tekrar tekrar dolaşıp $-\infty$ yapardık). $w(C) \geq 0$ ise çevrimi atıp daha kısa (veya eşit) bir yol elde ederiz; tekrar tekrar atarak basit bir yola ineriz.

Sonuç (kutula): basit yol en fazla V düğüm → en fazla $V - 1$ **kenar** kullanır.

> *"simple paths have at most V minus 1 edges."* — Ku, 14:06

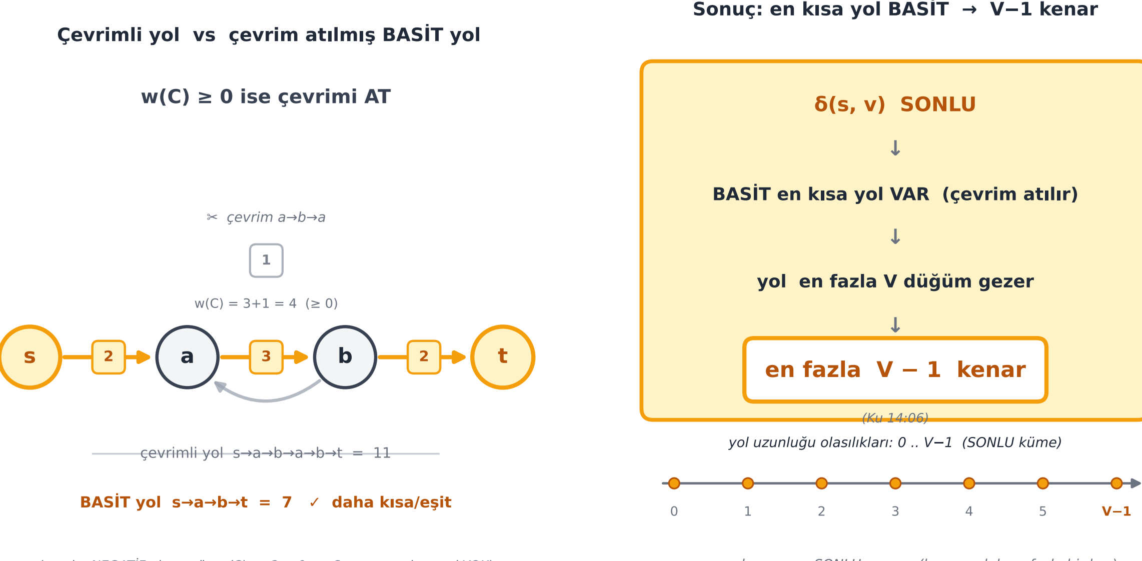
Yani δ sonluysa, yalnızca **en fazla** $V - 1$ **kenarlı** yolları kontrol etmek yeter (sonsuz değil, sonlu bir küme).

Şekil 25.2 bu gözlemi iki panelde gösterir: solda pozitif/sıfır ağırlıklı çevrimi atınca basit yolun daha kısa/eşit olduğu mini örnek (çevrimli yol 11 vs basit yol 7), sağda zincir akışı (δ sonlu → basit yol → en fazla $V - 1$ kenar) ve sonlu arama uzayı.

25.5 4. k-Kenar Mesafesi

k-kenar mesafesi $\delta_k(s, v)$: en fazla k kenar kullanan en kısa $s \to v$ yolunun ağırlığı. Eğer δ_k'yı $k = V - 1$ için hesaplarsak ve $\delta(s, v)$ sonluysa, gerçek en kısa yolu bulmuş oluruz (çünkü sonlu en kısa yol $\leq V - 1$ kenar).

Bu, problemi sonlandırır: sonsuz sayıda yol yerine, kenar sayısını $V - 1$ ile sınırlayıp **adım adım** mesafe hesaplarız.



Şekil 25.2: En kısa yollar BASİTTİR: δ sonlu → çevrim at → en fazla V−1 kenar (Bellman-Ford'u SONLANDIRIR) (L12 §3 İMZA). SOL panel pozitif/sıfır çevrim durumu: mini çizge s→a→b→t + b→a geri kenarı (çevrim a→b→a, w(C) = 3+1 = 4 ≥ 0). Çevrimli yol s→a→b→a→b→t (çevrim parçası soluk + makas) ağırlık 11 üstü çizili vs çevrimi ATILMIŞ BASİT yol s→a→b→t (amber kalın) ağırlık 7 — w(C) ≥ 0 ise çevrimi at → daha kısa/eşit. Dipnot: çevrim NEGATİF olursa (build_bf_example w(C) = −2 < 0) δ = −∞, en kısa yol YOK. SAĞ panel kural kutusu: δ(s,v) SONLU → BASİT en kısa yol VAR → en fazla V düğüm → en fazla V−1 kenar (Ku 14:06); altında 0..V−1 sonlu sayı doğrusu (sonsuz yol uzayı yerine sonlu arama). Veri MOTORDAN: path_weight basit = 7 ≤ çevrimli = 11 (w(C)=4≥0); build_bf_example çevrim = −2 (Ku 11:06, 14:06).

25.6 5. Tanık (Witness) ve $-\infty$

Peki $-\infty$ düğümleri? Anahtar gözlem: eğer V **kenarlı** en kısa mesafe (δ, $k = V$), $(V-1)$ **kenarlı** mesafeden (δ, $k = V - 1$) **kesinlikle küçükse**, bu yeni (daha kısa) yol basit olamaz (V kenar $> V - 1$ düğüm → tekrar var) → içinde bir negatif çevrim vardır. Böyle bir v'ye **tanık (witness)** denir.

> *"I'm going to call V is a witness."* — Ku, 19:53

İddia: $\delta(s, v) = -\infty \Leftrightarrow v$, bir tanıktan **erişilebilir**. Yani tüm tanıkları bulup onlardan erişilen her düğümü $-\infty$ işaretlemek yeter.

Şekil 25.3 örnek çizgeyi ve δ_k tablosunu birlikte gösterir: üstte negatif çevrimli çizge (kaynak `a`), altta $k = 0 \ldots 5$ satırları; $k = V = 4$ satırında `b` düğümünün $-5 \to -7$ düşmesi **tanık** vurgusudur (kenar tablosu motordan: `b` sütunu $[\infty, -5, -5, -5, -7, -7]$).

25.7 6. Her Negatif Çevrim Bir Tanık İçerir

Çalışılan Örnek — kanıt. İddiayı kanıtlamak için şunu göstermek yeter:

> *"It suffices to prove that every negative weight cycle contains a witness."* — Ku, 22:43

Bir negatif çevrim C al; her v için üçgen eşitsizliği: $\delta_V(s, v) \leq \delta_{V-1}(s, v') + w(v', v)$ ($v' = v$'nin çevrimdeki öncülü). Bu eşitsizliği **çevrimdeki tüm düğümler üzerinde topla**: sol taraf $\sum \delta_V$, sağ taraf $\sum \delta_{V-1} + w(C)$. $w(C) < 0$ olduğundan, sol toplam sağ toplamdan **kesinlikle küçük** olur. O hâlde çevrimde en az bir düğüm $\delta_V < \delta_{V-1}$ eşitsizliğini sağlamalı — yani bir **tanık** içerir. (Hiçbiri tanık olmasaydı toplam eşitsizliği çökerdi: çelişki.)

Şekil 25.4 bu kanıtı üç panelde toplar: solda örnek çizge (tanık `b` yıldızlı, çevrimden erişilen `b`/`c`/`d` koyu $-\infty$), ortada yayılım akışı (tanık → DFS erişilebilirlik → $-\infty$ boyaması), sağda çevrim üzerinde toplama + $w(C) < 0$ çelişki argümanı.

25.8 7. Graf Çoğaltma

Modifiye Bellman-Ford'un fikri: bir düğümün **birçok sürümünü** yap; v'nin k. sürümü "v'ye en fazla k kenarla ulaşmak" durumunu temsil etsin. Buna **graf çoğaltma (graph duplication)** denir.

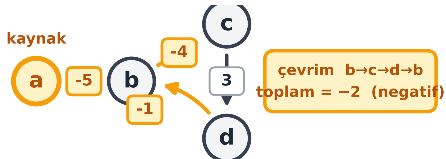
> *"this is an idea called graph duplication."* — Ku, 30:51

$V + 1$ seviye kur; seviye k'daki `v_k` düğümü, "v'ye en fazla k kenarla ulaşmak". Kenarlar yalnızca **daha yüksek** seviyeye gider → çizge bir **DAG** olur (geriye kenar yok).

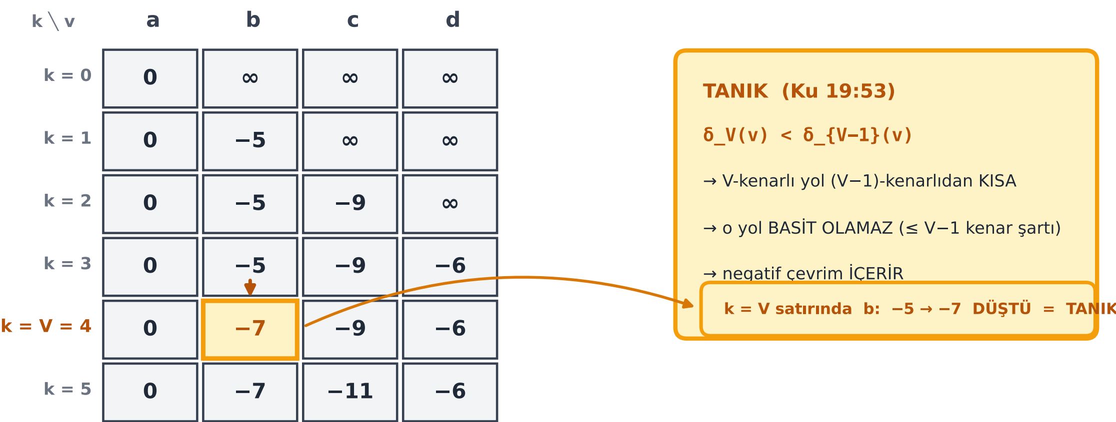
> *"if we connect edges from one level to only higher levels… then this graph is going to be a DAG."*
> — Ku, 33:14

k-kenar mesafesi δ_k + TANIK: δ_V < δ_{V−1} ise basit yol olamaz → negatif çevrim → −∞

Ağırlıklı çizge (kaynak a, negatif çevrim)



δ_k(a, v) = en fazla k kenarlı en kısa mesafe (her satır: bir kenar daha gevşet)

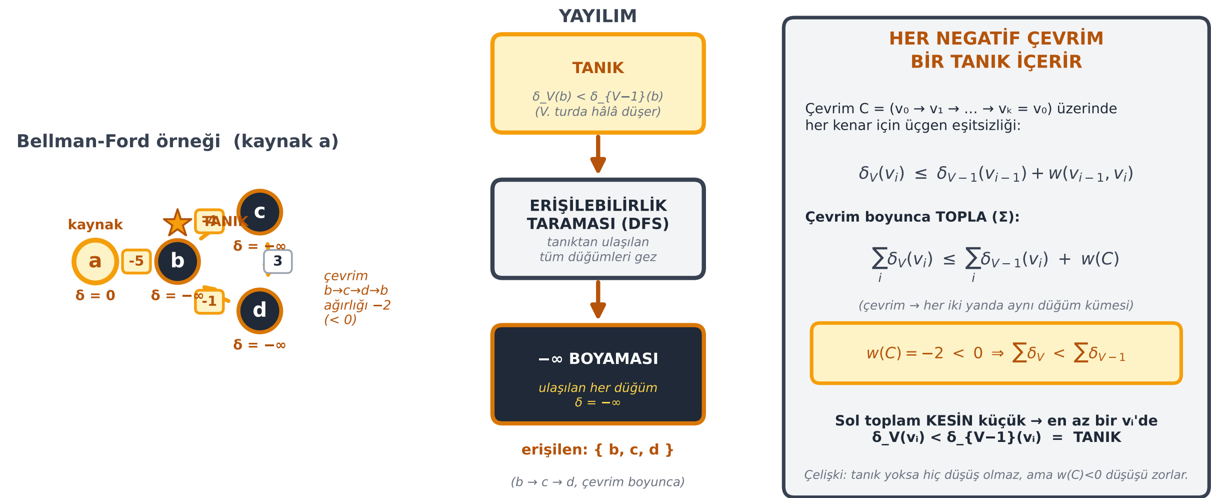


k \ v	a	b	c	d
k = 0	0	∞	∞	∞
k = 1	0	−5	∞	∞
k = 2	0	−5	−9	∞
k = 3	0	−5	−9	−6
k = V = 4	0	−7	−9	−6
k = 5	0	−7	−11	−6

δ(a, v) = −∞ ⟺ v bir tanıktan erişilebilir (negatif çevrim her turda ağırlığı düşürür → sınır yok)

burada b, c, d hepsi çevrimden erişilir → δ(a, b) = δ(a, c) = δ(a, d) = −∞; yalnız a = 0 sonlu

Şekil 25.3: k-kenar mesafesi δ_k + TANIK: δ_V < δ_{V−1} ise basit yol olamaz → negatif çevrim → −∞ (L12 §4-5 İMZA). ÜST: örnek çizge — a→b(−5) girişi + b/c/d çevrim üçgeni (b→c −4, c→d 3, d→b −1; toplam −2 amber negatif rozet). ALT: δ_k(a, v) tablosu — satırlar k = 0..5, sütunlar a/b/c/d; hücrelerde δ değerleri (∞ glifi). k = V = 4 satırında DÜŞEN b hücresi (−5 → −7) AMBER dolgu + aşağı-ok = TANIK; çevrim her turda −2 kazandırır. Sağ kenar TANIK kutusu (Ku 19:53): δ_V(v) < δ_{V−1}(v) → V-kenarlı yol (V−1)-kenarlıdan kısa → o yol BASİT OLAMAZ → negatif çevrim içerir → δ(a,v) = −∞. ALT NOT: δ(a,v) = −∞ ⟺ v bir tanıktan erişilebilir; burada b,c,d hepsi çevrimden erişilir → −∞, yalnız a = 0 sonlu. Veri MOTORDAN: k_edge_distances b = [∞,−5,−5,−5,−7,−7], c = [∞,∞,−9,−9,−9,−11], d = [∞,∞,∞,−6,−6,−6]; k=V=4'te −5→−7 TANIK düşüşü.



Şekil 25.4: Bellman-Ford: TANIK (V. turda düşer) → DFS erişilebilirlik → −∞ yayılımı · her negatif çevrim bir tanık içerir (L12 §5-6 İMZA). SOL panel örnek çizge (kaynak a): çevrim b→c→d→b (toplam −2 < 0); TANIK olan b amber yıldızlı; çevrimden erişilen b,c,d koyu dolgulu δ = −∞; a = 0 temiz. ORTA panel yayılım akışı (3 aşama dikey): TANIK (δ_V(b) < δ_{V−1}(b), V. turda hâlâ düşer) → ERİŞİLEBİLİRLİK TARAMASI (DFS, tanıktan ulaşılan her düğüm) → −∞ BOYAMASI; erişilen küme { b, c, d }. SAĞ panel kanıt kutusu (Ku 22:43): çevrim C üzerinde her kenar için üçgen eşitsizliği δ_V(v_i) ≤ δ_{V−1}(v_{i−1}) + w(v_{i−1}, v_i); çevrim boyunca TOPLA → Σ δ_V ≤ Σ δ_{V−1} + w(C); w(C) = −2 < 0 → Σ δ_V < Σ δ_{V−1} → en az bir v_i'de düşüş = TANIK (çelişki: tanık yoksa düşüş olmaz ama w(C) < 0 zorlar). Veri MOTORDAN: bellman_ford_classic(a) = {a:0, b:−∞, c:−∞, d:−∞}; V. tur tanık = [b] (δ_4(b)=−7 < δ_3(b)=−5); erişilen = {b,c,d}; çevrim w = −2.

DAG olduğu için DAG relaxation'ı (doğrusal) çağırabiliriz. Çoğaltılmış çizge V kat büyür: $V \cdot (V+1)$ düğüm, $V \cdot (V+E)$ kenar — bu da DAG relaxation'ı $O(V \cdot (V+E)) = O(V \cdot E)$ yapar (Bölüm 2 indirgemesiyle).

Şekil 25.5 örnek çizgeyi $V+1=5$ seviyeye açar (20 düğüm = 4×5): orijinal kenarlar seviye atlar, her düğüme 0-ağırlıklı kalma kenarı eklenir, seviye-içi kenar olmadığından sonuç bir DAG'dır; vurgulu yol $a_0 \to b_1 \to c_2 \to d_3 \to b_4$ çevrimin "açılmış" hâlidir.

25.9 8. Graf Dönüşümü Örneği

Örnek: a, b, c, d düğümlü yönlü çizge; $b \to c \to d \to b$ çevrimi ($-4+3+(-1) = -2$, negatif). $V+1=5$ kopya (seviye 0-4). Kurallar:

- **Seviye içi kenar YOK** (çevrimleri kırar).
- Orijinal her (u,v) kenarı için: seviye k'daki u'yu, seviye $k+1$'deki v'ye bağla (aynı ağırlık). Örn. `a_0 → b_1` (ağırlık -5).
- Her düğüm için 0**-ağırlıklı "kalma" kenarı**: `v_k → v_{k+1}` ("bir kenar kullanmadan aynı yerde dur" — *en fazla* k kenar koşulunu simüle eder).

> *"vertex Vk in level k represents reaching vertex V using at most k edges."* — Ku, 31:36

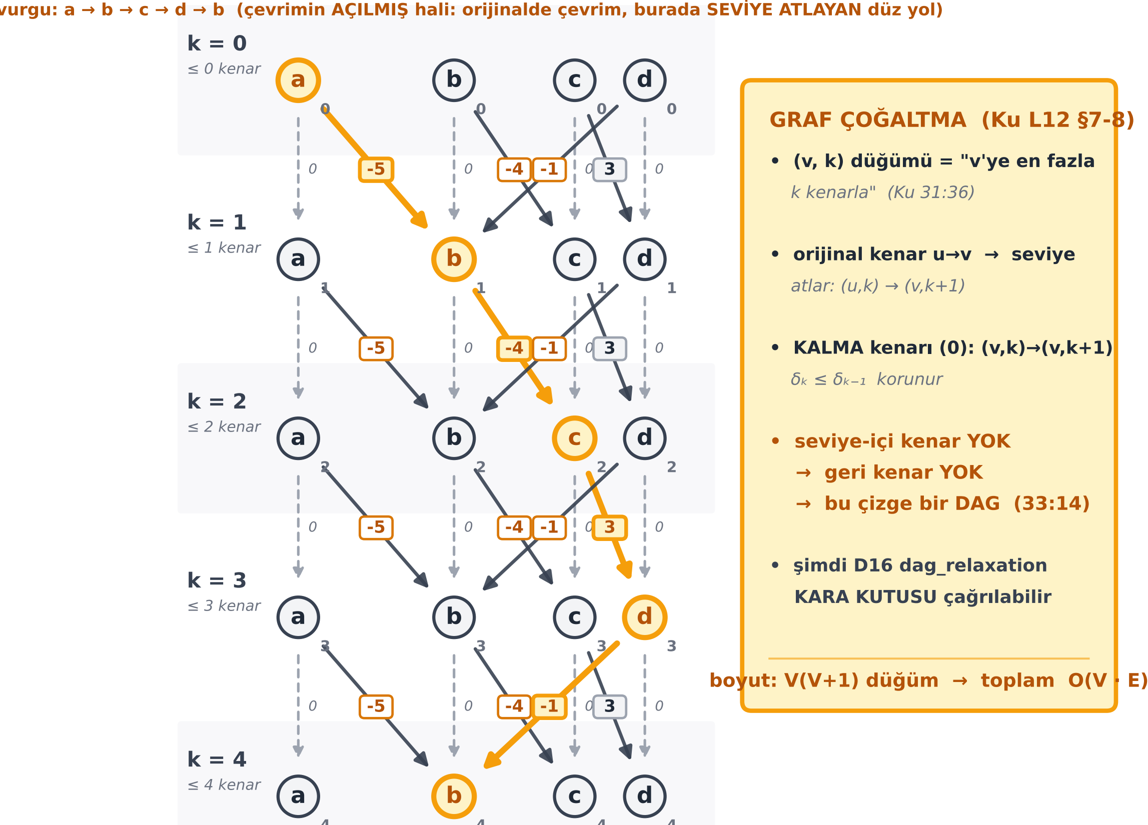
Böylece `a_0`'dan `b_3`'e bir yol = orijinal çizgede en fazla 3 kenarlı bir $a \to b$ yolu (örn. $a \to c \to d \to b$). Kenarlar daima seviye atladığından çizge çevrimsizdir.

> **ℹ Aynı çevrim, iki yerde**
>
> Şekil 25.5 ve bu bölümün örneği **aynı çizgeyi** kullanır: motorun `build_bf_example` çevrimi $b \to c \to d \to b = (-4)+3+(-1) = -2$, Notion §8 anlatısıyla birebir aynı çevrim. Yani çoğaltma figüründeki vurgu yolu, buradaki dönüşüm örneğinin sayısal karşılığıdır.

25.10 9. Bellman-Ford Algoritması

Dört adım:

```python
def bellman_ford(graph, s):
    # 1. V+1 seviyeli cogaltilmis DAG G' kur (seviye-atlamali + 0-agirlikli kalma kenarlari)
    G_prime = build_duplicated_dag(graph)              # O(V(V+E))
    # 2. DAG relaxation ile s_0'dan tum v_k'ya delta hesapla
    delta = dag_relaxation(G_prime, source=(s, 0))     # O(V(V+E))
    # 3. Sonlu mesafeleri ata: d(s,v) = delta(s_0, v_{V-1})
    d = {v: delta[(v, V - 1)] for v in graph.vertices}
    # 4. Taniklari bul, eristikleri -inf isaretle
    for v in graph.vertices:
        if delta[(v, V)] < delta[(v, V - 1)]:          # tanik kosulu
            for u in reachable_from(graph, v):         # O(E) her tanik
```



Şekil 25.5: Graf çoğaltma: V+1 seviye + 0-kalma kenarları → DAG (Bellman-Ford'un kalbi) (L12 §7-8 İMZA). Tek panel, 5 yatay şerit k = 0..4 (V+1 = 5 seviye, solda seviye etiketi + ≤ k kenar). Her şeritte a, b, c, d kopyaları küçük daireler. Orijinal kenarlar SEVİYE ATLAYAN oklar: a0→b1 (−5) amber örnek vurgulu; çevrim kenarları b→c(−4), c→d(3), d→b(−1) bir seviye atlayarak çizilir (orijinalde çevrim, burada düz). Her düğüme dik kesikli 0-ağırlıklı KALMA kenarı (v,k) → (v,k+1) ($\delta_k \leq \delta_{k-1}$). VURGU YOLU a0 → b1 → c2 → d3 → b4 amber kalın = çevrimin AÇILMIŞ hali. Sağ NOT kutusu (Ku §7-8): (v,k) = v'ye en fazla k kenarla (31:36); orijinal kenar → (u,k)→(v,k+1); kalma kenarı (v,k)→(v,k+1); seviye-içi kenar YOK → geri kenar YOK → bu çizge bir DAG (33:14); şimdi D16 dag_relaxation kara kutusu çağrılabilir; boyut V(V+1) düğüm → toplam O(V·E). Veri MOTORDAN: build_duplicated_dag → 4*5 = 20 düğüm; w2[(a0,b1)] = −5 (seviye atlar), w2[(a0,a1)] = 0 (kalma); adj2[a0] = [(a,1),(b,1)].

```
                d[u] = float('-inf')
    return d
```

Üçüncü adımda $d(s, v) = \delta(s_0, v_{V-1})$ ($V - 1$ kenarlı mesafe) atanır; bu, sonlu mesafeler için doğrudur.

Şekil 25.6 bu dört adımı dikey bir akışta toplar: (1) G' kur (çoğaltma), (2) DAG relaxation (Ders 16 kara kutusu), (3) sonlu ata, (4) tanık + $-\infty$ yay; toplam $O(V \cdot E)$. Yan notta motorun iki bağımsız sürümü (`via_dag` ve `classic`) örnek çizgede ve 100 rastgele çizgede birebir aynı sonucu verir.

25.11 10. Doğruluk ve Çalışma Süresi

Doğruluk (k üzerinden tümevarım). İddia: $\delta(s_0,$ v_k $) = \delta_k(s, v)$. Temel ($k = 0$): yalnız s_0 erişilir (0), gerisi ∞. Adım: v_k'ya en kısa yol, bir önceki seviyedeki bir komşudan gelir; G''nin komşulukları orijinal komşuluklar + "kalma" kenarıdır → minimum, tam olarak δ_k tanımını verir. Böylece sonlu mesafeler doğru atanır; tanık koşulu da $-\infty$'ları yakalar.

Çalışma süresi. G' kurma $O(V \cdot (V + E))$ + DAG relaxation aynı + her tanık için erişilebilirlik $O(E)$, en fazla V tanık → $O(V \cdot E)$. Toplam $O(V \cdot E)$.

> *"this thing takes order V times E work."* — Ku, 54:30

(Not: orijinal Bellman-Ford bu çoğaltmayı *yapmaz* — V tur kenar gevşetir; recitation'da $O(V)$ yer kullanan optimizasyon gösterilir.)

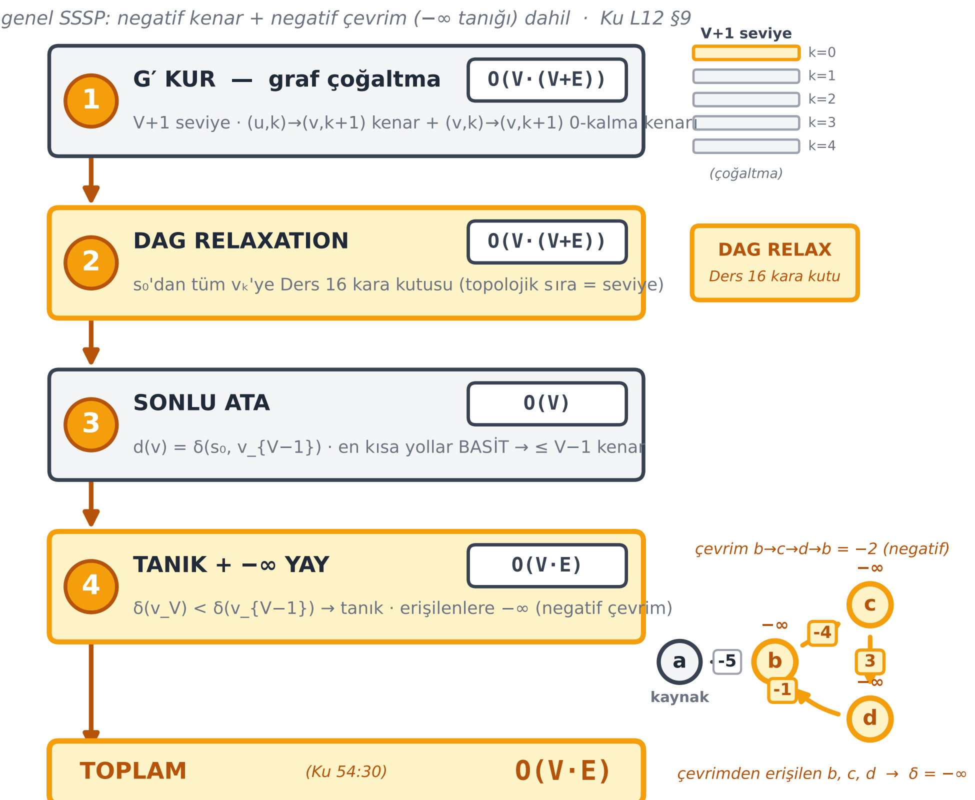
Şekil 25.7 bu dersi SSSP algoritma manzarasına yerleştirir: BFS (ağırlıksız), DAG relaxation (Ders 16, DAG), **Bellman-Ford** (bu ders, herhangi çizge $O(V \cdot E)$, amber vurgulu), Dijkstra (Ders 19, ≥ 0); altında Bellman-Ford'un indirgeme şeridi (çoğalt → DAG → çözülmüş kara kutu) ve motorun tutarlılık tanığı (`bellman_ford_classic == dag_sssp`).

25.12 Bu Dersin Özeti

1. **Hedef**: genel çizgede SSSP; ∞ (erişilemez), $-\infty$ (negatif çevrimden), sonlu; $O(V \cdot E)$.
2. **Basit yol**: δ sonlu → tekrarsız yol → en fazla $V - 1$ **kenar**.
3. **k-kenar mesafesi** δ_k: en fazla k kenarlı en kısa yol; $k = V - 1$ yeter (sonlu için).
4. **Tanık**: $\delta_V < \delta_{V-1}$ → negatif çevrim; $\delta = -\infty$ ⇔ tanıktan erişilir.
5. **Her negatif çevrim bir tanık içerir** (çevrim üzerinde toplam + üçgen eşitsizliği + $w(C) < 0$).
6. **Graf çoğaltma**: $V + 1$ seviye, seviye-atlamalı kenarlar + 0-ağırlıklı kalma → DAG.
7. **Algoritma**: G' kur → DAG relaxation → $\delta(s_0, v_{V-1})$ ata → tanıktan $-\infty$ yay; $O(V \cdot E)$.

> **!** Tek Bir Cümle
>
> Bellman-Ford, "sonlu en kısa yol $V - 1$ kenardan uzun olamaz" gerçeğini kullanır: çizgeyi $V + 1$ seviyeye çoğaltıp DAG yapar, DAG relaxation çağırır, V kenarda hâlâ kısalan tanıkları bulup onlardan $-\infty$ yayar — hepsi $O(V \cdot E)$.

Bellman-Ford = graf çoğaltma + Ders 16 DAG relaxation kara kutusu → O(V·E)



Yan not: klasik Bellman-Ford bu çoğaltmayı YAPMAZ — V−1 tur boyunca TÜM kenarları gevşetir, V. turda hâlâ
gevşeyen düğüm = tanık (Egzersiz 4). Motor iki sürümü de çalıştırdı: via_dag == classic == {a:0, b,c,d:−∞}
BİREBİR — örnek çizgede ve 100 rastgele çizgede de (sıfır uyuşmazlık).

Şekil 25.6: Bellman-Ford = graf çoğaltma + Ders 16 DAG relaxation kara kutusu → O(V·E) (L12 §9 İMZA). Dikey akış, dört adım kutusu (numara + başlık + açıklama + süre rozeti). (1) G KUR — graf çoğaltma: V+1 seviye, (u,k)→(v,k+1) kenar + (v,k)→(v,k+1) 0-kalma kenarı [O(V·(V+E))]; sağında V+1 seviye katman ikonu. (2) DAG RELAXATION — s_0'dan tüm v_k'ye Ders 16 kara kutusu (topolojik sıra = seviye) [O(V·(V+E))]; sağında 'Ders 16 kara kutu' rozeti. (3) SONLU ATA — d(v) = δ(s_0, v_{V−1}); en kısa yollar BASİT → ≤ V−1 kenar [O(V)]. (4) TANIK + −∞ YAY — δ(v_V) < δ(v_{V−1}) → tanık; erişilenlere −∞ (negatif çevrim) [O(V·E)]; sağında örnek çizge a→b(−5) + çevrim b→c→d→b = −2, çevrimden erişilen b,c,d düğümleri −∞ boyalı. En altta TOPLAM = O(V·E) (Ku 54:30). Yan not: klasik BF bu çoğaltmayı YAPMAZ — V−1 tur kenar gevşetir, V. turda hâlâ gevşeyen = tanık (Egzersiz 4); motor iki sürümü de çalıştırdı, via_dag == classic == {a:0, b,c,d:−∞} (örnek + 100 rastgele çizgede birebir). Veri MOTORDAN: bellman_ford_via_dag == bellman_ford_classic; tanık δ_V(b)=−7 < δ_{V−1}(b)=−5.

SSSP algoritma manzarası: Bellman-Ford en geneli — herhangi çizge, negatif çevrim tespiti (O(V·E))

algoritma	kabul ettiği çizge (kısıt)	çalışma süresi	
○ BFS	*ağırlıksız çizge*	O(V + E)	Ders 13
▷ DAG relaxation	*DAG + her ağırlık*	O(V + E)	Ders 16
★ BELLMAN-FORD	HERHANGİ çizge — negatif çevrim TESPİT + −∞	O(V · E)	BU DERS
◆ Dijkstra	*ağırlık ≥ 0*	O(V log V + E)	Ders 19

Bellman-Ford NASIL? — yeni algoritma icat etmez; problemi ÇÖZÜLMÜŞ bir DAG SSSP'sine İNDİRGER:

çoğalt (V+1 seviye) → DAG'a indirge (0-kalma kenarları) → çözülmüş kara kutu DAG relaxation

Bellman-Ford = en GENEL SSSP çözücü — negatif çevrimi bile tespit eder (δ = −∞ döndürür)

bedeli V çarpanı: O(V · E). Çizge daha kısıtlıysa (DAG → O(V+E), ağırlık ≥ 0 → Dijkstra) DAHA HIZLI yöntem seç.

`tutarlılık tanığı (motordan): aynı çoğaltma fikriyle bellman_ford_classic == dag_sssp (D16 DAG, kaynak e) → δ = {e:0, f:3, g:5, h:6, d:3, c:8}`

Şekil 25.7: SSSP algoritma manzarası: Bellman-Ford en geneli — herhangi çizge, negatif çevrim tespiti (O(V·E)) (L12 SENTEZ). Dört satırlık karşılaştırma panosu (algoritma | kabul ettiği çizge/kısıt | çalışma süresi | ders): (1) BFS — ağırlıksız çizge — O(V+E) [Ders 13]; (2) DAG relaxation — DAG + her ağırlık — O(V+E) [Ders 16]; (3) BELLMAN-FORD — HERHANGİ çizge, negatif çevrim TESPİT + $-\infty$ — O(V·E) [BU DERS, amber vurgulu]; (4) Dijkstra — ağırlık ≥ 0 — O(V log V + E) [Ders 19]. Altında BF indirgeme şeridi: çoğalt (V+1 seviye) → DAG'a indirge (0-kalma kenarları) → çözülmüş kara kutu DAG relaxation — BF yeni algoritma icat etmez, problemi çözülmüş bir DAG SSSP'sine İNDİRGER (OMSCS reduction refleksi). Alt kutu: BF = en GENEL SSSP çözücü, negatif çevrimi bile tespit eder ($\delta = -\infty$); bedeli V çarpanı O(V·E); çizge kısıtlıysa DAG (O(V+E)) veya Dijkstra (≥0) daha hızlı. Tutarlılık tanığı (motordan): bellman_ford_classic == dag_sssp (D16 DAG, kaynak e) → δ = {e:0, f:3, g:5, h:6, d:3, c:8}. Veri MOTORDAN: BF classic == via_dag == {a:0, b:$-\infty$, c:$-\infty$, d:$-\infty$}; çoğaltılmış DAG V+1 seviye = 20 düğüm.

25.13 Kontrol Soruları

Soru 1: Neden "δ sonluysa en fazla V−1 kenarlı yolları kontrol etmek yeter"? Bu Bellman-Ford'u nasıl sonlandırır?

Cevap: $\delta(s, v)$ sonluysa, en kısa yolların içindeki her çevrimin ağırlığı ≥ 0 olmalıdır (negatif olsaydı tekrar dolaşıp $-\infty$ yapardık). Sıfır/pozitif çevrimleri atarak **basit** (düğüm-tekrarsız) bir en kısa yol elde ederiz; basit yol en fazla V düğüm → $V - 1$ **kenar** içerir. Böylece sonsuz sayıda yol yerine, kenar sayısını $V - 1$ ile sınırlı **sonlu** bir kümeye bakarız — algoritma sonlanır (her $k = 0 \ldots V - 1$ için adım adım mesafe).

Soru 2: "Tanık (witness)" nedir ve −∞ düğümleriyle ilişkisi nedir?

Cevap: Tanık, V **kenarlı** en kısa mesafesi $(V - 1)$ **kenarlı** mesafesinden kesinlikle küçük olan bir düğümdür. Bu, V kenarlık daha kısa bir yolun var olması demektir; ama V kenar $> V - 1$ düğüm olduğundan o yol bir düğümü tekrar eder → içinde bir negatif ağırlıklı çevrim vardır. İlişki: $\delta(s, v) = -\infty \Leftrightarrow v$ bir tanıktan erişilebilir. Çünkü her negatif çevrim bir tanık içerir (kanıtlandı) ve negatif çevrimden erişilen her düğüm sınırsızca kısalan bir yola sahiptir. Algoritma tüm tanıkları bulur, onlardan erişilenleri $-\infty$ yapar.

Soru 3: Graf çoğaltma çizgeyi neden bir DAG'a çevirir, ve bu neden işe yarar?

Cevap: Çoğaltmada her düğümün $V + 1$ sürümü (her seviye için biri) yapılır ve **kenarlar yalnız daha yüksek seviyeye** gider (`v_k → v_{k+1}` seviye atlamalı ve "kalma" `v_k → v_{k+1}`). Geriye kenar olmadığından çevrim oluşamaz → **DAG**. İşe yarar çünkü DAG relaxation'ı (Ders 16, doğrusal $O(V+E)$) bu çoğaltılmış çizgeye uygulayabiliriz; çizge V kat büyüdüğünden $O(V \cdot (V + E)) = O(V \cdot E)$ elde ederiz. Yani çevrimli problemi, çözmeyi bildiğimiz çevrimsiz bir probleme **indirgemiş** oluruz.

Soru 4: Neden Bellman-Ford yönsüz çizgelerde "ilginç değil"?

Cevap: Yönsüz bir çizgede tek bir negatif ağırlıklı kenar bile bir negatif çevrim üretir: o kenar üzerinde ileri-geri gidersek (uzunluk-2 çevrim) toplam ağırlık negatiftir, sonsuza dek tekrarlanabilir. Yani yönsüzde "negatif çevrim var mı?" sorusu "negatif kenar var mı?" ile aynıdır ($O(E)$ basit kontrol) ve s'nin bağlı bileşeninde negatif kenar varsa her şey $-\infty$ olur. Bu yüzden problem yalnızca **yönlü** çizgelerde anlamlı; ders yönlü hâle odaklanır.

25.14 Egzersizler

Egzersiz 1. Küçük bir negatif-çevrimli yönlü çizgede, $k = 0 \ldots V$ için $\delta_k(s, v)$ tablosunu elle doldur; hangi düğümlerin tanık olduğunu ($\delta_V < \delta_{V-1}$) işaretle.

Egzersiz 2. Bir çizge için çoğaltılmış DAG G''yi ($V + 1$ seviye, kalma kenarları) çiz; `a_0`'dan bir `v_{V-1}`'e bir yolun, orijinalde en fazla $V - 1$ kenarlı bir yola karşılık geldiğini doğrula.

Egzersiz 3. "Her negatif çevrim bir tanık içerir" kanıtını, çevrim üzerinde toplama ve üçgen eşitsizliği

adımlarını yazarak yeniden üret.

Egzersiz 4. Bellman-Ford'u Python'da yaz (çoğaltma olmadan, klasik V-tur kenar gevşetme); negatif çevrim tespitini V. turda bir gevşetme olup olmadığına bakarak ekle.

Egzersiz 5. $O(V \cdot (V + E)) \to O(V \cdot E)$ indirgemesini bir çizgede uygula: BFS ile erişilemeyenleri at, kalan çizgede $V = O(E)$ olduğunu doğrula.

25.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 19 (L13) — Dijkstra

Ders 19 (L13): Dijkstra — Jason Ku ile, ağırlıklar **negatif değilse** çok daha hızlı bir algoritma: Dijkstra. Negatif çevrim derdi olmadığından, açgözlü (greedy) bir öncelik kuyruğuyla her düğümü bir kez "kesinleştirir" $\to O(V \log V + E)$, doğrusala yakın. Bellman-Ford'un $V \cdot E$'sinden hızlı; çoğu gerçek problem (yol ağı) için tercih edilen.
Ders 19 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (δ_k tablosu) ve 4 (klasik Bellman-Ford) çöz.
- "Simple yol $\leq V - 1$ kenar" ve "tanık" tanımlarını ezberden anlat.
- Ana cümleyi tekrar oku: *"Sonlu en kısa yol $V - 1$ kenardan uzun olamaz; çoğalt, DAG yap, gevşet, tanıkları $-\infty$ yap."*

25.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Bellman-Ford hedefi	Genel çizgede SSSP; ∞ / $-\infty$ / sonlu; $O(V \cdot E)$	Böl. 1
Basit en kısa yol	δ sonlu $\to$ tekrarsız $\to \leq V - 1$ kenar	Böl. 3
k-kenar mesafesi δ_k	En fazla k kenarlı en kısa yol; $k = V - 1$ yeter	Böl. 4
Tanık (witness)	$\delta_V < \delta_{V-1} \to$ negatif çevrim kanıtı	Böl. 5-6
$-\infty$ **kuralı**	$\delta = -\infty \Leftrightarrow$ bir tanıktan erişilebilir	Böl. 5
Graf çoğaltma	$V + 1$ seviye, seviye-atlamalı + kalma kenarı $\to$ DAG	Böl. 7-8
Algoritma	G' kur $\to$ DAG relaxation $\to \delta$ ($s_0 \to V - 1$ seviyesi) $\to$ tanık $-\infty$	Böl. 9
Çalışma süresi	$O(V \cdot E)$ (DAG relaxation V kat büyük çizgede)	Böl. 10

25.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "DAG kısıtını kaldıran en genel SSSP" fikrini kurar ve graf çoğaltma ile çevrimli/negatif problemi çözülmüş bir DAG'a indirger — köprülerin özeti:

1. **Bellman-Ford** → distance-vector yönlendirme (RIP), ağ topoloji keşfi, gecikme tabanlı yol.
2. **Negatif çevrim tespiti** → arbitraj (döviz çevrim grafı kâr döngüsü), kısıt tutarsızlığı (fark kısıtları).
3. **Graf çoğaltma** → durum-augmentasyonu: zaman-genişletilmiş çizge, yakıt/mod katmanları, takvim çakışması.
4. **k-kenar mesafesi** → sınırlı-atlama en kısa yol (en fazla k aktarma), kademeli yayılım.
5. **DAG'a indirgeme** → "zor problemi kolay kara kutuya çevir" (reduction); OMSCS CS 6515 ana teması.
6. **Üçgen eşitsizliği + toplama argümanı** → ortalama/amortize analiz, potansiyel fonksiyon kanıtları.

! Tek bir şey alıp gideceksen

Bellman-Ford'un sırrı tek bir gözlemdir — sonlu bir en kısa yol $V-1$ kenardan uzun olamaz. Çizgeyi $V+1$ seviyeye çoğaltıp çevrimsiz (DAG) hâle getirir, çözmeyi bildiğimiz DAG relaxation'ı çağırır; V kenarda hâlâ kısalan düğümleri "tanık" olarak yakalayıp onlardan erişilen her şeye $-\infty$ yayar. Çevrimli/negatif problemi, çevrimsiz bir probleme indirger — bedeli yalnızca bir V faktörü, yani $O(V \cdot E)$.

26 Dijkstra

Ağırlıklar negatif değilse mesafe en kısa yol boyunca artar; bu yüzden düğümleri bir öncelik kuyruğundan artan mesafe sırasında çekip kenarlarını gevşeterek tek-kaynak en kısa yolu çözeriz — değiştirilebilir öncelik kuyruğu decrease_key ile gevşeyen tahminleri günceller, BFS'in ağırlıklı genellemesi, neredeyse doğrusal O(V log V + E)

ℹ Bölüm bilgisi

- **Ku'nun videosu:** YouTube — Lecture 13: Dijkstra (~57 dk)
- **OCW sayfası:** MIT 6.006 Lecture 13: Dijkstra
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 19 (L13)
- **Hoca:** Jason Ku (ağırlıklı en kısa yollar ünitesinin negatif-olmayan ağırlık için hızlı algoritması)
- **Okuma süresi:** ~26 dk

Bir önceki ders (Ders 18) Bellman-Ford *herhangi* bir çizgede çalışıyordu ama $O(V \cdot E)$ — yoğun çizgede $O(V^3)$ — yavaştı. Bu ders, **ağırlıklar negatif değilse** çok daha hızlı bir algoritma verir: kaynaktan dışa **artan mesafe** sırasında ilerleyen Dijkstra, neredeyse doğrusal $O(V \log V + E)$.

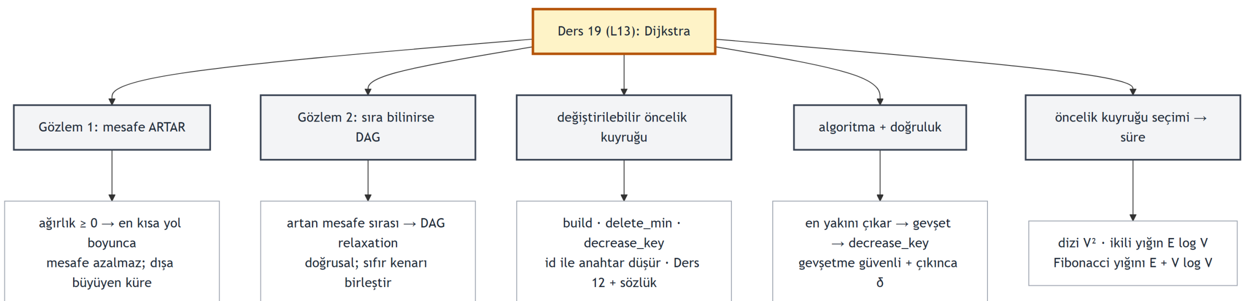
26.1 Bu Derste Ne Var?

Bellman-Ford (Ders 18) *herhangi* bir çizgede çalışıyordu ama $O(V \cdot E)$ (yoğun çizgede $O(V^3)$). Bu ders (Jason Ku), **ağırlıklar negatif değilse** çok daha hızlı bir algoritma verir: **Dijkstra**. Negatif çevrim derdi olmadığından, kaynaktan dışa **artan mesafe** sırasında ilerler ve $O(V \log V + E)$ — doğrusala yakın — süreye iner.

> *"If weights greater than or equal to 0, then distances increase along shortest paths."* — Ku, 8:35

Üç ana fikir:

1. **Negatif olmayan ağırlık → tekdüze mesafe** — en kısa yol boyunca mesafe azalmaz; bu, "kaynaktan dışa büyüyen küre" sezgisini mümkün kılar.
2. **Artan sıra bilinirse DAG relaxation** — düğümleri artan mesafe sırasında işleyebilseydik, çizgeyi bir DAG'a çevirip doğrusal çözerdik.
3. **Değiştirilebilir öncelik kuyruğu** — sıradaki en yakın düğümü verimlice bulmak için `decrease_key` destekli bir öncelik kuyruğu.



Şekil 26.1: Ders 19'un (L13) kavram haritası: kök = Dijkstra (Ku) — negatif olmayan ağırlıklı tek-kaynak en kısa yol, BFS'in ağırlıklı genellemesi. Beş dal — (1) Gözlem 1: ağırlık negatif değilse mesafe en kısa yol boyunca artar; kaynaktan dışa büyüyen küre sezgisi. (2) Gözlem 2: düğümlerin artan mesafe sırası bilinirse çizge bir DAG'a iner → DAG relaxation doğrusal çözer; sıfır ağırlıklı kenarlar birleştirilir. (3) değiştirilebilir öncelik kuyruğu: build, delete_min, decrease_key; id ile anahtarı düşür; öncelik kuyruğu (Ders 12) artı çapraz bağ sözlüğü. (4) algoritma artı doğruluk: en yakını çıkar, kenarları gevşet, decrease_key ile güncelle; gevşetme güvenli artı çıkınca delta. (5) öncelik kuyruğu seçimi süreleri belirler: dizi kare V, ikili yığın E log V, Fibonacci yığını E artı V log V. Sonuç: BFS'i ağırlıklı dünyaya taşır; sırrı tek gözlem, negatif yoksa mesafe artar.

Builder Notu — Negatif Yoksa Açgözlülük Doğru

Bellman-Ford genel çizgeyi $O(V \cdot E)$'de çözüyordu. Dijkstra, ağırlıkların negatif olmadığı kabulüyle tek bir gözlemden güç alır: mesafe en kısa yol boyunca artar, dolayısıyla "en yakın düğümü kesinleştir, bir daha dokunma" açgözlü stratejisi doğrudur. Bedeli $V \cdot E$ değil, neredeyse doğrusal $O(V \log V + E)$'dir.

- **İleriye → GPS/yönlendirme:** Dijkstra, Google Maps/Waze ve ağ yönlendirmenin (OSPF link-state) çekirdek algoritmasıdır.
- **İleriye → sezgisel arama:** Dijkstra'ya sezgisel (heuristic) eklenince `A*` olur; oyun yol bulma ve robotik hareket planlamada standart.
- **İleriye → öncelik kuyruğu seçimi:** array (yoğun) vs binary heap (seyrek) vs Fibonacci heap (teorik) — veri yapısı seçimi karmaşıklığı belirler.
- **Geriye → DAG relaxation (Ders 16) + öncelik kuyruğu (Ders 12):** Dijkstra ikisini birleştirir.

Tek cümle: *Negatif ağırlık yoksa mesafe en kısa yol boyunca artar; bu sayede düğümleri artan mesafe sırasında bir öncelik kuyruğundan çekip kenarlarını gevşeterek SSSP'yi $O(V \log V + E)$'de çözeriz — BFS'in ağırlıklı genellemesi.*

26.2 1. Manzara: Üç Algoritma ve Dijkstra'nın Yeri

Şimdiye dek ağırlıklı SSSP için üç yol gördük: (1) **BFS** (ağırlıkları tek-kenarlara açarak — yalnız pozitif ve küçük-toplamlı), (2) **DAG relaxation** (çevrimsiz, herhangi ağırlık, doğrusal), (3) **Bellman-Ford** (genel, negatif çevrim dahil, $O(V \cdot E)$). Bellman-Ford yoğun çizgede $O(V^3)$ — yavaş.

Çoğu gerçek problemde ağırlıklar **negatif değildir** ("evime negatif mesafe yok"). Bu kısıtla çok daha iyisini yapabiliriz: **Dijkstra**, neredeyse doğrusal $O(V \log V + E)$. (Buradaki $\log V$ pratikte ~30-60'tan büyük

olmaz.)

26.3 2. Gözlem 1: Negatif Olmayan Ağırlık → Mesafe Artar

İlk anahtar gözlem: ağırlıklar ≥ 0 ise, en kısa yol boyunca mesafe **(zayıf) tekdüze artar**.

> *"If weights greater than or equal to 0, then distances increase along shortest paths."* — Ku, 8:35

Yani $s \to v$ en kısa yolu bir u'dan geçiyorsa, $\delta(s, u) \leq \delta(s, v)$ (aradaki alt-yolun ağırlığı negatif olamaz). Bu, "kaynak merkezli bir küreyi dışa büyütüp önce yakın düğümleri keşfet" stratejisini mümkün kılar. Negatif ağırlık olsaydı, çok uzaktaki bir düğüm negatif kenarla küre içine "sızabilirdi" — bu strateji çökerdi.

Şekil 26.2 bu gözlemi eşmerkezli mesafe kürelerinde gösterir: kaynak s merkezde, her düğüm $\delta(s, v)$ yarıçaplı bir kürede; en kısa yol $s \to c \to a \to b$ daima dışa doğru (δ: $0 \to 3 \to 7 \to 9$, kesinlikle artan). Sağdaki karşı-örnek panel, hayali bir -8 kenarı eklenince uzak düğümün iç küreye "sızıp" yapının çöktüğünü — negatif ağırlığın neden yasak olduğunu — gösterir.

26.4 3. Gözlem 2: Artan Sıra Bilinirse DAG Relaxation

İkinci gözlem: SSSP'yi **daha hızlı** çözebiliriz — eğer düğümleri **artan mesafe sırasında** önceden bilirsek.

> *"we can solve single source shortest paths faster if we're given an order of vertices in increasing distance beforehand."* — Ku, 11:22

Neden? Gözlem 1 sayesinde, bu sıraya göre **geriye giden** her (pozitif) kenar en kısa yola katılamaz → atılır. Geriye yalnız ileri kenarlar kalır → bir **DAG** → DAG relaxation (doğrusal). (0-ağırlıklı kenarlar istisna: aynı mesafedeki düğümler arası 0-kenar, sırayı çevirerek düzeltilir; 0-ağırlıklı çevrim tek düğüme **birleştirilir/coalesce**.) Sorun: bu sırayı önceden bilmiyoruz — onu *hesaplamamız* gerek.

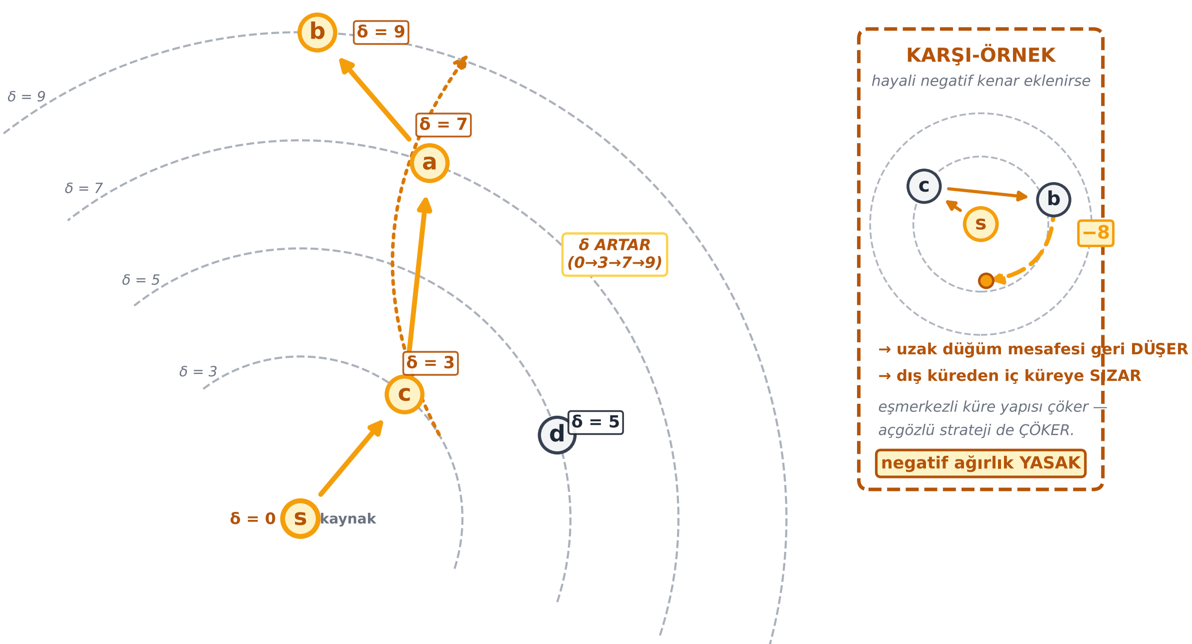
26.5 4. Dijkstra'nın Fikri

Dijkstra (BFS'in ağırlıklı genellemesi; Hollandalı bilgisayar bilimci, "Dijkstra"da J sesi yok — Y'den gelir) iki gözlemi birleştirir:

> *"Relax edges from vertices in increasing distance from source."* — Ku, 23:09

Kenarları, düğümleri **artan mesafe sırasında** alarak gevşet. Ama "sıradaki en yakın düğüm" hangisi? Bunu önceden bilmediğimizden, **kademeli** hesaplarız:

> *"find the next vertex efficiently using a data structure."* — Ku, 23:47



Şekil 26.2: Gözlem 1 (Ku L13 §2): ağırlık ≥ 0 → mesafe en kısa yol BOYUNCA ARTAR — eşmerkezli mesafe küreleri sezgisi (L13 §2). SOL: kaynak s merkezde (0,0); yaylar δ = 3, 5, 7, 9 (örnek çizgenin δ değerleri). Her düğüm kendi δ yarıçaplı kürede; en kısa yol s→c→a→b amber, düğüm mesafeleri 0→3→7→9 ARTAN; yol boyunca δ büyür oku. SAĞ — KARŞI-ÖRNEK: aynı küre fikrine b'den iç küreye hayali −8 kenarı eklenirse uzak düğüm mesafesi geri DÜŞER (dış küreden iç küreye SIZAR) → eşmerkezli yapı çöker → açgözlü strateji çöker; negatif ağırlık YASAK gerekçesi. ALT NOT: çıkarılma sırası = artan mesafe s, c, d, a, b; "distances increase along shortest paths" (Ku 8:35). Veri MOTORDAN: dijkstra(s) = {s:0, c:3, d:5, a:7, b:9}; en kısa yol s→c→a→b mesafeleri 0→3→7→9 kesinlikle ARTAN; tüm ağırlıklar ≥ 0.

26.6 5. Değiştirilebilir Öncelik Kuyruğu

Gereken yapı: **değiştirilebilir öncelik kuyruğu (changeable priority queue)** — üç işlem:

> *"changeable priority queue has three operations… build… delete min… Decrease the key."* — Ku, 24:42

- **build(items):** n öğeyle kur.
- **delete_min():** en küçük anahtarlı öğeyi çıkar.
- **decrease_key(id, k):** belirli **id**'li öğenin anahtarını daha küçük bir k'ya düşür.

Fark: her öğenin bir **anahtarı** (key, mesafe tahmini) **ve** benzersiz bir **id**'si (düğüm) vardır. `decrease_key`, id ile öğeyi bulup anahtarını günceller. İmplementasyon: normal bir öncelik kuyruğu (Ders 12) + bir **sözlük** (id → kuyruktaki konum) cross-link. Düğüm id'leri 0 … $V-1$ ise, sözlük yerine **direct access array** → $O(1)$ (hash'in beklenen $O(1)$'i yerine kesin $O(1)$).

Şekil 26.3 imza şekildir: solda binary min-heap ağacı (her düğümde `(key, id)`) + altında id→konum cross-link sözlüğü; ortada `decrease_key(2, 0)` adımı (id 2'nin anahtarı $8 \to 0$, köke sift-up); sağda üç işlem ve maliyetleri. Çapraz bağ, gevşeyen bir düğümü taramadan $O(1)$ konumlandırmayı — dolayısıyla $O(\log n)$ `decrease_key`'i — mümkün kılar.

26.7 6. Dijkstra Algoritması

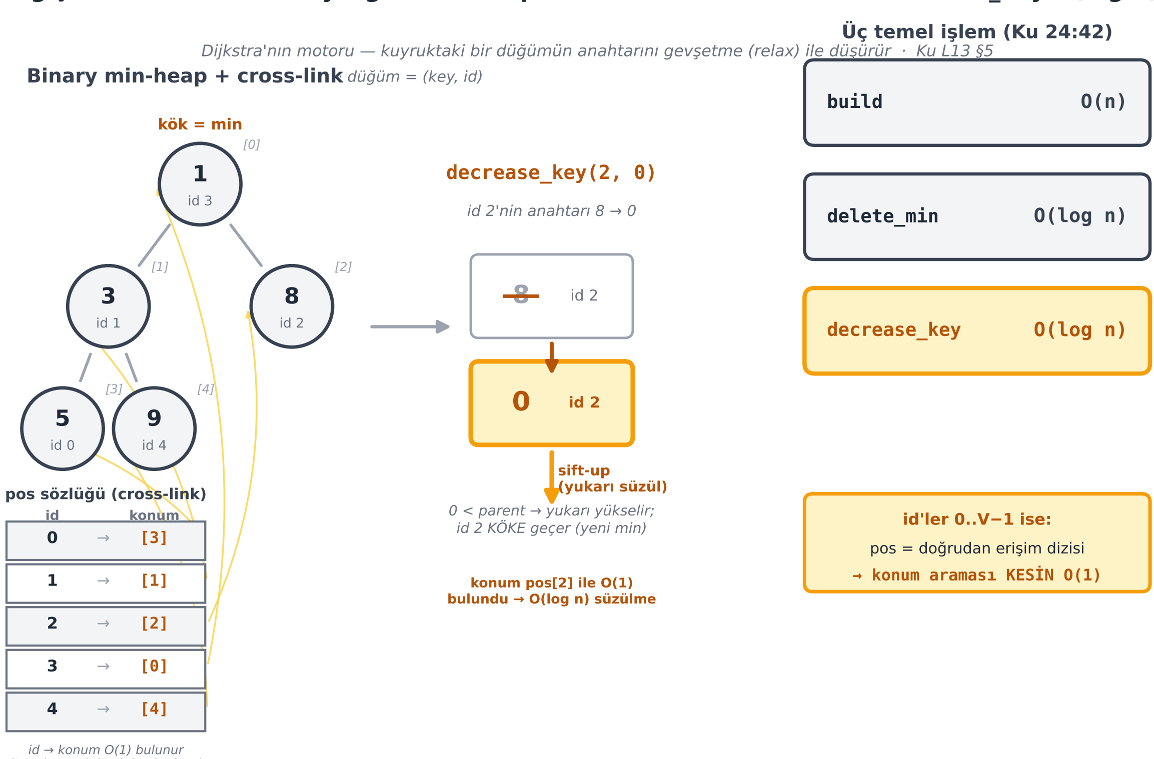
Başlat: $d(s, v) = \infty$ tüm v; $d(s, s) = 0$. Kuyruğu kur. Boşalana dek: en küçük tahminli u'yu çıkar, çıkış kenarlarını gevşet (gevşetince `decrease_key` ile kuyruğu güncelle).

> *"Dijkstra… designed this very nice generalization of BFS for weighted graphs."* — Ku, 22:02

```
def dijkstra(adj, weight, s):
    d = {v: float('inf') for v in adj}
    d[s] = 0
    Q = ChangeablePriorityQueue((v, d[v]) for v in adj)   # build
    while Q:                                               # bos degilse
        u = Q.delete_min()                                # en yakin dugum
        for v in adj[u]:
            if d[u] + weight[(u, v)] < d[v]:              # ucgen esitsizligi ihlali
                d[v] = d[u] + weight[(u, v)]              # relax
                Q.decrease_key(v, d[v])                   # kuyrugu guncelle
    return d
```

Çalışılan Örnek. Pozitif ağırlıklı, çevrimli yönlü çizge. $s = 0$ ile başla; s'yi çıkar, $a \to 10$, $c \to 3$ gevşet. Kuyruktaki en küçük: c (3). c'yi çıkar, $a \to 7$ $(3 + 4)$, $b \to 11$, $d \to 5$ gevşet. En küçük: d (5). d'yi çıkar, $b \to 10$ $(5 + 5)$. En küçük: a (7). a'yı çıkar, $b \to 9$ $(7 + 2)$. Sonra b (9). Sonuç: $\delta = \{s{:}0, c{:}3, d{:}5, a{:}7, b{:}9\}$ — doğru en kısa mesafeler.

Şekil 26.4 bu örneğin tam adım izini gösterir: üstte çalışılan çizge (amber kenarlar = en kısa yol ağacı), altta 5 satırlık tablo. Her satır: `delete_min` edilen düğüm (kesinleşti/donduruldu), gevşetilen kenarlar, kuyrukta kalanlar ve `d` tablosunun anlık durumu. b düğümü $\infty \to 11 \to 10 \to 9$ üç ayrı turda düşer — açgözlü çıkarma sonradan iyileşmeyi engellemez, çıkarılma anında değer kesinleşir.

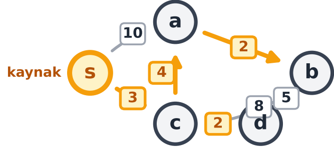


Şekil 26.3: Değiştirilebilir öncelik kuyruğu = binary min-heap + id→konum CROSS-LINK sözlüğü → decrease_key O(log n) (L13 §5 İMZA). SOL: 5 düğümlü min-heap ağacı, her düğümde (key, id) çifti + altında pos sözlüğü tablosu (id → heap-konumu satırları); her satırdan ağaçtaki düğüme ince amber cross-link oku (id → konum O(1); D14 işaretçi dersinin akrabası). ORTA: decrease_key(2, 0) adımı — id 2'nin anahtarı 8 üstü çizili → 0, kökten yukarı sift-up okları; pos[2] ile konum O(1) bulundu → O(log n) süzülme. SAĞ: üç işlem kutusu build O(n) / delete_min O(log n) / decrease_key O(log n) (Ku 24:42) + id'ler 0..V−1 ise pos = doğrudan erişim dizisi → kesin O(1) notu. Veri MOTORDAN: ChangeablePQ([(0,5),(1,3),(2,8),(3,1),(4,9)]) heapify sonrası a = [(1,3),(3,1),(8,2),(5,0),(9,4)], is_valid_heap True; decrease_key(2,0) sonra delete_min() == (2,0) (id 2 köke geçti).

Dijkstra adım izi: en yakını çıkar · gevşet · decrease_key — çıkarma sırası = artan mesafe

çalışılan örnek (Ku L13 §6) · ChangeablePQ (delete_min / decrease_key, O((V+E) log V))

Çalışılan örnek (w: E → ℤ≥0, kaynak s)



amber kenarlar = en kısa yol ağacı (gevşetmeyle d'yi belirleyen)

	delete_min	gevşetilen kenarlar (v: eski→yeni)	kuyruk kalan	d tablosu (anlık) s	a	b	c	d
#1	s d=0 kesinleşti · donduruldu	a: ∞→10, c: ∞→3	c, d, a, b	0	10	∞	3	∞
#2	c d=3 kesinleşti · donduruldu	a: 10→7, b: ∞→11, d: ∞→5	d, a, b	0	7	11	3	5
#3	d d=5 kesinleşti · donduruldu	b: 11→10	a, b	0	7	10	3	5
#4	a d=7 kesinleşti · donduruldu	b: 10→9	b	0	7	9	3	5
#5	b d=9 kesinleşti · donduruldu	— (gevşetme yok)	∅ (boş)	0	7	9	3	5

Gözlem B: her düğüm BİR kez delete_min edilir, sonra dondurulur — bir daha dokunulmaz (ağırlık ≥ 0 → mesafe yol boyunca ARTAR).

Çıkarılma sırası = ARTAN mesafe: s(0) → c(3) → d(5) → a(7) → b(9) (motor order ASSERT). b: ∞→11→10→9 üç turda düşer, son delete_min'de 9.

Şekil 26.4: Dijkstra adım izi: en yakını çıkar · gevşet · decrease_key — çıkarma sırası = ARTAN mesafe (L13 §6 İMZA). ÜST: çalışılan örnek çizge (build_dijkstra_example) sabit yerleşim; düğüm = daire, kenar = yönlü ok + ağırlık rozeti; amber kenarlar = en kısa yol ağacı (gevşetmeyle d'yi belirleyen: s→c, c→a, c→d, a→b). ALT: 5 satırlık adım tablosu (motor order = artan mesafe). Her satır: delete_min edilen düğüm (amber daire + d-değeri + kesinleşti·donduruldu rozeti) · o turda gevşetilen kenarlar (v: eski→yeni, b sütunu ∞→11→10→9 amber vurgu) · kuyruk kalan mini-kutusu · d tablosunun anlık durumu. ALT NOT (Gözlem B): her düğüm BİR kez çıkar, sonra dondurulur — bir daha dokunulmaz; çıkarılma sırası s(0)→c(3)→d(5)→a(7)→b(9). Veri MOTORDAN: dijkstra_trace order = [(s,0),(c,3),(d,5),(a,7),(b,9)]; c relaxed [(a,10,7),(b,∞,11),(d,∞,5)], d [(b,11,10)], a [(b,10,9)]; trace.d == dijkstra == {s:0,a:7,b:9,c:3,d:5}; b evrimi ∞→11→10→9.

26.8 7. Doğruluk: İki Gözlem

İddia: algoritma sonunda tüm v için $d(s,v)=\delta(s,v)$. İki gözleme dayanır:

- **Gözlem A:** gevşetme bir kez d'yi gerçek δ'ya eşitlerse, sonda da öyle kalır. Çünkü gevşetme yalnız **düşürür** ve **güvenlidir** (relaxation safe, Ders 16): tahmin daima gerçek bir yolun ağırlığı veya ∞ — en kısa mesafenin altına inemez.

  "*relaxation is safe.*" — Ku, 41:57

- **Gözlem B:** bir düğüm Q'dan **çıkarıldığında** tahmini zaten δ'dır. Tüm düğümler çıkarıldığından, hepsi doğru olur.

  "*it suffices to show that my estimate equals the shortest distance when v is removed from the Q.*" — Ku, 43:02

Çalışılan Örnek — kanıt (tümevarım). Q'dan çıkarılan ilk k düğüm üzerinde tümevarım. Temel ($k=1$): ilk çıkan s, $d(s)=0=\delta$. Adım: v', k'. çıkan düğüm; $s\to v'$ en kısa yolunda, Q'dan henüz çıkmamış ilk düğüm y, öncülü x ise — x çıkmış olduğundan (tümevarım) $d(x)=\delta(x)$, ve y, x'in en kısa yolundaki ardılı olduğundan $d(y)=\delta(y)$. Negatif olmayan ağırlık → $\delta(y)\le\delta(v')$. Güvenlilik → $d(v')\ge\delta(v')$. Ve v' minimum çıkarıldığından $d(v')\le d(y)=\delta(y)\le\delta(v')$. Tüm eşitsizlikler eşitliğe sıkışır → $d(v')=\delta(v')$. ✓ (Burada **negatif olmayan ağırlık şart** — Gözlem 1'in özü.)

Şekil 26.5 bu zinciri görselleştirir: üstte $s \ldots x \to y \ldots v'$ en kısa yol şeması (çıkmış düğümler dolu, kuyruktakiler beyaz), altta sıkışan eşitsizlik zinciri $\delta(v')\le d(v')\le d(y)=\delta(y)\le\delta(v')$ ve her bağın gerekçesi. Son bağ ($\delta(y)\le\delta(v')$) **ağırlık** ≥ 0'a dayanır — negatif olsaydı tam burada kırılırdı. Motor, bağımsız bir çapraz kanıt olarak Dijkstra ile Bellman-Ford'un aynı örnekte birebir aynı δ'yı verdiğini doğrular.

26.9 8. Çalışma Süresi: Öncelik Kuyruğu Seçimi

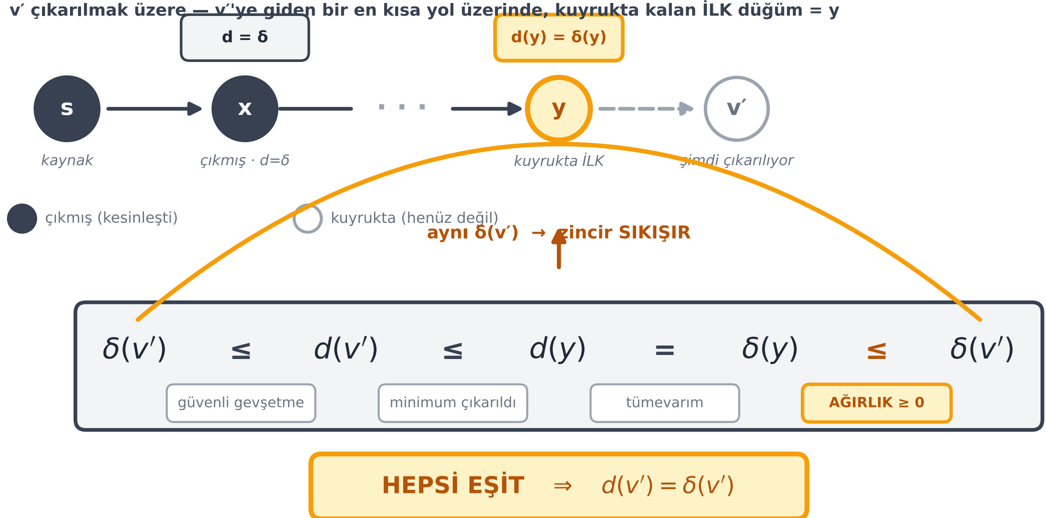
Algoritma her şeyi kuyruk işlemleriyle yapar: 1 build (B), V delete_min (M her biri), E decrease_key (D her biri):

$$T = B + V\cdot M + E\cdot D$$

Öncelik kuyruğu seçimi sonucu belirler:

- **Array (direct access):** delete_min $O(V)$ (tüm diziyi tara), decrease_key $O(1)$ → $O(V^2)$. Yoğun çizgede ($E\sim V^2$) doğrusal — basit ve iyi.
- **Binary heap (ikili yığın):** delete_min $O(\log V)$, decrease_key $O(\log V)$ (ebeveynle yukarı swap) → $O((V+E)\log V)=O(E\log V)$. Seyrek çizgede iyi.
- **Fibonacci heap:** $O(E+V\log V)$ — hem seyrek hem yoğunda en iyi teorik sınır.

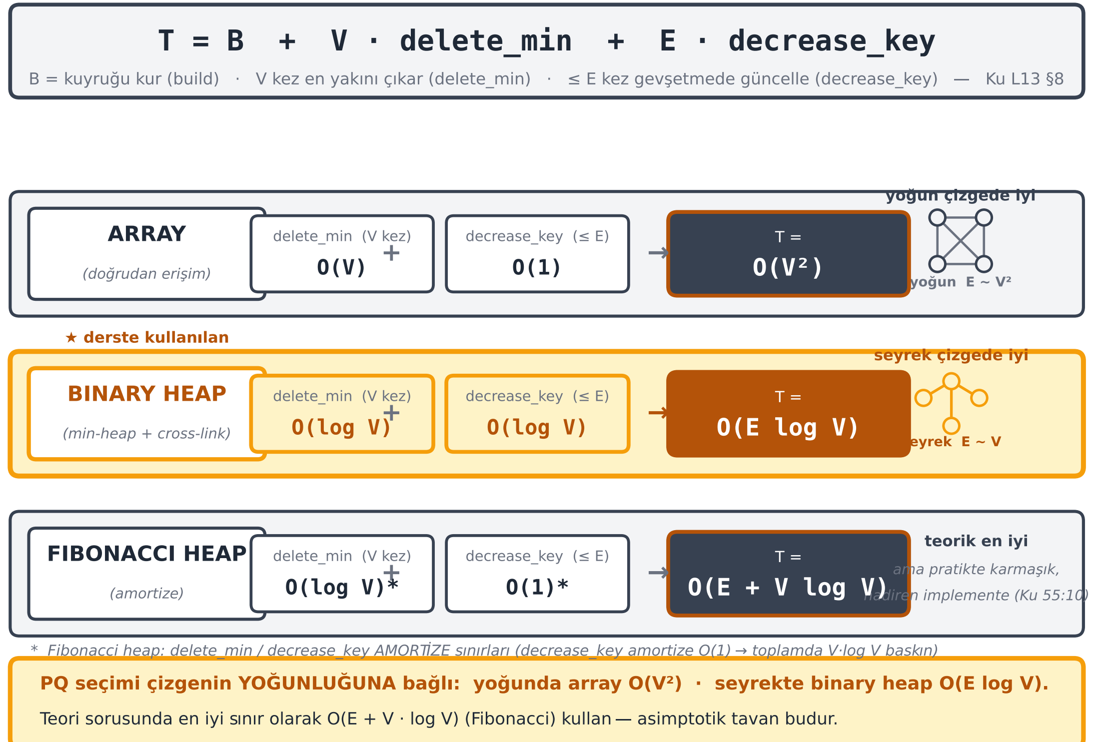
  "*This data structure is called the Fibonacci heap.*" — Ku, 55:10



Şekil 26.5: Dijkstra doğruluğu: SIKIŞAN eşitsizlik zinciri → d(v) = δ(v) (L13 §7 İMZA). ÜST ŞERİT: en kısa yol şeması s … x (çıkmış, d=δ rozeti) → y (kuyrukta İLK, d(y)=δ(y)) … v (şimdi çıkarılıyor); çıkmış düğümler slate DOLU, kuyruktakiler BEYAZ. ALT: eşitsizlik zinciri büyük kutu δ(v) ≤ d(v) ≤ d(y) = δ(y) ≤ δ(v), her bağın altında gerekçe — güvenli gevşetme / minimum çıkarıldı / tümevarım / AĞIRLIK ≥ 0 (sonuncusu amber: negatifte tam burada çöker); sıkışma çemberi en sol ve en sağ δ(v) aynı → HEPSİ EŞİT → d(v) = δ(v). ALT NOT (Ku 43:02 + 41:57): son bağ y'nin v yolunun ön-eki olmasına dayanır (ön-ek ≤ tam yol, ağırlık negatif değilse); negatifte δ(y) > δ(v) olabilir → zincir KIRILIR. Bağımsız çapraz kanıt — motor iki algoritmayı da çalıştırdı: dijkstra == bellman_ford == {s:0, c:3, d:5, a:7, b:9} BİREBİR.

Pratik: Fibonacci heap karmaşıktır, nadiren implemente edilir; çizgenin seyrek/yoğun olduğu bilinirse heap/array yeter. Teori sorusunda $O(E + V \log V)$ sınırını kullan.

Şekil 26.6 bu üç seçimi tek formül üzerinden karşılaştırır: $T = B + V \cdot \text{delete_min} + E \cdot \text{decrease_key}$ formülüne array / binary heap (amber: derste kullanılan) / Fibonacci heap takılır → $O(V^2)$ / $O(E \log V)$ / $O(E + V \log V)$. Seçim çizgenin yoğunluğuna bağlıdır; teori sorusunda en iyi sınır Fibonacci'nin $O(E + V \log V)$'sidir.



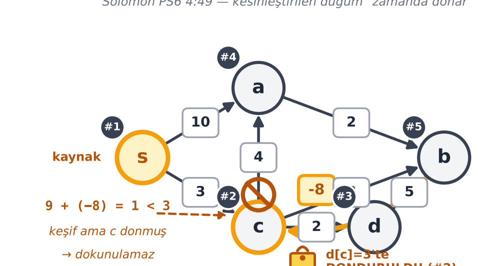
Şekil 26.6: PQ seçimi Dijkstra'nın çalışma süresini belirler: T = B + V·delete_min + E·decrease_key (L13 §8 İMZA). ÜST: genel formül panosu (Ku) — B = kuyruğu kur, V kez delete_min, ≤ E kez decrease_key. 3 PQ satırı: (1) ARRAY (doğrudan erişim) delete_min O(V) / decrease_key O(1) → T = O(V^2), yoğun çizgede iyi; (2) BINARY HEAP (amber, derste kullanılan, min-heap + cross-link) ikisi de O(log V) → T = O(E log V), seyrek çizgede iyi; (3) FIBONACCI HEAP (amortize) delete_min O(log V)* / decrease_key O(1)* → T = O(E + V log V), teorik en iyi ama pratikte karmaşık nadiren implemente (Ku 55:10). ALT ŞERİT: PQ seçimi çizgenin YOĞUNLUĞUNA bağlı (yoğunda array O(V^2), seyrekte binary heap O(E log V)); teori sorusunda en iyi sınır O(E + V log V) kullan. Veri MOTORDAN: dijkstra(s) = {s:0, c:3, d:5, a:7, b:9}; ChangeablePQ = binary heap is_valid_heap True; delete_min sırası ARTAN mesafe s(0) c(3) d(5) a(7) b(9) (Gözlem B).

Şekil 26.7 Dijkstra'nın dayandığı kabulü sentezler: ağırlık ≥ 0 kalktığında ne olur. Örnek çizgeye negatif bir $b \to c$ (-8) kenarı eklenir (negatif çevrim $b \to c \to d \to b = -1$). "Kesinleştir, bir daha dokunma" Dijkstra (c'yi 3'te dondurur) bu kenarı görmez ve orijinal sonlu yanıtı verir — **yanlış**; oysa Bellman-Ford negatif çevrimden erişilen herkesi $-\infty$ işaretler. Bu, **Ders 20'deki PS6 (Solomon) Problem 1**'in köprüsüdür

— orada bu çizge elle adım adım işlenir.

Negatif kenar Dijkstra'yı KIRAR — "kesinleştir, bir daha dokunma" varsayımı çöker

örnek çizgeye b→c(−8) eklendi · Ku L13 sentez + PS6 Problem 1 köprüsü



Aynı çizge — iki algoritma, FARKLI yanıt

düğüm	Dijkstra	Bellman-Ford
s	0	0
a	7	−∞
b	9	−∞
c	3	−∞
d	5	−∞

= fark (Dijkstra yanlış) · s hariç tüm düğümler

Ağırlık ≥ 0 varsayımı KIRILINCA Gözlem 1 (mesafe yol boyunca artar) düşer → kesinleştirme yanlış: donmuş düğüme daha iyi yol

sonradan bulunsa da güncellenmez. Çözüm: Bellman-Ford (Ders 18) — negatif kenar + çevrimi işler. PS6 Problem 1 bunu

elle adım adım çalıştırır (Ders 20 teaser). Motor: dijkstra(finalize) ≠ bellman_ford_classic, fark = {a, b, c, d}.

Şekil 26.7: Negatif kenar Dijkstra'yı KIRAR — "kesinleştir, bir daha dokunma" varsayımı çöker (L13 sentez + PS6 köprüsü). SOL: örnek çizge + eklenen b→c(−8) kalın kırık amber kenar; Dijkstra işleme sırası rozetleri (#1..#5); c "d[c]=3'te DONDURULDU (#2)" kilit ikonu; b işlenince (#5) b→c üzerinden 9 + (−8) = 1 < 3 keşfi (kesik amber ok) ama c donmuş → dokunulamaz (YASAK işareti); negatif çevrim b→c→d→b = −8+2+5 = −1; Solomon PS6 4:49 "zamanda donar" notu. SAĞ: karşılaştırma tablosu düğüm / Dijkstra / Bellman-Ford (motor değerleri BİREBİR); fark hücreleri amber (s hariç tüm düğümler). ALT NOT: ağırlık ≥ 0 varsayımı kırılınca Gözlem 1 düşer → kesinleştirme yanlış; çözüm Bellman-Ford (Ders 18); PS6 Problem 1 elle işler (Ders 20 teaser). Veri MOTORDAN: dijkstra(finalize) = {s:0,a:7,b:9,c:3,d:5} (YANLIŞ), bellman_ford_classic = {s:0, a,b,c,d:$-\infty$} (DOĞRU); fark = {a,b,c,d}; donmuş keşif 9+(−8)=1<3; çevrim = −1.

26.10 Bu Dersin Özeti

1. **Dijkstra**: negatif olmayan ağırlıklı SSSP; BFS'in genellemesi; ~doğrusal.
2. **Gözlem 1**: ağırlık ≥ 0 → mesafe en kısa yol boyunca (zayıf) artar.
3. **Gözlem 2**: artan sıra bilinirse → DAG relaxation (0-ağırlık birleştir/çevir).
4. **Değiştirilebilir öncelik kuyruğu**: build / delete_min / decrease_key (id + key, cross-link sözlük).
5. **Algoritma**: $d = \infty$, $d(s) = 0$; en yakını çıkar, kenarları gevşet, `decrease_key` ile güncelle.
6. **Doğruluk**: gevşetme güvenli + Q'dan çıkınca δ; negatif olmayan ağırlık şart.
7. **Süre**: $B + V \cdot M + E \cdot D$ → array $O(V^2)$ / heap $O(E \log V)$ / Fibonacci $O(E + V \log V)$.

Tek Bir Cümle

Dijkstra, negatif olmayan ağırlıkta "mesafe en kısa yol boyunca artar" gözlemini kullanır: düğümleri bir öncelik kuyruğundan artan mesafe sırasında çeker, kenarlarını gevşetir, böylece SSSP'yi $O(E + V \log V)$'de — BFS'in ağırlıklı genellemesi olarak — çözer.

26.11 Kontrol Soruları

Soru 1: Dijkstra neden negatif ağırlıklarda çalışmaz? Hangi gözlem çöker?

Cevap: Dijkstra, **Gözlem 1**'e (ağırlık $\geq 0 \rightarrow$ mesafe en kısa yol boyunca artar) dayanır; bu sayede "en yakın düğümü kesinleştir, bir daha dokunma" açgözlü stratejisi doğrudur. Negatif ağırlık olsaydı, çok uzaktaki bir düğüm negatif bir kenarla beklenenden daha kısa bir yola sahip olabilirdi — yani Q'dan erken çıkarılan bir düğümün mesafesi sonradan daha da kısalabilirdi, kesinleştirme yanlış olurdu. Doğruluk ispatındaki "[illegible]" adımı [illegible] ağırlıkta bu eşitsizlik bozulur. (Negatif için Bellman-Ford gerekir.)

Soru 2: "Değiştirilebilir öncelik kuyruğu" normal öncelik kuyruğundan nasıl farklı, neden gerekli?

Cevap: Normal öncelik kuyruğu yalnız build ve delete_min sunar; bir öğenin anahtarını *sonradan* değiştiremezsin. Dijkstra'da bir düğümün mesafe tahmini gevşetmeyle düşer — bu yüzden **decrease_key(id, k)** gerekir: belirli bir düğümü (id) bulup anahtarını (key) düşürmek. İmplementasyon: normal öncelik kuyruğu + bir sözlük [illegible] $O(1)$ bulup [illegible] Düğüm id'leri $0 \ldots |V|-1$ ise sözlük yerine direct access array → kesin $O(1)$.

Soru 3: Dijkstra'nın çalışma süresi neden öncelik kuyruğu seçimine bağlı? Hangi durumda hangisi?

Cevap: Toplam süre $T = B + |V| \cdot$ delete_min $+ |E| \cdot$ decrease_key; delete_min ve decrease_key maliyetleri kuyruk implementasyonuna göre değişir: **Array** → delete_min $O(|V|)$, decrease_key $O(1)$ → $O(|V|^2)$, **yoğun** çizgede ($|E| \sim |V|^2$) en iyi. **Binary heap** → ikisi de $O(\log |V|) \rightarrow O(|E| \log |V|)$, **seyrek** çizgede en iyi. **Fibonacci heap** → $O(|E| + |V| \log |V|)$, her durumda en iyi teorik ama pratikte karmaşık.

Soru 4: Gözlem 2'de "artan mesafe sırası bilinirse DAG relaxation" derken 0-ağırlıklı kenarlar neden sorun, nasıl çözülür?

Cevap: Sıralama, "geriye giden kenar atılır" mantığına dayanır; ama 0-ağırlıklı bir kenar iki düğümü [illegible]

26.12 Egzersizler

Egzersiz 1. Küçük bir pozitif-ağırlıklı çizgede Dijkstra'yı elle çalıştır: her adımda Q'dan çıkan düğümü, gevşetilen kenarları ve güncel d değerlerini yaz.

Egzersiz 2. Değiştirilebilir öncelik kuyruğunu binary heap + direct access array ile Python'da implemente et (build / delete_min / decrease_key); decrease_key'in $O(\log V)$ olduğunu doğrula.

Egzersiz 3. Aynı çizgede Dijkstra'yı array, binary heap ve (kavramsal) Fibonacci heap sınırlarıyla karşılaştır; çizge seyrek/yoğun iken hangisinin kazandığını göster.

Egzersiz 4. Bir negatif kenarlı küçük çizge bul; Dijkstra'nın neden yanlış sonuç verdiğini, hangi düğümün erken kesinleştiğini adım adım göster.

Egzersiz 5. Dijkstra'ya ebeveyn işaretçisi ekle (gevşetirken $P(v) = u$); algoritma sonunda en kısa yol ağacını yeniden kur.

26.13 Sonraki Ders İçin Hazırlık

Sonraki: Ders 21 (L14) — Tüm-Çiftler En Kısa Yollar (APSP)

Ders 21 (L14): Tüm-Çiftler En Kısa Yollar (APSP) — Jason Ku ile, tek kaynak yerine **tüm düğüm çiftleri** arasında en kısa yol. (Araya **Ders 20 = PS6** problem oturumu girer.) Naif yol: her düğümden Dijkstra/Bellman-Ford. Daha akıllısı **Johnson algoritması**: negatif ağırlıkları, bir potansiyel fonksiyonla negatif-olmayana "yeniden ağırlıklandırıp" Dijkstra'yı V kez çalıştırmak.
Ders 21 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (elle Dijkstra) ve 2 (değiştirilebilir PQ) çöz.
- Üç çalışma süresi sınırını (array/heap/Fibonacci) ve hangi çizgede hangisi olduğunu ezberle.
- Ana cümleyi tekrar oku: *"Negatif yoksa mesafe artar; en yakını çek, gevşet, decrease_key."*

26.14 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Dijkstra kapsamı	Negatif olmayan ağırlıklı SSSP; BFS genellemesi	Böl. 1
Gözlem 1	Ağırlık $\geq 0 \rightarrow$ mesafe en kısa yol boyunca artar	Böl. 2
Gözlem 2	Artan sıra bilinirse $\rightarrow$ DAG relaxation	Böl. 3
Değiştirilebilir PQ	build / delete_min / decrease_key (id + key)	Böl. 5
Dijkstra döngüsü	En yakını çıkar $\rightarrow$ kenarları gevşet $\rightarrow$ decrease_key	Böl. 6

Kavram	Tanım	Sayfada
Doğruluk	Gevşetme güvenli + Q'dan çıkınca δ; negatif yok şart	Böl. 7
Array PQ	$O(V^2)$; yoğun çizgede iyi	Böl. 8
Heap / Fibonacci	$O(E \log V)$ / $O(E + V \log V)$	Böl. 8

26.15 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, "negatif yoksa mesafe artar" gözlemiyle açgözlü SSSP'yi kurar ve değiştirilebilir öncelik kuyruğu ile neredeyse doğrusal süreye iner — köprülerin özeti:

1. **Dijkstra** → GPS/yönlendirme (Google Maps, Waze), ağ link-state (OSPF), oyun yol bulma.
2. **Sezgisel arama** → Dijkstra + heuristic = `A*`; robot hareket planlama, video oyunu NPC, harita rotalama.
3. **Değiştirilebilir öncelik kuyruğu** → olay simülasyonu (event queue), zamanlayıcı, Prim MST.
4. **Açgözlü + güvenli gevşetme** → açgözlü algoritma kanıt deseni (exchange argument).
5. **PQ veri yapısı seçimi** → seyrek/yoğun bilinciyle karmaşıklık ayarı; gerçek sistemde profil-temelli seçim.
6. **Fibonacci heap** → teorik optimal vs pratik basitlik; "asimptotik en iyi her zaman pratik en iyi değildir".

! Tek bir şey alıp gideceksen

Dijkstra'nın sırrı tek bir gözlemdir — negatif ağırlık yoksa mesafe en kısa yol boyunca artar, dolayısıyla "en yakın düğümü kesinleştir, bir daha dokunma" açgözlü stratejisi doğrudur. Düğümleri bir öncelik kuyruğundan artan mesafe sırasında çekip kenarlarını gevşeterek SSSP'yi neredeyse doğrusal $O(E + V \log V)$'de çözeriz. Bu, BFS'in ağırlıklı dünyaya taşınmış hâlidir.

27 Problem Oturumu 6

Ders 16-19'un uygulaması: beş ağırlıklı en kısa yol problemi indirgemeyle Dijkstra/BFS'e iner — negatif kenarın kırdığı dondurma, Johnson ile ağırlıklı yarıçap, süpernode artı ikili arama, durum makinesi ile graf çoğaltma, darboğaz için modifiye gevşetme

i Oturum bilgisi

- **Solomon'un videosu:** YouTube — Problem Session 6 (≈88 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 6
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 20 (Problem Oturumu 6)
- **Hoca:** Justin Solomon
- **Okuma süresi:** ≈24 dk

27.1 Bu Problem Oturumu Ne Hakkında?

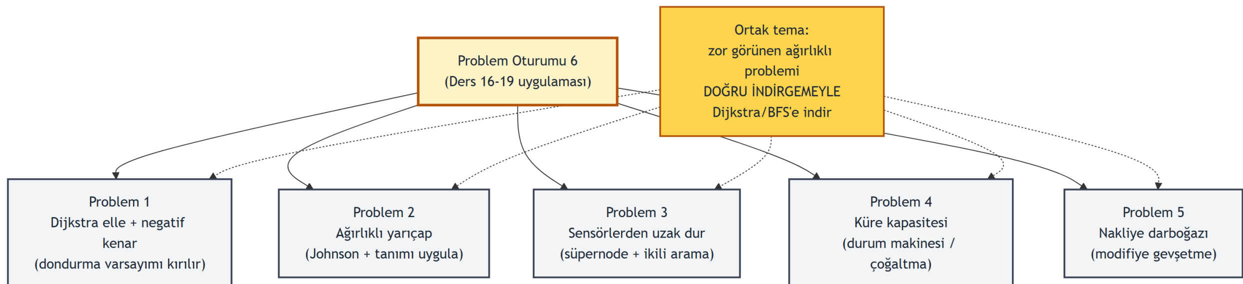
Altıncı problem oturumu (Justin Solomon) **ağırlıklı en kısa yollar** üzerine beş problemi işler (Şekil 39.1): Dijkstra'yı elle çalıştırmak, tüm-çiftler en kısa yol (Johnson) ve birkaç "kılık değiştirmiş" en kısa yol problemi. Tema yine **indirgeme**: problem çoğu zaman doğrudan Dijkstra değil, ama **süpernode**, **graf çoğaltma**, **ikili arama** veya **modifiye gevşetme** ile Dijkstra/BFS'e iner.

> *"if there's an obvious algorithm staring us in the phase to solve a given algorithms problem, we should try that first before we try something more clever."* — Solomon, 15:01

Her problem "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir.

Yaklaşım — ortak strateji: önce bariz olanı dene, sonra indirgemeyi ara

Beş problemin tamamı aynı refleksle başlar: önce "asıl ne soruluyor" ve "hangi sabitler verildi" diye metni soy; eğer ortada bariz bir algoritma (doğrudan Dijkstra, tanımı uygula) varsa onu dene. Bariz değilse problemi **bildiğin bir araca** indirge: süpernode (çok hedef → tek kaynak), graf çoğaltma (konum + durum → katman), cevap-üzerinde ikili arama (log n bütçe → eşik ara) veya modifiye gevşetme (üçgen-eşitsizliği analoğu her güncelleme formülü Dijkstra iskeletine takılır). Bu oturum, ağırlıklı en kısa yol kasını bu beş indirgemeyle çalıştırır.



Şekil 27.1: Problem Oturumu 6'nın kavram haritası: kök (PS6) beş probleme dallanır ve ortadaki ortak tema düğümü beşini birden yönlendirir. Problem 1 Dijkstra'yı elle çalıştırır, sonra negatif bir kenarın dondurma varsayımını nasıl kırdığını gösterir; Problem 2 ağırlıklı yarıçabı Johnson algoritmasını kara kutu olarak çağırıp tanımı doğrudan koda dökerek çözer; Problem 3 sensör problemini bir süpernode artı cevap-üzerinde ikili aramaya indirir; Problem 4 küre kapasitesini durum makinesine çevirip çizgeyi kapasite katmanlarına çoğaltır; Problem 5 darboğaz yolunu toplama yerine min/maks ile gevşeten modifiye bir Dijkstra ile bulur. Ortak tema — zor görünen ağırlıklı problemi doğru indirgemeyle Dijkstra veya BFS'e çevir — Solomon'un her probleme aynı kapıdan girmesini sağlar.

27.2 Problem 1: Dijkstra Elle ve Negatif Kenarın Kırdığı Varsayım

İfade. (a) Verilen pozitif-ağırlıklı çizgede s'den Dijkstra'yı çalıştır (mesafeler + işlem sırası). (b) Bir kenarın ağırlığını **negatif** yap; Dijkstra çalıştırılırsa ne kırılır?

Yaklaşım — dondurma varsayımı: işlenen düğüme bir daha dokunma

Dijkstra her adımda **en yakın işlenmemiş** düğümü çeker, komşularını üçgen eşitsizliğiyle günceller. Kilit özellik: bir düğüm işlenince (kesinleştirilince) **dondurulur** — bir daha dokunulmaz. Bu açgözlü doğruluk, ağırlıkların negatif-olmamasına dayanır: ileride keşfedilen hiçbir yol, çoktan kesinleşmiş bir düğümü kısaltamaz. Negatif bir kenar bu garantiyi yok eder; çözümü görmek için tek negatif kenarlı çizgeyi elle adım adım çalıştırmak yeterlidir.

"once I visit a vertex, I never touch it again. It gets frozen in time." — Solomon, 4:49

Çözüm.

(a) Pozitif ağırlıklarla Dijkstra. Kaynaktan başla, en yakın işlenmemiş düğümü çek, kenarlarını gevşet, dondur; sırayla tüm düğümler kesinleşir. Çıkarılma sırası mesafeye göre artar (Ders 19 Gözlem 1'in sonucu: ağırlık ≥ 0 → mesafe en kısa yol boyunca artar).

(b) Negatif kenar dondurmayı kırar. Çizgeye bir negatif kenar (örneğimizde $b \to c$ ağırlık -8) eklenince, b işlendiğinde $b \to c$ üzerinden $9 + (-8) = 1 < 3$ bulunur — yani c'ye daha kısa bir yol keşfedilir, **ama c çoktan dondurulmuştur**. Dijkstra'nın "işlenen düğüme bir daha dokunma" varsayımı kırılır; c'yi kuyruğa geri eklemek gerekir ama Dijkstra buna izin vermez. (Bu örnekte ek olarak $b \to c \to d \to b = -8+2+5 = -1$ negatif bir çevrim oluşur, bu yüzden çevrime erişilen her düğüm Bellman-Ford'da **eksi sonsuz** olur.)

```
def dijkstra_vs_bf(adj, weight, s):            # dondurma davranışını izle
    d = {v: INF for v in adj}; d[s] = 0
```

```
    Q = ChangeablePQ((v, d[v]) for v in adj)
    done = set(); violations = []
    while len(Q):
        u, _ = Q.delete_min()
        done.add(u)                              # DONDUR
        for v in adj[u]:
            cand = d[u] + weight[(u, v)]
            if v in done:                        # donmuş düğüme daha kısa yol!
                if cand < d[v]:
                    violations.append((u, v, cand, d[v]))
                continue                         # Dijkstra DOKUNMAZ (varsayım)
            if cand < d[v]:
                d[v] = cand; Q.decrease_key(v, d[v])
    return d, bellman_ford_classic(adj, weight, s), ..., violations
```

> *"that breaks our assumption, which is that as soon as I visit a vertex, I never have to touch it again."* — Solomon, 10:59

Şekil 27.2 bu çöküşü motordan **gerçek** verilerle gösterir: örnek çizgeye $b \to c$ (-8) kenarı eklenir; donmuş davranışı taklit eden Dijkstra hâlâ $\{s{:}0,\ a{:}7,\ b{:}9,\ c{:}3,\ d{:}5\}$ (negatif kenarı yok sayan, **yanlış** sonuç) verirken, ihlal listesi tam olarak $[(b, c, 1, 3)]$'tür — b işlenince $9 + (-8) = 1 < 3 = d[c]$ keşfedilir ama c donmuştur. Bellman-Ford ise negatif çevrimi yakalar: s hariç tüm düğümler eksi sonsuza düşer.

Karmaşıklık. (a) standart Dijkstra. (b) negatif kenar → Dijkstra geçersiz; **Bellman-Ford** gerekir (Ders 18).

27.3 Problem 2: Ağırlıklı Yarıçap ve Johnson

İfade. Ağırlıklı yönlü çizge (negatif ağırlık olabilir, negatif çevrim **yok**). **Ağırlıklı eksantriklik** $\varepsilon(u) = \max_v \delta(u, v)$; **ağırlıklı yarıçap** $= \min_u \varepsilon(u)$. Yarıçapı $O(V^3)$'te bul.

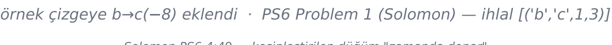
> Yaklaşım — akıllı olma, tanımı doğrudan algoritmaya çevir
>
> Bu, Ders 17 (PS5)'teki yarıçap probleminin ağırlıklı versiyonudur. Burada hile aramaya gerek yok: tanımı doğrudan algoritmaya dök. Eksantriklik için tüm-çiftler mesafe gerekir; bunu **Johnson algoritması** verir (Ders 21'de açılacak; negatif çevrim olmadığından çalışır). Johnson'ı bir kara kutu olarak çağır, sonra iç içe döngüyle eksantriklikleri ve minimumlarını topla. Karmaşıklık bütçesi $O(V^3)$ bu naif yaklaşımı zaten rahatça karşılar.

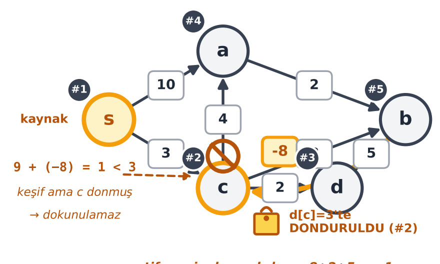
> *"that's called Johnson's algorithm."* — Solomon, 15:49

Çözüm. Üç adım, doğrudan tanımdan:

1. **Johnson** ile tüm $\delta(u, v)$ çiftlerini hesapla — $O(V \cdot E + V^2 \log V) \le O(V^3)$ (basit çizgede $E \le 2V^2$).
2. Her u için iç içe döngüyle $\varepsilon(u) = \max_v \delta(u, v)$ — $O(V^2)$.
3. $\min_u \varepsilon(u)$ — $O(V)$.

Negatif kenar dondurmayı KIRAR — "kesinleştir, bir daha dokunma" varsayımı çöker





Aynı çizge — iki algoritma, FARKLI yanıt

düğüm	Dijkstra	Bellman-Ford
s	0	0
a	7	eksi sonsuz
b	9	eksi sonsuz
c	3	eksi sonsuz
d	5	eksi sonsuz

= fark (Dijkstra yanlış) · s hariç tüm düğümler

Tek negatif kenar HEM dondurma ihlalini doğurur (b→c ile 9−8 = 1 < 3, ama c donmuş → güncellenmez) HEM de (burada)

negatif çevrim yaratır (b→c→d→b = −1) → erişilen her düğüm BF'de eksi sonsuz. Dijkstra'nın açgözlü varsayımı çöker.

Çözüm: Bellman-Ford (Ders 18) — negatif kenarı + çevrimi işler. PS6 Problem 1 bunu elle adım adım çalıştırır.

Şekil 27.2: Negatif kenar dondurmayı KIRAR — Problem 1 İMZA. Örnek çizgeye b→c(−8) eklenir (motordan GERÇEK). SOL: çizge (D19 yerleşimi); b→c kenarı kalın amber kesikli; Dijkstra işleme sıra rozetleri #1..#5; c düğümü ‘d[c]=3’te DONDURULDU (#2)’ kilit ikonu; b işlenince (#5) b→c üzerinden 9+(−8)=1 < 3 keşfi → ihlal okunda YASAK işareti (donmuş düğüme dokunulamaz, Solomon 4:49); negatif çevrim notu b→c→d→b = −1. SAĞ: düğüm / Dijkstra / Bellman-Ford tablosu (motor değerleri BİREBİR) — Dijkstra {s:0,a:7,b:9,c:3,d:5} (negatif kenarı ihmal etti, YANLIŞ), Bellman-Ford s hariç hepsi ‘eksi sonsuz’ (negatif çevrim yakalandı); fark hücreleri amber. Alt not: tek negatif kenar hem dondurma ihlali hem negatif çevrim doğurur; çözüm Bellman-Ford (Ders 18).

İlk terim baskındır, dolayısıyla toplam $O(V^3)$ olur ve problem bütçesiyle uyumludur.

```
def weighted_radius(adj, weight):                # O(V³) — tanımı uygula
    delta = johnson(adj, weight)                 # tüm-çiftler δ (kara kutu)
    if delta is None:
        return None, None                        # negatif çevrim → tanımsız
    best, center = None, None
    for u in adj:                                # her u: ε(u) = maks δ
        ecc = max(delta[u][v] for v in adj)
        if best is None or ecc < best:           # R = min_u ε(u)
            best, center = ecc, u
    return best, center
```

Johnson burada **kara kutu**dur — içinde süpernode + Bellman-Ford potansiyeli ile kenarları negatif-olmayana yeniden ağırlıklandırıp V kez Dijkstra çalıştırır, ama bu problem için önemli olan tek şey çıktısının tüm-çiftler mesafe tablosu olduğudur. Motor üzerinde bir doğrulama: rastgele üretilen 30 negatif-çevrimsiz çizgede `weighted_radius` (Johnson tabanlı) ile her düğümden Bellman-Ford koşan bağımsız bir $V \times$ BF tanığı **birebir** aynı yarıçapı verir — yani indirgemenin doğruluğu farklı bir mekanizmayla teyit edilir.

Karmaşıklık. $O(V^3)$ — yalnızca "tanımı uygula" ile, hiçbir akıllı numara olmadan.

27.4 Problem 3: Atniss ve Sensörler — Süpernode ve İkili Arama

İfade. n kavşaklı boru ağı, her kavşak derecesi ≤ 4 (pozitif uzunluklu borular), bazı kavşaklarda **hareket sensörü**. Atniss, s'den t'ye giderken **herhangi sensöre olan minimum uzaklığı en büyük tutan** yolu ister; $O(n \log n)$'de.

Yaklaşım — kılık değiştirmiş erişilebilirlik: iki katman + cevap-üzerinde ikili arama

Bu doğrudan en kısa yol *değil*; kılık değiştirmiş bir **erişilebilirlik** problemidir. İki katmana ayır: (i) her kavşağın en yakın sensöre uzaklığını bul, (ii) "en az k uzakta kalarak $s \to t$ gidilir mi?" sorusunu k üzerinde **ikili arama** ile çöz. İkinci katmanın ipucu sınıfta yalnız bir algoritmanın $\log n$ sürede koştuğudur: ikili arama. Yanıt monoton (büyük k geçilebilir ise küçük k de geçilir) olduğundan, eşiği sıralı bir dizi indeksinde aramak güvenlidir.

> *"it's sort of a readability problem in disguise."* — Solomon, 24:16

Çözüm. Üç adımda:

1. **Süpernode** x ekle, tüm sensörlere 0-ağırlıklı kenarla bağla; x'ten Dijkstra → her kavşağın en yakın sensör mesafesi ($O(n \log n)$).

   "add a new vertex… and make that vertex distance 0 to every one of my sensors." — Solomon, 31:10

2. G_k = sensör-mesafesi $> k$ olan kavşakların alt-çizgesi; G_k'da BFS, "k yarıçapı dışında $s \to t$ var mı?" sorusunu $O(n)$'de yanıtlar.

3. En büyük $k^\star$'yı bul: kavşakları sensör-mesafesine göre **sırala** ($O(n \log n)$), dizi **indeksi** üzerinde ikili arama ($\log n$ adım $\times$ $O(n)$ BFS).

> *"we only have one algorithm that runs in log n time in this class, which is binary search."* — Solomon, 41:44

```
def max_min_sensor_dist(adj, weight, sensors, s, t):    # O(n log n)
    sd = nearest_sensor_dist(adj, weight, sensors)      # 1. süpernode Dijkstra
    cands = sorted(set(sd.values()))                    # eşik adayları (sıralı)
    lo, hi, best = 0, len(cands) - 1, None
    while lo <= hi:                                      # 3. indeks üzerinde ikili arama
        mid = (lo + hi) // 2; k = cands[mid]
        if survives_threshold(adj, sd, s, t, k - 1):    # 2. G_k'da BFS
            best = k; lo = mid + 1
        else:
            hi = mid - 1
    return best
```

İndirgemenin doğruluğunu küçük bir çizgede bağımsız bir tanık teyit eder: tüm basit yolları DFS ile gezip her yolun minimum sensör-mesafesini alan, sonra bunların maksimumunu döndüren kaba-kuvvet, ikili aramanın bulduğu $k^\star$ ile **aynı** değeri verir.

Karmaşıklık. Dijkstra $O(n \log n)$ + ikili arama $\log n \times O(n)$ = $O(n \log n)$.

27.5 Problem 4: Ashley ve Critter'lar — Graf Çoğaltma ve Durum Makinesi

İfade. n açıklık, her açıklık derecesi $\leq$ 5; her patika (uzunluk l_t, üzerinde c_t critter). Yürüyen kişi her patikadaki tüm critter'ları toplamak **zorunda**; k küre kapasitesi var, mağazalarda boşaltır. Küre yetmezse "üzülür". Üzülmeden en kısa yol; bütçe $O(nk \log nk)$.

> Yaklaşım — durum makinesi: konuma 'kalan kapasite' durumunu ekle, çizgeyi katmanla
>
> Klasik **durum makinesi / graf çoğaltma**: konuma ek olarak "kalan kapasite" durumunu izle. Her fiziksel açıklığı $k+1$ kopyaya çoğalt — her kopya "kaç boş küre kaldı" bilgisini taşır. Bir critter patikası kapasiteyi tüketir; mağaza tam doldurur. Yeterli boş küre olmadan bir patikaya girilemez (o kenar yoktur), bu yüzden "üzülme" durumu otomatik olarak modelin dışında kalır. Bu çoğaltılmış çizgede Dijkstra, hem konumu hem kapasiteyi takip ederek doğru cevabı verir.

> *"it's kind of like a state machine."* — Solomon, 56:42

Çözüm. Her açıklığın $k + 1$ **kopyası**: $v_{c,i}$ = "açıklık c, i boş küre". Kenarlar (her (a, b) patikası, c critter):

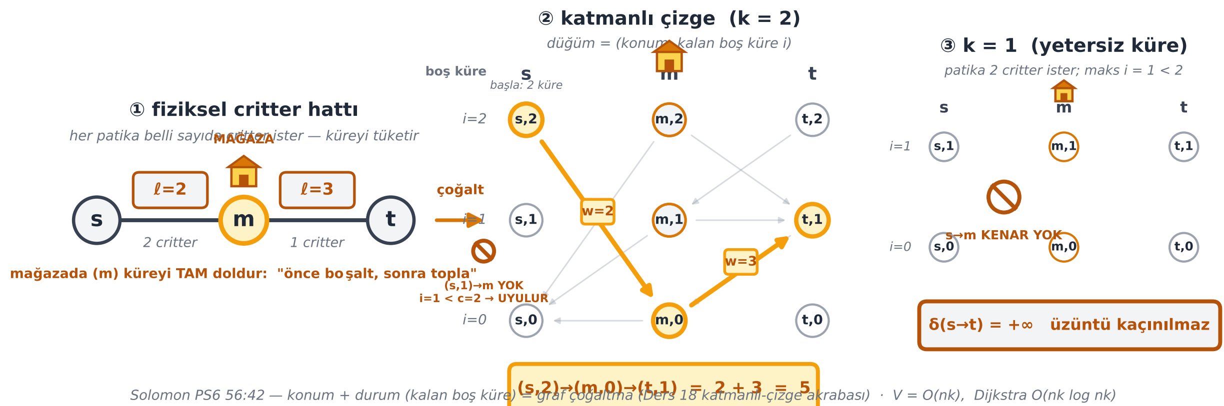
- **Mağaza yoksa:** $v_{a,i} \to v_{b,i-c}$ ($i \geq c$ gereği — yoksa üzülür);
- **Mağaza varsa:** $v_{a,i} \to v_{b,k-c}$ (önce boşalt, sonra topla).

Burada $V = O(kn)$ ve $E = O(kn)$. Dijkstra'yı $v_{s,k}$'dan çalıştır; hedef herhangi $v_{t,i}$. Sonuç ∞ ise "**üzüntü kaçınılmaz**". (Bu çizge DAG **değil** — iki mağaza arası ileri-geri bedava çevrim olabilir, bu yüzden DAG relaxation değil Dijkstra gerekir.)

```
def shortest_no_sadness(paths, stores, k, s, t):         # O(nk log nk)
    adj, weight = build_capacity_graph(paths, stores, k)  # k+1 kapasite katmanı
    d = dijkstra(adj, weight, (s, k))                     # tam küre ile başla
    return min(d[(t, i)] for i in range(k + 1))          # herhangi (t,i) min; ∞ = üzüntü
```

Şekil 27.3 bu çoğaltmayı motordan **gerçek** bir küçük örnekle gösterir: $s \xrightarrow{l=2,\,c=2} m \xrightarrow{l=3,\,c=1} t$, mağaza m, kapasite $k = 2$. Katmanlı çizgede tek geçerli yol $(s, 2) \to (m, 0) \to (t, 1)$ ağırlık $2 + 3 = 5$'tir; m mağazalı olduğu için çıkışta küre tam dolar ($k - c = 2 - 1 = 1$). Kapasite $k = 1$'e düşürülünce $s \to m$ patikası $c = 2$ critter ister ama maksimum $i = 1 < 2$ olduğundan o kenar hiç oluşmaz → sonuç ∞ = **üzüntü kaçınılmaz**.

Karmaşıklık. Dijkstra, $O(kn)$ boyutlu çizgede → $O(nk \log nk)$.



Şekil 27.3: Durum makinesi — konum + kalan küre = düğüm; çizgeyi k+1 katmana çoğalt — Problem 4 İMZA. Üç panel (motordan GERÇEK: paths=[(s,m,2,2),(m,t,3,1)], stores={m}). FİZİKSEL HAT: s –[ℓ=2, 2 critter]– m(MAĞAZA ikonu) –[ℓ=3, 1 critter]– t. KATMANLI ÇİZGE (k=2): 3 sütun (s/m/t) × 3 satır (boş küre i=2/1/0); aktif en kısa yol amber (s,2)→(m,0) w=2 (2 critter toplandı), m MAĞAZALI çıkışta boşalt (m,0)→(t,1) w=3; geçersiz örnek (s,1)→m YOK (i=1 < c=2 → uyulur); yol toplam (s,2)→(m,0)→(t,1) = 2+3 = 5 rozeti. k=1 PANEL: hiçbir s→m kenarı yok (patika 2 critter ister, maks i=1) → δ(s→t)=+∞ üzüntü kaçınılmaz. Alt not: Solomon 56:42, V=O(nk), Dijkstra O(nk log nk).

27.6 Problem 5: Nakliye ve Darboğaz — Modifiye Dijkstra

İfade. n kamyon rotası (kaynak, hedef, **maks ağırlık** w_i, maliyet c_i); yönlü. (1) Gönderilebilen **en ağır** server $w^\star$, (2) o ağırlığı **en az maliyetle** taşıma. ($\leq 3\sqrt{n}$ şehir.)

Yaklaşım — darboğaz dünyasının üçgen eşitsizliği: modifiye gevşetme + iki aşama

Bir yolun **darboğazı (bottleneck)** = yoldaki minimum kenar ağırlığıdır; $B(s,t)$ = tüm π yolları üzerinde darboğazın maksimumu ($\max_\pi$). Anahtar gözlem bir eşitsizliktir: $B(s,t) \geq \min(B(s,v),\ w(v,t))$, bir $v^\star$ için eşitlikle — bu, **darboğaz dünyasının üçgen eşitsizliği**dir. Üçgen-eşitsizliği analoğu olan her güncelleme formülü Dijkstra iskeletine takılabilir: toplama yerine min, en küçüğü kesinleştirme yerine en büyüğü kesinleştirme. Sonra $w^\star$ bilinince problem ikinci aşamada sıradan bir maliyet-Dijkstra'ya iner.

> *"the bottleneck of pi is going to be the minimum edge weight of any edge in pi."* — Solomon, 1:12:24

Çözüm.

(1) En ağır $w^\star$. Darboğaz eşitsizliği, Dijkstra'nın güncelleme formülünün analoğudur: toplama yerine min/maks ile gevşet — gelen komşular arasında en büyük darboğazı seç. **Modifiye Dijkstra** $w^\star$'ı verir.

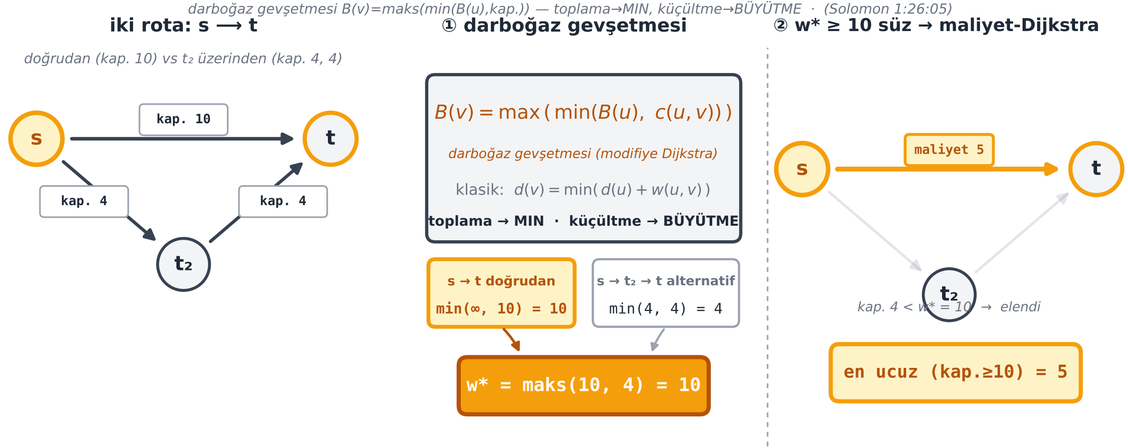
> *"rather than updating by summing edge lengths, we update by using this formula."* — Solomon, 1:26:05

(2) En az maliyet. $w^\star$ bilinince, yalnız $w^\star$ **taşıyabilen** rotaları tut; kenar ağırlığı = maliyet olan bir çizgede Dijkstra → en az maliyet. $V = 3\sqrt{n}$ olduğundan array-PQ Dijkstra $O(V^2)$ = $O(n)$.

```
def widest_path(adj, capacity, s):              # (1) darboğaz-maks
    B = {v: 0 for v in adj}; B[s] = INF         # kaynağın darboğazı sonsuz
    Q = ChangeablePQ((v, -B[v]) for v in adj)    # max-PQ: anahtarları negatif tut
    done = set()
    while len(Q):
        u, _ = Q.delete_min()
        done.add(u)
        for v in adj[u]:
            if v in done:
                continue
            cand = min(B[u], capacity[(u, v)])  # TOPLA değil MIN (darboğaz-üçgeni)
            if cand > B[v]:                     # KÜÇÜLT değil BÜYÜT
                B[v] = cand; Q.decrease_key(v, -cand)
    return B
```

Şekil 27.4 bu iki aşamayı motordan **gerçek** verilerle gösterir: $s \to t$ doğrudan (kapasite 10, maliyet 5) ile $s \to t_2 \to t$ alternatifi (kapasiteler $4, 4$; maliyetler $1, 1$). Aşama 1'de darboğaz gevşetmesi doğrudan rota için $\min(\infty, 10) = 10$, alternatif için $\min(4,4) = 4$ bulur; $\max(10,4) = 10$ → $w^\star = 10$. Aşama 2'de yalnız kapasitesi ≥ 10 olan kenarlar kalır (alternatif rota $4 < 10$ ile elenir), kalan çizgede maliyet-Dijkstra en ucuz taşımayı 5 olarak verir.

Karmaşıklık. (1) modifiye Dijkstra; (2) array-PQ Dijkstra $O(V^2) = O((\sqrt{n})^2) = O(n)$.



Şekil 27.4: Darboğaz = modifiye Dijkstra: gevşetmeyi değiştir, iskeleti koru — Problem 5. İki aşama (motordan GERÇEK). SOL: iki rota s t — doğrudan (kapasite 10, maliyet 5) vs t_2 üzerinden (kapasiteler 4,4; maliyetler 1,1). AŞAMA : darboğaz gevşetme formül kutusu B(v)=maks(min(B(u),kap)) — toplama→MIN, küçültme→BÜYÜTME (klasik d(v)=min(d(u)+w) ile kıyas); rota darboğazları doğrudan min(∞,10)=10 amber-kazanan vs alternatif min(4,4)=4; w=*maks(10,4)=10 rozeti. AŞAMA : w*≥10 süz → alternatif rota (kap 4 < 10) elenir soluk, doğrudan kazanır; kalan çizgede maliyet-Dijkstra → en ucuz=5. Alt not: üçgen-eşitsizliği-analogu her güncelleme formülü Dijkstra iskeletine takılır; V=3√n → array-PQ O(V^2)=O(n) (Solomon 1:26:05).

27.7 Ne Öğrendik?

! Altı Taşınabilir Araç

Bu oturum, Ders 16-19'un ağırlıklı en kısa yol teorisini beş somut problemde uyguladı ve altı taşınabilir araç kazandırdı:

1. **Dijkstra'nın dondurma varsayımı:** negatif kenar bu varsayımı kırar (düğüm geri eklenmeli) — bu yüzden Dijkstra yalnız negatif-olmayan ağırlıkta çalışır; negatif kenarda Bellman-Ford gerekir.
2. **"Önce bariz olanı dene":** yarıçap gibi problemler, akıllı numara yerine tanımı doğrudan kodlamayla (Johnson + iç içe döngü) $O(V^3)$'te çözülür.
3. **Süpernode:** birçok hedefe (sensör) tek 0-ağırlıklı kaynakla bağlanıp tek Dijkstra ile "en yakın hedef" mesafesini bul.
4. **Cevap-üzerinde ikili arama:** "$\log n$ bütçe" → ikili arama ipucu; monoton evet/hayır eşiğini ($k^\star$) sıralı dizi indeksinde ara.
5. **Graf çoğaltma / durum makinesi:** konum + durum (kalan kapasite) → $k+1$ katman; $O(kn)$ çizgede Dijkstra.
6. **Modifiye gevşetme:** üçgen-eşitsizliği analoğu olan her güncelleme formülü (darboğaz) Dijkstra iskeletine takılabilir — toplama yerine min/maks.

27.8 Sonraki

Ders 21 (L14) — Tüm-Çiftler En Kısa Yollar (APSP) ve Johnson

Sırada **Ders 21 (L14): Tüm-Çiftler En Kısa Yollar (APSP)** var — Jason Ku ile, bu oturumda kara kutu olarak kullandığımız **Johnson algoritmasını** açıyoruz: negatif ağırlıklı bir çizgede tüm düğüm çiftleri arası en kısa yol. Püf nokta, bir **potansiyel fonksiyonla** kenarları negatif-olmayana "yeniden ağırlıklandırıp" Dijkstra'yı V kez çalıştırmaktır.

28 Tüm-Çiftler En Kısa Yollar (Johnson)

Her düğüme süpernode'dan en kısa mesafeyi potansiyel atayıp kenarları w = w + h(u) − h(v) ile yeniden ağırlıklandırırsak (üçgen eşitsizliği bunu negatif-olmayan yapar, telescoping en kısa yolları korur), negatif ağırlıklı tüm-çiftler en kısa yolu V×Dijkstra hızında O(V^2 log V + V·E) çözeriz — Johnson yeni bir algoritma değil, Bellman-Ford ile Dijkstra'yı birleştiren akıllı bir indirgemedir; çizge ünitesinin son dersi

> **ℹ Bölüm bilgisi**
>
> - **Ku'nun videosu:** YouTube — Lecture 14: APSP and Johnson (~57 dk)
> - **OCW sayfası:** MIT 6.006 Lecture 14: APSP and Johnson
> - **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 21 (L14)
> - **Hoca:** Jason Ku (tüm-çiftler en kısa yollar; çizge ünitesinin **SON dersi**)
> - **Okuma süresi:** ~27 dk
>
> Bu, **çizge ünitesinin son dersidir**. Tek kaynak yerine *tüm* düğüm çiftleri arasında en kısa yol: **APSP (all-pairs shortest paths)**. Naif yol — her düğümden Bellman-Ford — $O(V^2 \cdot E)$ (yoğun çizgede V^4'e yakın). **Johnson**, negatif ağırlıklı bir çizgeyi akıllıca **yeniden ağırlıklandırıp** Dijkstra'yı V kez çalıştırmayı mümkün kılar → $O(V^2 \log V + V \cdot E)$, neredeyse doğrusal.

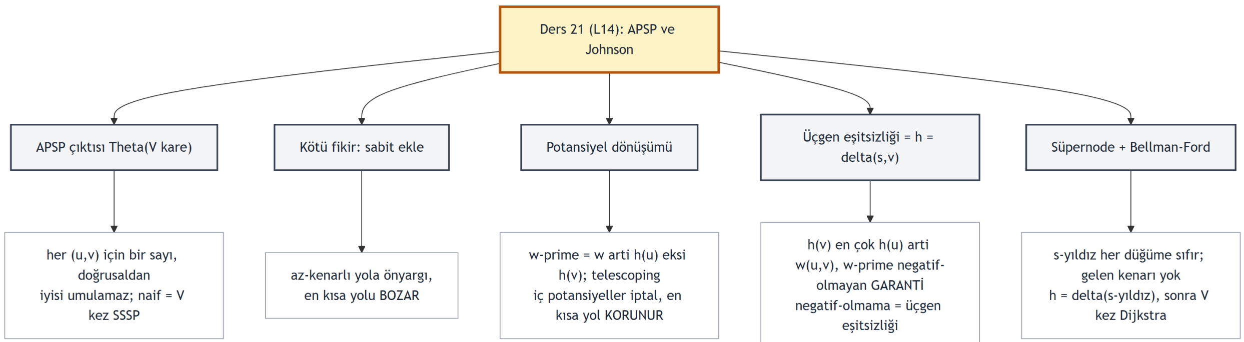
28.1 Bu Derste Ne Var?

Bu, çizge ünitesinin son dersi (Jason Ku). Tek kaynak yerine **tüm düğüm çiftleri** arasında en kısa yol: **APSP**. Naif yol — her düğümden Bellman-Ford — $O(V^2 \cdot E)$ (yoğun çizgede V^4'e yakın). **Johnson algoritması** ise, negatif ağırlıklı bir çizgeyi akıllıca **yeniden ağırlıklandırıp** Dijkstra'yı V kez çalıştırmayı mümkün kılar → $O(V^2 \log V + V \cdot E)$, neredeyse doğrusal.

> *"make edge weights non-negative while preserving shortest paths."* — Ku, 14:35

Üç ana fikir:

1. **APSP çıktısı** $\Theta(V^2)$ — her çift için bir sayı; doğrusaldan iyisini umut edemeyiz, ama V×Dijkstra'yı negatif ağırlıkta da istiyoruz.
2. **Potansiyel ile yeniden ağırlıklandırma** — her düğüme bir potansiyel $h(v)$ atayıp $w'(u, v) = w(u, v) + h(u) - h(v)$; bu **en kısa yolları korur** (telescoping).
3. $h = \delta(s, v)$ — süpernode'dan en kısa mesafeyi potansiyel seçince, üçgen eşitsizliği kenarları **negatif-olmayan** yapar; sonra V×Dijkstra.



Şekil 28.1: Ders 21'in (L14) kavram haritası: kök = APSP ve Johnson (Ku) — çizge ünitesinin son dersi; tüm-çiftler en kısa yol. Beş dal — (1) APSP çıktısı Theta(V kare): her (u,v) çifti için bir sayı, doğrusaldan iyisi umulamaz; naif = V kez SSSP. (2) Kötü fikir: her kenara sabit ekle; az-kenarlı yola önyargı, en kısa yolu bozar. (3) Potansiyel dönüşümü w' = w + h(u) - h(v); telescoping ile iç potansiyeller iptal, en kısa yol korunur. (4) Üçgen eşitsizliği: negatif-olmama koşulu h(v) <= h(u) + w(u,v) tam olarak üçgen eşitsizliği; h = delta(s,v) seçimi w' >= 0 garanti eder. (5) Süpernode: s* her düğüme sıfır ağırlık, gelen kenarı yok; Bellman-Ford ile h, sonra V kez Dijkstra. Sonuç: Johnson yeni algoritma değil, Bellman-Ford ile Dijkstra'yı birleştiren akıllı tutkal; O(V kare log V artı V çarpı E).

Builder Notu — Johnson = Akıllı Tutkal

APSP'nin çıktısı zaten $\Theta(V^2)$ — her çift için bir mesafe. Naif yol her düğümden Bellman-Ford'dur ($O(V^2 \cdot E)$, yoğun çizgede V^4'e yakın). Johnson, negatif ağırlığı bir potansiyelle ortadan kaldırıp Dijkstra'yı V kez çalıştırarak bunu $O(V^2 \log V + V \cdot E)$'ye indirir. Yeni bir kanıt yoktur — ağır iş Bellman-Ford (bir kez) ve Dijkstra (V kez) kara kutularındadır.

- **İleriye → mesafe matrisi:** APSP, yol ağında "tüm-çiftler süre matrisi" önbelleği (rota servisleri, lojistik optimizasyon) demektir.
- **İleriye → fark kısıtları:** potansiyel/yeniden-ağırlıklandırma, fark-kısıt sistemleri (difference constraints) ve doğrusal programlama dualitesiyle akrabadır.
- **İleriye → reduction ustalığı:** Johnson yeni bir algoritma değil — "imzalı problemi negatif-olmayan probleme indirgeyen tutkal"; OMSCS CS 6515 ana refleksi.
- **Geriye → Bellman-Ford + Dijkstra:** Johnson ikisini kara kutu olarak birleştirir.

Tek cümle: *Her düğüme süpernode'dan en kısa mesafeyi potansiyel atayıp kenarları $w' = w + h(u) - h(v)$ ile yeniden ağırlıklandırırsak (üçgen eşitsizliği bunu negatif-olmayan yapar, telescoping en kısa yolları korur), negatif ağırlıklı APSP'yi V×Dijkstra hızında $O(V^2 \log V + V \cdot E)$ çözeriz.*

28.2 1. APSP Problemi ve $\Theta(V^2)$ Çıktısı

APSP: ağırlıklı bir çizgede *her* (u, v) çifti için $\delta(u, v)$ döndür. (Negatif ağırlıklı çevrim varsa **iptal et** — bu derste yok varsayıyoruz, ama negatif kenarlar olabilir.)

> *"the shortest path distance from u to v for every u and v."* — Ku, 1:42

Çıktı boyutu $\Theta(V^2)$ (her çift için bir sayı); dolayısıyla doğrusaldan (çizge boyutu) iyisini umut edemeyiz — en az karesel.

28.3 2. Naif Çözüm: V × SSSP

En basit yol: her düğümden bir SSSP algoritması. Seçeneğe göre:

- **V × Bellman-Ford:** $O(V^2 \cdot E)$ — genel ama yavaş (yoğun çizgede V^4'e yakın).
- **V × Dijkstra** (ağırlık ≥ 0): $O(V^2 \log V + V \cdot E)$ — seyrek çizgede neredeyse doğrusal.

Seyrek çizgede fark: V×Bellman-Ford $\sim V^3$, V×Dijkstra $\sim V^2 \log V$ — tıpkı insertion sort (n^2) ile merge sort ($n \log n$) arasındaki ayrım. **Hedef:** negatif ağırlıklı çizgede de V×Dijkstra hızını elde etmek.

Şekil 28.2 bu manzarayı tek panelde toplar: solda örnek çizge ve 4×4 δ matrisi (motorun Johnson çıktısı; çıktı boyutu $\Theta(V^2) = 16$ hücre), sağda iki naif yöntem — V×Bellman-Ford ($O(V^2 \cdot E)$, yavaş) ile V×Dijkstra (hızlı, ama "$w \geq 0$ şart" kilidiyle). Hedef, en altta: negatif ağırlıkta da V×Dijkstra hızını yakalamak → Johnson.

Şekil 28.2: APSP problemi + naif çözümler (Ku L14 §1-2): çıktı $\Theta(V^2)$; naif = V × SSSP. SOL ÜST: örnek çizge (build_johnson_example) sabit yerleşim; düğüm = daire, kenar = yönlü ok + ağırlık rozeti; negatif kenarlar a→b(−2), c→d(−1) amber vurgulu. SOL ALT: 4×4 δ matrisi (motor johnson çıktısı; satır=kaynak, sütun=hedef; +∞ erişilmez; köşegen 0) + sağında "çıktı boyutu $\Theta(V^2)$ = 16 hücre, doğrusaldan iyisi umulamaz" rozeti. SAĞ: iki naif yöntem — (1) V × Bellman-Ford $O(V^2 \cdot E)$ slate "yoğun graf ~V^4 YAVAŞ"; (2) V × Dijkstra $O(V^2 \log V + V \cdot E)$ amber "HIZLI" ama kilit ikonu "ağırlık ≥ 0 ŞART". ALT NOT: hedef = negatif ağırlıkta da V × Dijkstra hızı → Johnson (make non-negative while preserving, Ku 14:35). Veri MOTORDAN: johnson == brute_apsp (V×BF bağımsız tanık BİREBİR); 4×4 = 16 hücre; δ[a]={a:0,b:−2,c:1,d:0}; δ[d]={a,b,c:+∞,d:0}.

28.4 3. Fikir: Yeniden Ağırlıklandırma

Anahtar fikir: kenar ağırlıklarını **negatif-olmayan** yapacak şekilde değiştir — ama **en kısa yolları koruyarak**.

> *"make edge weights non-negative while preserving shortest paths."* — Ku, 14:35

Başarırsak, yeni çizge G' üzerinde V×Dijkstra çalıştırıp, mesafeleri orijinal G'ye geri çeviririz. (Mümkün değildir eğer G negatif ağırlıklı çevrim içeriyorsa — o zaman en kısa yol basit değildir, oysa negatif-olmayan çizgede en kısa yollar basittir → çelişki.)

28.5 4. Kötü Fikir: Her Kenara Sabit Ekle

Akla gelen ilk şey: en küçük (negatif) ağırlığın tersini her kenara ekle → hepsi negatif-olmayan. **Ama en kısa yolları bozar.**

> *"Makes weights non-negative, but does not preserve shortest paths."* — Ku, 23:45

Neden? Her kenara aynı sabit eklemek, **az kenarlı** yollara önyargı (bias) yaratır: 3-kenarlı bir yol $+3c$, 1-kenarlı yol $+c$ artar. Kenar sayısı farklı yollar farklı değişir → en kısa yol değişebilir.

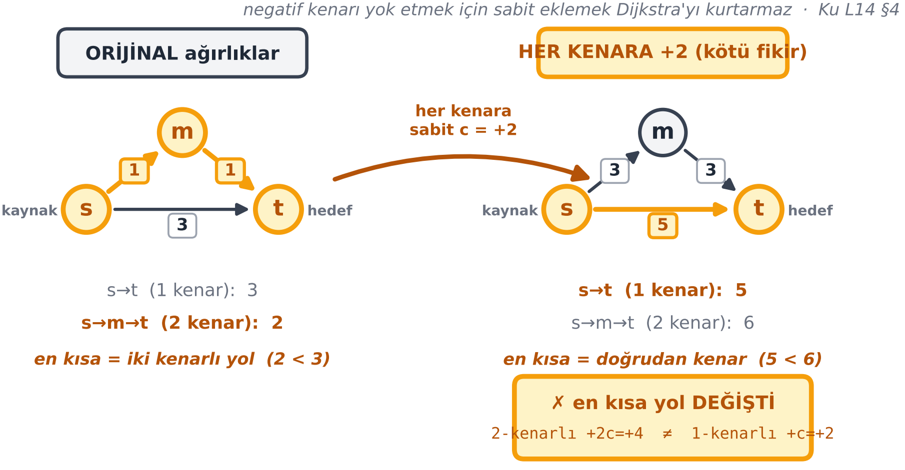
Şekil 28.3 bunu birebir bir karşı-örnekle gösterir: solda orijinal çizge ($s \to t$ doğrudan = 3; $s \to m \to t$ = 2; en kısa = 2-kenarlı yol), sağda her kenara $+2$ eklenmiş hâli (doğrudan = 5; iki-kenarlı = 6; en kısa **değişti** → doğrudan kenar). Sebep: 2-kenarlı yol $+2c$ kayar, 1-kenarlı yol $+c$ kayar — farklı kayma sıralamayı bozar. Doğru dönüşüm (potansiyel) her yolu *aynı* sabitle kaydırmalıdır.

28.6 5. İyi Fikir: Potansiyel Dönüşümü

Daha akıllı dönüşüm: bir düğüm v için, **tüm çıkış kenarlarına h ekle, tüm giriş kenarlarından h çıkar**.

> *"If I add a number to all outgoing edges from a vertex, and I subtract that same number from the weights of all of the incoming edges… preserves shortest paths."* — Ku, 25:20

Çalışılan Örnek — neden korur. Bir yolu düşün: v'den **geçen** her yol, v'ye bir giriş kenarı ($-h$) ve bir çıkış kenarı ($+h$) kullanır → değişim iptal olur (net 0). v'ye **hiç değmeyen** yol etkilenmez. Yalnızca v'nin **başlangıç** (yol $+h$) veya **bitiş** (yol $-h$) olduğu durumda değişir — ama o zaman v'den çıkan/giren *tüm* yollar aynı miktarda değişir → en kısa yol yine en kısa kalır.



Şekil 28.3: KÖTÜ FİKİR (Ku L14 §4): her kenara sabit ekle — kenar sayısına bağlı kayma en kısa yolu DEĞİŞTİRİR. İki panel, aynı üçgen çizge (s, m, t; doğrudan s→t alt kenar). SOL — ORİJİNAL: s→t (1 kenar) = 3, s→m→t (2 kenar) = 2; KAZANAN iki-kenarlı yol amber (2 < 3). SAĞ — HER KENARA +2: s→t = 5, s→m→t = 6; KAZANAN DEĞİŞTİ → doğrudan kenar amber + “× en kısa yol DEĞİŞTİ” uyarı rozeti (2-kenarlı +2c=+4 ≠ 1-kenarlı +c=+2). ORTA kutu: k-kenarlı yol +k·c kayar; farklı kenar sayısı → farklı kayma → sıralama bozulur (Ku 23:45). ALT NOT: doğru dönüşüm her yolu AYNI sabitle kaydırmalı → potansiyel h: w = w + h(u) − h(v) telescoping; sağlama (motor) build_johnson_example reweight w = {ab:0, ac:4, bc:1, bd:1, cd:0} HEPSİ ≥ 0. Veri MOTORDAN (path_weight): orijinal 3 vs 2, +c sonrası 5 vs 6; kayma 1-kenarlı +2, 2-kenarlı +4; kazanan yer değiştirdi.

28.7 6. Potansiyel Fonksiyon ve Telescoping

Bunu her düğüme bağımsız uygula: bir **potansiyel fonksiyonu** $h : V \to \mathbb{Z}$. Yeni çizge G': her kenar $w'(u, v) = w(u, v) + h(u) - h(v)$.

> *"define a potential function h that maps vertices to integers."* — Ku, 31:39

Çalışılan Örnek — telescoping kanıtı. $v_0 \to v_1 \to \ldots \to v_k$ yolunun yeni ağırlığı: $\sum[w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)]$. Ara terimler **teleskoplanır** (her iç düğümün $+h$ ve $-h$'si iptal) → $w'(v_0 \to v_k) = w(v_0 \to v_k) + h(v_0) - h(v_k)$. Yani v_0'dan v_k'ya *her* yol aynı sabit ($h(v_0) - h(v_k)$) kadar değişir → en kısa yol değişmez.

Şekil 28.4 Johnson'ın imza fikrini gösterir: üstte $a \to b \to c$ yolu, her kenarın üstünde dönüşüm açılımı ($w'(a, b) = -2 + 0 - (-2) = 0$, $w'(b, c) = 3 + (-2) - 0 = 1$); ortada telescoping — iki açılım alt alta, iç terimler $-h(b)$ ve $+h(b)$ amber çift-çizgiyle iptal, geriye $w(a \to c) + h(a) - h(c)$ kalır; sağda geçiş düğümü sezgisi ($-h(v) + h(v) = 0$); altta sonuç — her yol aynı sabitle kayar, en kısa yol korunur, üstelik $h = \delta(s^*, v)$ seçimiyle $w' \geq 0$.

28.8 7. Negatif-Olmama Koşulu = Üçgen Eşitsizliği

Hangi h kenarları negatif-olmayan yapar? $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$ koşulunu düzenle:

$$h(v) \leq h(u) + w(u, v)$$

> *"That looks like almost exactly the definition of the triangle inequality."* — Ku, 40:16

Bu, **üçgen eşitsizliğinin** ta kendisi! Demek ki $h(v) = \delta(s, v)$ (bir kaynaktan en kısa mesafe) seçersek — ve bu mesafeler sonluysa — koşul kendiliğinden sağlanır. Reweight sonrası tüm kenarlar negatif-olmayan olur.

Şekil 28.5 bu denkliği üç bölgede gösterir: solda cebirsel türetme ($w'(u, v) \geq 0 \Leftrightarrow w(u, v) + h(u) - h(v) \geq 0 \Leftrightarrow h(v) \leq h(u) + w(u, v)$ = üçgen eşitsizliği), ortada üçgen şeması (s^* tepe, u ve v alt; $h(v) \leq h(u) + w$), sağda 5-kenarlı reweight tablosu (her satırda $w/h(u)/h(v)/w'$; hepsi ≥ 0). Altta: $h = \delta(s^*, \cdot)$ seçersek üçgen eşitsizliği otomatik sağlanır.

28.9 8. Süpernode ile Potansiyeli Hesapla

Sorun: tüm düğümlere erişebilen bir kaynak olmayabilir (çizge bağlı olmayabilir). Çözüm: **süpernode**.

> *"Add new vertex s with 0-weight edge to every vertex."* — Ku, 42:00

Yeni süpernode s'ten her düğüme 0-ağırlıklı kenar ekle (G_s). s'ten Bellman-Ford ile $\delta(s, v)$ hesapla (negatif ağırlık olabileceğinden Dijkstra değil). İki durum: $ (s, v) = $ eksi sonsuz ise **negatif çevrim** vardır (s'in gelen kenarı yok → çevrim s'ten geçemez → orijinal G'de) → **iptal et**. Aksi halde tüm $\delta(s, v)$ sonlu → $h(v) = \delta(s, v)$ geçerli potansiyel.

Şekil 28.6 bu adımı gösterir: solda örnek çizge + üstte süpernode s^*, ondan her düğüme 0-ağırlıklı kesikli kenarlar, "s^*'e gelen kenar yok → yeni çevrim yaratmaz" rozeti, her düğümün altında Bellman-Ford'la

Potansiyel dönüşümü + telescoping: w′(yol) = w(yol) + h(başlangıç) − h(bitiş)

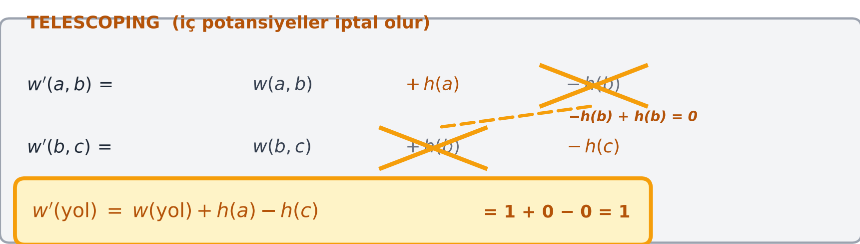
Johnson'ın tutkalı — iç potansiyeller telescoping ile iptal, en kısa yol korunur · Ku L14 §5-6

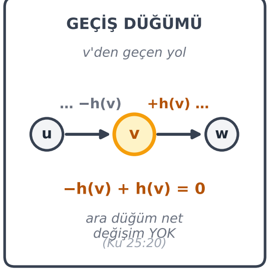
Yol a → b → c (potansiyel dönüşümü altında)

$w'(a,b) = w(a,b) + h(a) - h(b)$ $w'(b,c) = w(b,c) + h(b) - h(c)$

a → b → c

= -2 + 0 − (−2) = 0 = 3 + (−2) − 0 = 1





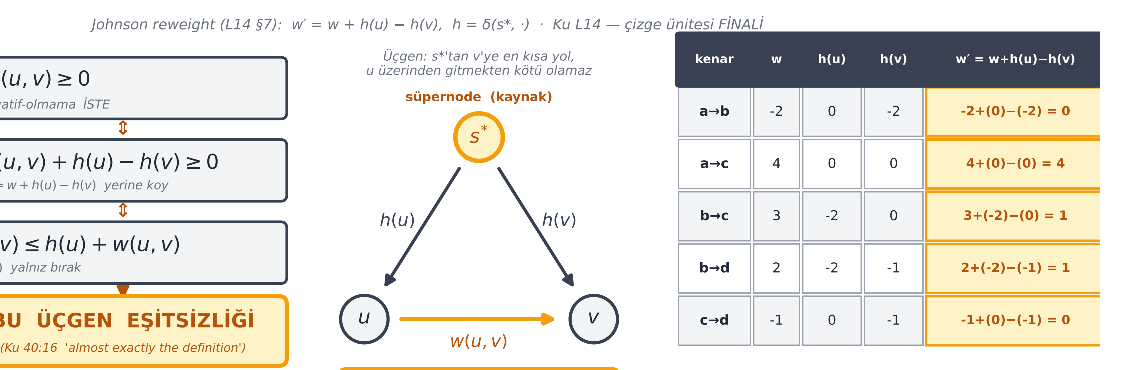
SONUÇ: aynı (u, v) çifti arasındaki HER yol AYNI sabitle h(u) − h(v) kayar

→ ağırlıklar kayar ama SIRALAMA değişmez · en kısa yol KORUNUR · ayrıca h(v)=δ(s*,v) seçimiyle w′ ≥ 0 (üçgen eşitsizliği) → Dijkstra

Motor teyidi: w′ tüm kenarlarda ≥ 0 · Johnson(δ) == brute_apsp(δ) (telescoping en kısa yolu korur — birebir)

Şekil 28.4: Potansiyel dönüşümü + telescoping (Ku L14 §5-6 İMZA): w (yol) = w(yol) + h(başlangıç) − h(bitiş). ÜST: yol a → b → c, her kenar üstünde dönüşüm açılımı w (a,b)=w(a,b)+h(a)−h(b) (= −2+0−(−2) = 0) ve w (b,c)=w(b,c)+h(b)−h(c) (= 3+(−2)−0 = 1). ORTA TELESCOPING: iki açılım alt alta; iç terimler −h(b) (satır 1) ve +h(b) (satır 2) AMBER çift-çizgiyle iptal (−h(b)+h(b)=0); kalan w(yol)+h(a)−h(c) kutuda (= 1+0−0 = 1). SAĞ mini panel: v'den GEÇEN yol için ara düğüm gelen kenar … −h(v), giden kenar +h(v) … → net 0 (Ku 25:20). ALT şerit SONUÇ: aynı (u,v) çifti arasındaki HER yol AYNI sabitle h(u)−h(v) kayar → SIRALAMA değişmez, en kısa yol KORUNUR; ayrıca h(v)=δ(s*,v) seçimiyle w ≥ 0 (üçgen eşitsizliği) → Dijkstra. Veri MOTORDAN: h = {a:0, b:−2, c:0, d:−1}; w = {ab:0, ac:4, bc:1, bd:1, cd:0} HEPSİ ≥ 0; path_weight(wp,[a,b,c]) == path_weight(w,[a,b,c]) + h[a] − h[c] (1 == 1+0−0); johnson == brute_apsp (telescoping en kısa yolu korur).

Negatif-olmama koşulu w′(u,v) ≥ 0 = üçgen eşitsizliği h(v) ≤ h(u) + w(u,v)



kenar	w	h(u)	h(v)	w′ = w+h(u)−h(v)	
a→b	-2	0	-2	-2+(0)−(-2) = 0	✓
a→c	4	0	0	4+(0)−(0) = 4	✓
b→c	3	-2	0	3+(-2)−(0) = 1	✓
b→d	2	-2	-1	2+(-2)−(-1) = 1	✓
c→d	-1	0	-1	-1+(0)−(-1) = 0	✓

TÜM w′ ≥ 0 — negatif-olmama SAĞLANDI

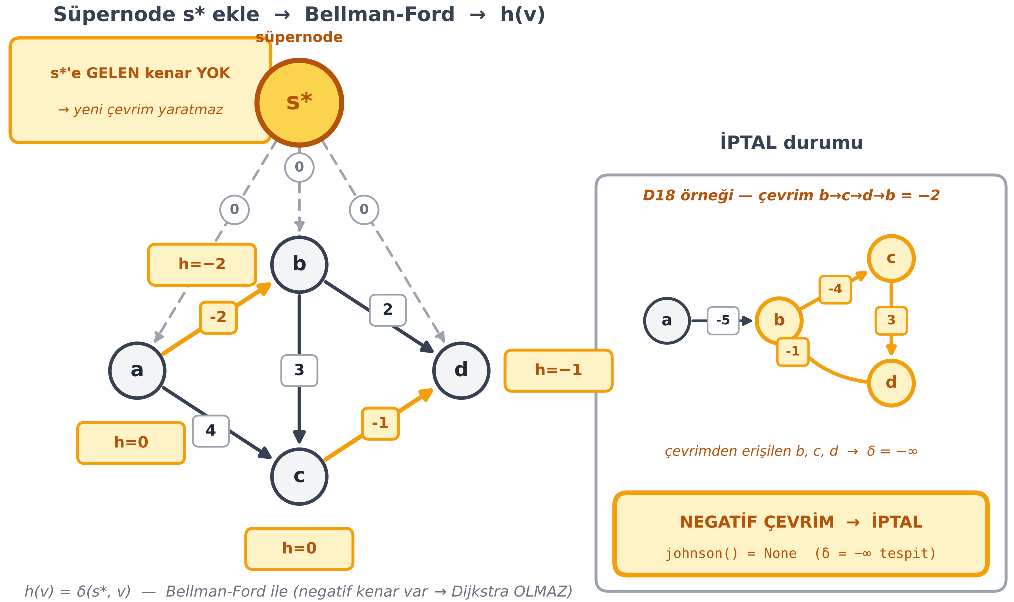
h = δ(s*, ·) en kısa mesafeleri SEÇERSEK, üçgen eşitsizliği OTOMATİK sağlanır → tüm w′ ≥ 0 GARANTİ.

Süpernode s* her düğüme erişir + gelen kenarı yok → yeni negatif çevrim yaratmaz; potansiyel dönüşümü en kısa yolları KORUR.

Motor: johnson(adj, w) == brute_apsp(adj, w) (V×Bellman-Ford bağımsız tanık — BİREBİR, sıfır uyuşmazlık).

Şekil 28.5: Negatif-olmama koşulu w (u,v) ≥ 0 = üçgen eşitsizliği h(v) ≤ h(u) + w(u,v) (Ku L14 §7, 40:16 "almost exactly the definition"). SOL: cebir kutusu adım adım — w (u,v) ≥ 0 İSTE w(u,v)+h(u)−h(v) ≥ 0 h(v) ≤ h(u)+w(u,v); son satır amber "BU ÜÇGEN EŞİTSİZLİĞİ". ORTA: üçgen şeması s* (süpernode/kaynak, tepe) → u (mesafe h(u)) + s* → v (mesafe h(v)) + kenar u→v (ağırlık w, amber kapanış); sonuç rozeti h(v) ≤ h(u) + w(u,v); "s*'tan v'ye en kısa yol u üzerinden gitmekten kötü olamaz". SAĞ: 5 kenar tablosu (a→b, a→c, b→c, b→d, c→d), her satırda w / h(u) / h(v) / w = w+h(u)−h(v) ve ✓ (hepsi ≥ 0); altta "TÜM w ≥ 0 — negatif-olmama SAĞLANDI". ALT NOT:* h = $\delta(s,\cdot)$ en kısa mesafe SEÇERSEK üçgen eşitsizliği OTOMATİK → tüm w ≥ 0 GARANTİ; süpernode s* her düğüme erişir + gelen kenarı yok → yeni negatif çevrim yaratmaz. Veri MOTORDAN: h = {a:0, b:−2, c:0, d:−1}; w = {ab:0, ac:4, bc:1, bd:1, cd:0} HEPSİ ≥ 0; johnson == brute_apsp (V×BF bağımsız tanık BİREBİR).

hesaplanan $h(v)$ (a:0, b:-2, c:0, d:-1); sağda İPTAL durumu — D18'in negatif-çevrimli örneği ($b \to c \to d \to b = -2$), `johnson()` None döner ($ = $ eksi sonsuz tespit).



Şekil 28.6: Johnson §8 (Ku L14, 42:00): süpernode s* + Bellman-Ford → potansiyel h(v) = δ(s*, v)* → *reweight w ≥ 0. SOL: örnek çizge (build_johnson_example) + ÜSTTE süpernode s* (büyük amber); s*'den her düğüme 0-ağırlıklı KESİKLİ kenar ("0" rozetli); "s*'e GELEN kenar YOK → yeni çevrim yaratmaz" amber rozeti; her düğüm altında h = δ(s*, v) rozeti (a:0, b:−2, c:0, d:−1); orijinal negatif kenarlar a→b(−2), c→d(−1) amber;"h Bellman-Ford ile — negatif kenar var → Dijkstra OLMAZ" notu. SAĞ mini panel — İPTAL durumu: D18 örnek çizgesi (çevrim b→c→d→b = −2); s* eklenip Bellman-Ford koşunca çevrimden erişilen b,c,d → δ = −∞ → johnson() = None → "NEGATİF ÇEVRİM → İPTAL" kutusu. ALT NOT: Ku 42:00 "add new vertex with 0-weight edges"; s*'e gelen kenar yok → çevrim yaratmaz; h sonsuz değilse w = w + h(u) − h(v) ≥ 0 GARANTİ (üçgen eşitsizliği) → V × Dijkstra mümkün. Veri MOTORDAN: h = {a:0, b:−2, c:0, d:−1}; w = {ab:0, ac:4, bc:1, bd:1, cd:0} HEPSİ ≥ 0; johnson == brute_apsp; johnson(build_bf_example()) is None (İPTAL); çevrim b→c→d→b ağırlığı = −2.

28.10 9. Johnson Algoritması ve Çalışma Süresi

Johnson, bir **indirgeme** algoritmasıdır: imzalı (signed) APSP → negatif-olmayan APSP.

> *"It's really a reduction problem or a reduction algorithm."* — Ku, 47:08

```
def johnson(graph):
    # 1. Supernode s ekle: s'ten her dugume 0-agirlikli kenar
    Gs = add_supernode(graph)
    # 2. Bellman-Ford ile h(v) = delta(s, v)
    h = bellman_ford(Gs, source=s)          # O(V*E)
    if any(h[v] == float('-inf') for v in graph):
        return "NEGATIF CEVRIM"             # iptal
    # 3. Yeniden agirliklandir: w'(u,v) = w(u,v) + h(u) - h(v)  >= 0
    G_prime = reweight(graph, h)            # O(E)
    # 4. Her dugumden Dijkstra (G' negatif-olmayan)
    delta_prime = {u: dijkstra(G_prime, u) for u in graph}   # V * Dijkstra
    # 5. G mesafelerini geri cevir: delta(u,v) = delta'(u,v) - h(u) + h(v)
    return recover_original_distances(delta_prime, h)        # O(V*(V+E))
```

Çalışma süresi. (1) $O(V+E)$, (2) Bellman-Ford $O(V \cdot E)$, (3) $O(E)$, (4) V×Dijkstra $O(V^2 \log V + V \cdot E)$, (5) $O(V \cdot (V+E))$. Baskın terim (4) → $O(V^2 \log V + V \cdot E)$.

> *"Johnson's is really just glue to transform a graph in a clever way."* — Ku, 56:24

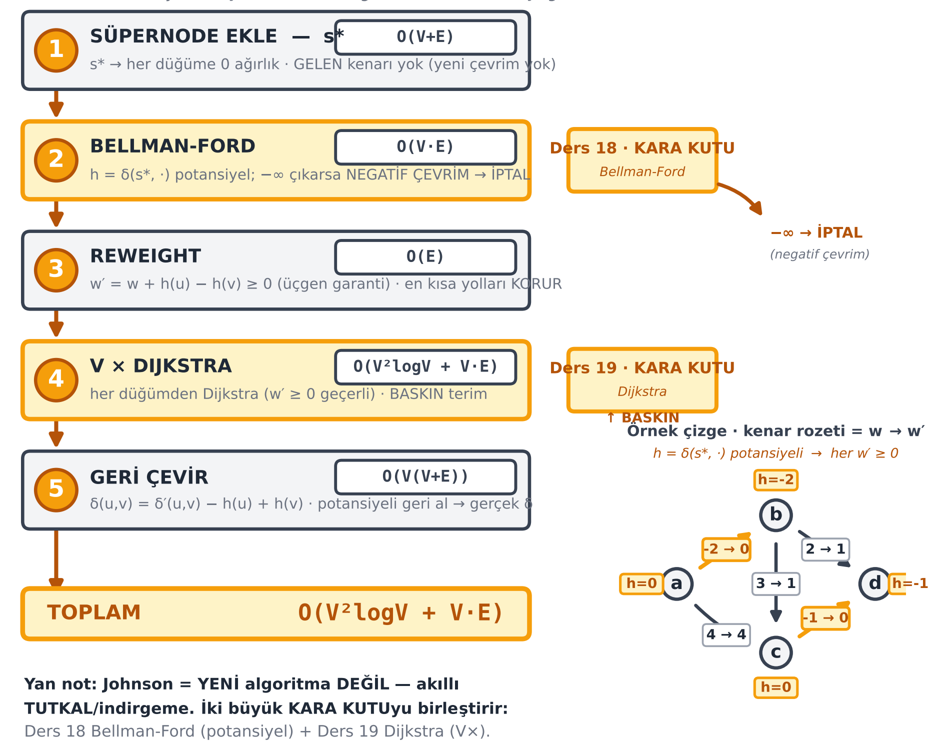
Yeni bir tümevarım/kanıt yok; ağır iş Bellman-Ford (bir kez) + Dijkstra (V kez) kara kutularında. Johnson yalnızca "akıllı tutkal". Negatif ağırlıklı APSP'yi, V×Bellman-Ford'un V^3'ünden çok daha hızlı, seyrek çizgede neredeyse doğrusal çözer.

Şekil 28.7 beş adımı dikey bir akışta toplar: (1) süpernode ekle $O(V+E)$, (2) Bellman-Ford $O(V \cdot E)$ — Ders 18 kara kutusu, eksi sonsuz çıkarsa İPTAL, (3) reweight $O(E)$, (4) V×Dijkstra $O(V^2 \log V + V \cdot E)$ — Ders 19 kara kutusu, BASKIN terim, (5) geri çevir $O(V(V+E))$; toplam $O(V^2 \log V + V \cdot E)$. Sağda örnek çizge ($w \to w'$ rozetleriyle) ve "Johnson = yeni algoritma değil, akıllı tutkal/indirgeme" yan notu.

28.11 Bu Dersin Özeti

1. **APSP**: tüm çift $\delta(u, v)$; çıktı $\Theta(V^2)$; negatif çevrim → iptal.
2. **Naif**: V×Bellman-Ford $O(V^2 \cdot E)$ / V×Dijkstra $O(V^2 \log V + V \cdot E)$.
3. **Yeniden ağırlıklandırma**: kenarları negatif-olmayan yap ama en kısa yolları koru → G''de V×Dijkstra.
4. **Kötü fikir**: sabit ekle → az-kenarlı yollara bias, bozar.
5. **Potansiyel** h: $w' = w + h(u) - h(v)$; telescoping ile en kısa yol korunur.
6. $h(v) = \delta(s, v)$: negatif-olmama koşulu = üçgen eşitsizliği.
7. **Süpernode + Bellman-Ford**: h'yi hesapla, negatif çevrimi yakala; sonra V×Dijkstra → $O(V^2 \log V + V \cdot E)$.

> **! Tek Bir Cümle**
>
> Johnson, süpernode'dan Bellman-Ford ile bulduğu $\delta(s, v)$'yi potansiyel yapıp kenarları $w' = w + h(u) - h(v)$ ile negatif-olmayana çevirir (üçgen eşitsizliği garanti eder, telescoping en kısa yolları korur), sonra V×Dijkstra çağırarak imzalı APSP'yi $O(V^2 \log V + V \cdot E)$'de çözer.



Şekil 28.7: Johnson APSP boru hattı (Ku L14 §9 İMZA): süpernode + Bellman-Ford potansiyeli + V×Dijkstra → O(V^2 log V + V·E). 5 adım dikey akış: (1) SÜPERNODE EKLE — s* → her düğüme 0 ağırlık, GELEN kenarı yok, O(V+E); (2) BELLMAN-FORD — h = δ(s*,·) potansiyeli (Ders 18 KARA KUTU), −∞ çıkarsa NEGATİF ÇEVRİM → İPTAL, O(V·E); (3) REWEIGHT — w = w + h(u) − h(v) ≥ 0 (üçgen garanti, en kısa yolları KORUR), O(E); (4) V × DIJKSTRA — her düğümden Dijkstra (Ders 19 KARA KUTU, w ≥ 0 geçerli), BASKIN terim, O(V^2logV + V·E); (5) GERİ ÇEVİR — δ(u,v) = δ (u,v) − h(u) + h(v), O(V(V+E)). TOPLAM O(V^2logV + V·E). Sağ panel: örnek çizge, kenar rozeti w → w (a→b −2→0, a→c 4→4, b→c 3→1, b→d 2→1, c→d −1→0), düğüm potansiyeli h = {a:0, b:−2, c:0, d:−1}. Yan not: Johnson = YENİ algoritma DEĞİL, akıllı TUTKAL/indirgeme; iki büyük kara kutuyu (Ders 18 + Ders 19) birleştirir; çizge ünitesinin KAPANIŞ dersi (Ku 47:08 + 56:24 "just glue"). Veri MOTORDAN: h ve w EXACT; johnson == brute_apsp (V×BF BAĞIMSIZ tanık); δ(a,d) = 0 = path_weight(w,[a,b,d]).

28.12 Kontrol Soruları

Soru 1: "Her kenara aynı sabit ekle" neden en kısa yolları bozar, ama potansiyel dönüşümü neden bozmaz?

Cevap: Her kenara aynı c eklemek, bir yolun ağırlığını (kenar sayısı)×c kadar artırır — yani farklı kenar sayılı yollar farklı değişir, az-kenarlı yola önyargı doğar; eskiden en kısa olan çok-kenarlı yol artık en kısa olmayabilir. Potansiyel dönüşümünde $w' = w + h(u) - h(v)$; bir yol boyunca ara düğümlerin $+h(u)$ ve $-h(v)$ terimleri **teleskoplanır**, yalnızca $h(v_0) - h(v_k)$ kalır. Yani aynı (v_0, v_k) çifti arasındaki *her* yol aynı sabit kadar değişir — göreli sıralama korunur, en kısa yol yine en kısa kalır.

Soru 2: Potansiyeli neden h(v) = δ(s, v) seçiyoruz? Bu neyi garanti eder?

Cevap: Reweight sonrası kenarların negatif-olmaması için $w(u, v) + h(u) - h(v) \geq 0$, yani $h(v) \leq h(u) + w(u, v)$ gerekir — bu tam olarak üçgen eşitsizliğidir. En kısa yol mesafeleri $\delta(s, \cdot)$ üçgen eşitsizliğini **her zaman** sağlar ($\delta(s, v) \leq \delta(s, u) + w(u, v)$). Dolayısıyla $h(v) = \delta(s, v)$ seçilirse, tüm yeni kenar ağırlıkları otomatik olarak negatif-olmayan olur — başka bir koşul aramaya gerek kalmaz.

Soru 3: Süpernode neden gerekli, ve neden onu eklemek yeni bir negatif çevrim yaratmaz?

Cevap: $h(v) = \delta(s, v)$'yi hesaplamak için *tüm* düğümlere sonlu mesafede ulaşan bir kaynak gerekir; ama orijinal çizge bağlı olmayabilir, hiçbir düğüm hepsine ulaşamayabilir (ulaşılamayan için $ = $ artı sonsuz, işe yaramaz). Süpernode s, her düğüme 0-ağırlıklı kenarla bağlanır → her düğüme sonlu (≤ 0) mesafe. Yeni negatif çevrim yaratmaz çünkü s'in **hiç gelen kenarı yoktur** — hiçbir çevrim s'ten geçemez. Dolayısıyla Bellman-Ford bir eksi sonsuz bulursa, o çevrim zaten orijinal G'de vardı → güvenle iptal edilir.

Soru 4: Johnson neden "yeni bir algoritma değil, tutkal" olarak tanımlanıyor?

Cevap: Johnson kendi başına yeni bir tümevarım/kanıt veya çekirdek hesaplama içermez; ağır işi zaten bildiğimiz iki kara kutu yapar: **Bellman-Ford** (potansiyeli bulmak için bir kez) ve **Dijkstra** (negatif-olmayan G' üzerinde V kez). Johnson'ın katkısı, imzalı (negatif ağırlıklı) problemi, Dijkstra'nın çalışabileceği negatif-olmayan bir probleme **indirgeyen** akıllı yeniden-ağırlıklandırmadır. Yani bir *reduction*: zor bağlamı, hızlı çözebildiğimiz kolay bağlama çeviren tutkal.

28.13 Egzersizler

Egzersiz 1. Küçük bir negatif-kenarlı (çevrimsiz) çizgede süpernode ekle, Bellman-Ford ile $h(v) = \delta(s, v)$ hesapla, kenarları w' ile yeniden ağırlıklandır; tüm $w' \geq 0$ olduğunu doğrula.

Egzersiz 2. Telescoping kanıtını bir yol üzerinde adım adım yaz: ara terimlerin iptal olup $w(v_0 \to v_k) + h(v_0) - h(v_k)$ kaldığını göster.

Egzersiz 3. "Her kenara sabit ekle" fikrinin bir karşı-örneğini kur: iki düğüm arasında farklı kenar sayılı iki yolla, sabit eklemenin en kısayı değiştirdiğini göster.

Egzersiz 4. Johnson'ı Python'da yaz (Bellman-Ford + reweight + V×Dijkstra); çalışma süresinin $O(V^2 \log V + V \cdot E)$ olduğunu argümanla göster.

Egzersiz 5. Süpernode'un gelen kenarı olmamasının neden kritik olduğunu, gelen kenarı olsaydı hangi yanlış sonucun doğabileceğini açıkla.

28.14 Sonraki Ders İçin Hazırlık

Sonraki: Ders 23 (L15) — Dinamik Programlama 1 (Erik Demaine)

Ders 23 (L15): Dinamik Programlama — 1 (Erik Demaine). Araya **Ders 22 (Quiz 2 Gözden Geçirme)** girer; çizge ünitesinin son dersi olan bu Johnson dersini ve önceki ağırlıklı en kısa yol derslerini quizden önce gözden geçirme fırsatıdır.
Yeni bir üniteye geçiyoruz: artık size hazır bir algoritma sunmak yerine, **kendi algoritmanızı tasarlamayı** öğreteceğiz — **dinamik programlama (DP)**. En kısa yolların "optimal alt yapı" ve "örtüşen alt problemler" sezgileri, DP'nin temelini oluşturur.
Ders 23 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (reweight) ve 2 (telescoping) çöz.
- Johnson'ın 5 adımını ve her adımın süresini ezberden anlat.
- Ana cümleyi tekrar oku: *"Süpernode'dan δ(s,v)'yi potansiyel yap, reweight et, V×Dijkstra çalıştır."*

28.15 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
APSP	Tüm çift $\delta(u,v)$; çıktı $\Theta(V^2)$; negatif çevrim → iptal	Böl. 1
Naif çözüm	V×Bellman-Ford $O(V^2 \cdot E)$ / V×Dijkstra $O(V^2 \log V + V \cdot E)$	Böl. 2
Yeniden ağırlıklandırma	Kenarları ≥ 0 yap, en kısa yolları koru	Böl. 3
Sabit ekleme (kötü)	Az-kenarlı yola bias → bozar	Böl. 4
Potansiyel h	$w' = w + h(u) - h(v)$; telescoping korur	Böl. 5-6
$h = \delta(s,v)$	Negatif-olmama = üçgen eşitsizliği	Böl. 7
Süpernode	$s \to$ her düğüm 0-ağırlık; Bellman-Ford; eksi sonsuz → iptal	Böl. 8
Johnson süresi	$O(V^2 \log V + V \cdot E)$ (V×Dijkstra baskın)	Böl. 9

28.16 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, APSP çıktısının zorunlu $\Theta(V^2)$ alt sınırından başlayıp Johnson'ın akıllı indirgemesiyle negatif ağırlıklı tüm-çiftler en kısa yolu neredeyse doğrusala indirir — köprülerin özeti:

1. **APSP** → yol ağı mesafe matrisi (rota servisi önbelleği), lojistik, ağ gecikme matrisi.
2. **Potansiyel / yeniden ağırlıklandırma** → fark-kısıt sistemleri (difference constraints), doğrusal programlama dualitesi.
3. **Üçgen eşitsizliği = negatif-olmama** → metrik gömme, geçerli sezgisel (admissible heuristic) tasarımı.
4. **Süpernode** → çok-kaynak/çok-hedef indirgemesi (akış, kümeleme).
5. **Johnson = reduction** → "zor bağlamı kolay kara kutuya çevir"; OMSCS CS 6515 indirgeme refleksi.
6. **Negatif çevrim tespiti (tek Bellman-Ford)** → erken durma ile büyük iş tasarrufu; arbitraj/tutarsızlık kontrolü.

Tek bir şey alıp gideceksen

Johnson, negatif ağırlıklı tüm-çiftler en kısa yolu, V×Bellman-Ford'un yavaşlığına katlanmadan çözer. Sırrı bir **potansiyel fonksiyonudur**: süpernode'dan Bellman-Ford ile bulunan $\delta(s, v)$'yi her düğüme atayıp kenarları $w' = w+h(u)-h(v)$ ile yeniden ağırlıklandırır. Üçgen eşitsizliği bunu negatif-olmayan yapar, telescoping en kısa yolları korur — böylece hızlı Dijkstra'yı V kez çalıştırabiliriz. Johnson yeni bir algoritma değil; zor problemi kolay kara kutuya çeviren akıllı bir indirgemedir. Bununla **çizge ünitesi kapanır**; sırada algoritma *tasarlamayı* öğreten dinamik programlama var.

29 Quiz 2 Gözden Geçirme

Çizge bloğu toplu tekrarı — modelleme ve indirgeme, çizge algoritma haritası, SSSP hiyerarşisi, Johnson, graf değiştirme stratejileri ve dört gerçek sınav problemi

ℹ Oturum bilgisi

- **Ku'nun videosu:** YouTube — Quiz 2 Review (≈82 dk)
- **OCW sayfası:** MIT 6.006 Quiz 2 Review
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 22 (Quiz 2 Review)
- **Hoca:** Jason Ku
- **Okuma süresi:** ≈28 dk

Bu **normal bir ders değil** — Quiz 2 öncesi **toplu tekrar** oturumudur (çizge bloğu). Kursun ağırlıklı/ağırlıksız çizge derslerini tek çatı altında toparlar ve sınavda nasıl düşünüleceğini öğretir.

29.1 Bu Quiz Review Ne Hakkında?

Bu, Jason Ku ile **Quiz 2 öncesi toplu tekrar** oturumudur. Kursun **çizge bloğunu** (Ders 13-21: iki ağırlıksız + dört ağırlıklı çizge dersi, PS5 + PS6) tek çatı altında toparlar ve **sınavda nasıl düşünüleceğini** öğretir. Quiz 1 materyali "fair game"dir ama vurgu **çizge algoritmalarındadır**. İçerik üç eksende ilerler: (1) çizge algoritma haritası, (2) modelleme + graf-değiştirme stratejileri, (3) dört gerçek sınav problemi çözümü.

> *"there's really a small number of graph algorithms but they can solve a lot of different problems."*
> — Ku, 1:27

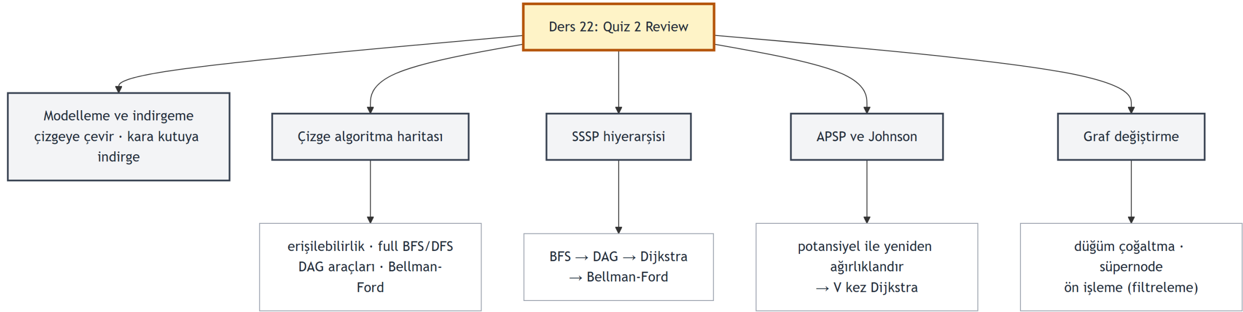
Ku'nun ana mesajı: çizge ünitesi "**indirgeme**" ünitesidir — az sayıda güçlü kara kutu (BFS, DFS, Dijkstra, Bellman-Ford, Johnson) var; iş, problemi doğru bir çizgeye **modelleyip** o kara kutulardan birine indirgemektir.

> *"defining a graph is an important aspect of that problem solving."* — Ku, 9:50

💡 Builder Notu — Bu = Çizge Midterm

Quiz 2, kursun **çizge bloğu sınavıdır**: erişilebilirlik, bileşenler, topolojik sıralama, SSSP (dört algoritma), APSP/Johnson. Veri yapıları ve sıralama Quiz 1'de kaldı; dinamik programlama Quiz 3'e kalır.

- **Bu = ikinci sınav.** OMSCS CS 6515 (Graduate Algorithms) ekseninde Quiz 2, çizge algoritmaları



Şekil 29.1: Ders 22'nin (Quiz 2 Review) kavram haritası: kök = Quiz 2 Review. Beş dal — (1) modelleme ve indirgeme: sözel problemi bir çizgeye çevir (düğüm/kenar/ağırlık/yön tanımla), sonra bilinen kara kutuya indirge, algoritmayı asla modifiye etme. (2) çizge algoritma haritası: erişilebilirlik, full BFS ve DFS ile bağlı bileşen, DAG araçları (topolojik sıra ve çevrim), Bellman-Ford ile negatif çevrim. (3) SSSP hiyerarşisi: BFS sonra DAG relaxation sonra Dijkstra sonra Bellman-Ford — artan genellik. (4) APSP ve Johnson: potansiyel ile yeniden ağırlıklandır, sonra her düğümden Dijkstra. (5) graf değiştirme stratejileri: düğüm çoğaltma ile durum izleme, süpernode ile çok kaynak ya da çok hedef, ön işleme ile filtreleme. Sonuç: Quiz 2 = icat değil modelleme sınavı.

+ indirgeme refleksidir — lisansüstü dersin giriş varsayımı.

- **Modelleme = gerçek dünya refleksi.** "Sözel problemi bir çizgeye çevir" becerisi, lisans sonrası en sık kullanılan algoritmik beceridir.
- **İleriye → graduate algorithms:** "algoritmayı modifiye etme, kara kutuya indirge" disiplini, lisansüstü derste ve gerçek sistemde temeldir.

Tek cümle: *Quiz 2, "yeni çizge algoritması yaz" demez; "problemi doğru bir çizgeye modelle, ne sakladığını ve aradığını net söyle, BFS/Dijkstra/Bellman-Ford/Johnson kara kutusuna indirge, sonra süreyi grafın boyutundan analiz et" der.*

29.2 1. Quiz 2 Neyi Ölçer — Modelleme ve İndirgeme

Sınav, çizge algoritması **icat etmeni** beklemez (onların doğruluğunu derslerde sayfalarca kanıtladık). İki refleks öne çıkar:

- **Modelleme:** sözel problemde çizge *verilmemiş* olabilir — onu **sen tanımlarsın** (düğüm? kenar? ağırlık? yönlü mü?). Önce temiz bir **soyut problem** ifade et: "bu soyut problemi çözebilseydim, sözel problemi kolayca çözerdim."

  "see if you can state cleanly an abstract problem." — Ku, 10:54

- **İndirgeme, modifiye etme değil:** verilen algoritmaları *değiştirmeye* çalışma; onları kara kutu olarak kullan. Karmaşıklık, **grafı "bariz-olmayan" yapmaktan** gelir (girdi grafı $\neq$ çözümde kullanacağın graf).

  "the way in which we introduce complexity into problems is to make the graph non-obvious." — Ku, 13:19

29.3 2. Çizge Algoritmaları Haritası

Az algoritma, çok problem. Ana kara kutular:

- **Erişilebilirlik (tek kaynak):** s'den nereye ulaşılır; bir bağlı bileşen, $\leq |E|$ düğüm.
- **Full BFS/DFS:** tüm grafı gez, **bağlı bileşenleri** say — $O(V + E)$.
- **DAG araçları:** full DFS ters bitiş sırası → **topolojik sıralama**; geri kenar → **çevrim tespiti**.
- **Bellman-Ford:** negatif ağırlıklı çevrim **tespit/bul**.

> *"there's really a small number of graph algorithms but they can solve a lot of different problems."* — Ku, 1:27

29.4 3. SSSP Hiyerarşisi: BFS → DAG → Dijkstra → Bellman-Ford

Tek-kaynak en kısa yol için, **artan genellik** (azalan kısıt) sırasıyla:

- **BFS** — ağırlıksız; $O(V + E)$.
- **DAG relaxation** — çevrimsiz, *herhangi* ağırlık; $O(V + E)$.
- **Dijkstra** — ağırlık ≥ 0; $O(V \log V + E)$.
- **Bellman-Ford** — *herhangi* (negatif çevrim dahil); $O(V \cdot E)$.

> *"in general you want to choose an algorithm that's higher on this list but sometimes the algorithms higher on this list don't apply."* — Ku, 5:41

Kritik sınav tuzağı: listede yukarıdakini seç **ama yalnız uygulanabilirse**. Bir DAG değilken DAG relaxation kullanırsan **yanlış** olur (sıfır puan). Sıkışınca, doğru-ama-yavaş Bellman-Ford yaz — yanlış-ama-hızlıdan iyidir.

> *"you'll get more points because it is a correct algorithm than if you apply a faster algorithm that doesn't apply."* — Ku, 6:13

Builder Notu — Modelleme = Gerçek-Dünya Refleksi

"Sözel problemi bir çizgeye çevir" becerisi sınıf dışında her yerde: bağımlılık çözümü (paket yöneticisi = topolojik sıra), navigasyon (en kısa yol = Dijkstra), sosyal/öneri grafları (BFS ile erişilebilirlik). Quiz 2'nin sana öğrettiği refleks tam olarak budur: önce **düğüm/kenar/ağırlık** tanımla, sonra hangi kara kutunun uygulandığını söyle. "En kısıtlı uygulanabilir algoritmayı seç" disiplini, pratikte performans bilincidir.

29.5 4. APSP ve Johnson

Tüm-çiftler en kısa yol: en basiti her düğümden SSSP ($V\times$ Bellman-Ford veya $V\times$ Dijkstra). **Johnson**, negatif ağırlıklı çizgede $V\times$ Bellman-Ford'un yavaşlığından kurtarır: süpernode'dan Bellman-Ford ile potansiyel $h = \delta(s, v)$ bul, kenarları $w' = w + h(u) - h(v)$ ile negatif-olmayana çevir (üçgen eşitsizliği garanti eder, en kısa yollar korunur), $V\times$ Dijkstra çalıştır → $O(V^2 \log V + V \cdot E)$.

29.6 5. Graf Değiştirme Stratejileri

Girdi grafı, çözümde kullanacağın graf olmayabilir. Üç klasik dönüşüm:

- **Düğüm çoğaltma (state):** gezerken bir "durum" izlemek gerekiyorsa, düğümleri çoğalt — her olası durum için ayrı düğüm (örneğin kalan kapasite, kaç adımda bir mola).

  > *"you can expand the number of vertices in your graph to keep track of what state I'm in."* — Ku, 13:47

- **Süpernode / yardımcı düğüm:** birçok kaynaktan/hedefe aynı anda aramak için, hepsine kenarlı bir yardımcı düğüm ekle, tek SSSP çalıştır.

  > *"adding an auxiliary node… run a single source shortest path algorithm from that super node."* — Ku, 14:30

- **Ön işleme (preprocessing):** bazı kenarları yasakla, yön ver veya grafı filtrele (yasak düğümleri at, çevrimi kır, gereksiz kısmı buda).

> Builder Notu — Süpernode ve Düğüm Çoğaltma Sistem Örnekleri
>
> İki dönüşüm gerçek sistemlerde sürekli görünür:
>
> - **Süpernode** = çok-kaynak/çok-hedef indirgemesi. Bir veri merkezinde "en yakın herhangi bir CDN düğümü" sorusu, tüm CDN düğümlerine sıfır-ağırlıklı bağlı tek yardımcı kaynaktan tek SSSP'ye iner. Bu derste Problem 3 (donut filtreleme) tam olarak bu kalıbı kullanır.
> - **Düğüm çoğaltma** = durum-augmentasyonu. Zaman-genişletilmiş çizge (her düğüm × zaman dilimi), mod/kapasite katmanları (her düğüm × kalan yakıt). PS6'da "kaç boş küre taşıyorum" durumu bu şekilde katmanlandı; bu derste Problem 4 (≥V kenarlı yol) katmanlı çizge ile çözülür.

29.7 6. Sınav Taktiği ve Puan Kaybı

- **Önce grafı tanımla.** Sözel problemde graf yoksa, "düğüm = …, kenar = …, ağırlık = …" diye açıkça kur. (En sık puan kaybı: graf tanımlamamak.)
- **Grafı tam belirt:** kaç düğüm/kenar, çevrimsiz mi, ağırlıklar ne. Graderın işini kolaylaştır.
- **Çözdüğün problemi söyle:** "bu grafta en kısa yol / bağlı bileşen / topolojik sıra arıyorum." Yanlış algoritma seçsen bile, doğru *problemi* belirtmek puan getirir.
- **Doğruluk cümlesi:** "yeni graftaki X, orijinal problemdeki Y'ye karşılık gelir."
- **Süreyi analiz et:** grafın boyutu (V, E) + algoritmanın o boyuttaki süresi. (Unutmak büyük kayıp.)

> *"almost any question in this class can… get 80 to 90 percent of the points by writing maybe three lines."* — Ku, 1:03:03

29.8 Bu Quiz Review'in Özeti

1. **Quiz 2 = çizge bloğu** (Ders 13-21); az algoritma, çok problem; ana iş **modelleme + indirgeme**.
2. **Algoritma haritası**: erişilebilirlik, full BFS/DFS (bileşen), DAG (topolojik/çevrim), Bellman-Ford (negatif çevrim).
3. **SSSP hiyerarşisi**: BFS $\to$ DAG relaxation $\to$ Dijkstra $\to$ Bellman-Ford; yukarıyı seç ama uygulanabilirse.
4. **APSP/Johnson**: potansiyel ile yeniden ağırlıklandır $\to$ $V\times$ Dijkstra $\to$ $O(V^2 \log V + V \cdot E)$.
5. **Graf değiştirme**: düğüm çoğaltma (state), süpernode (çok kaynak/hedef), ön işleme (filtrele/yönlendir).
6. **Taktik**: grafı tanımla + tam belirt + çözdüğün problemi söyle + doğruluk cümlesi + süre analizi.

! Tek Bir Cümle

Quiz 2, icat değil **modelleme** sınavıdır: problemi doğru bir çizgeye çevir, ne sakladığını ve aradığını açıkça yaz, BFS/Dijkstra/Bellman-Ford/Johnson kara kutusuna indirge — algoritmayı asla modifiye etme — ve süreyi grafın boyutundan analiz et.

29.9 Quiz-tarzı Problemler

Aşağıda dört quiz-tarzı problem var; her birinin çözümünü açmadan önce kendin dene. Her problem aynı reçeteyi izler: **grafı tanımla → kara kutuya indirge → süreyi analiz et**. Tüm sayılar QR2 motoruyla doğrulanmıştır.

Şekil 29.2 Problem 1'in modelini gösterir: beyaz pikseller düğüm, 4-komşu beyaz çiftler kenar; blob sayısı = bağlı bileşen sayısı.

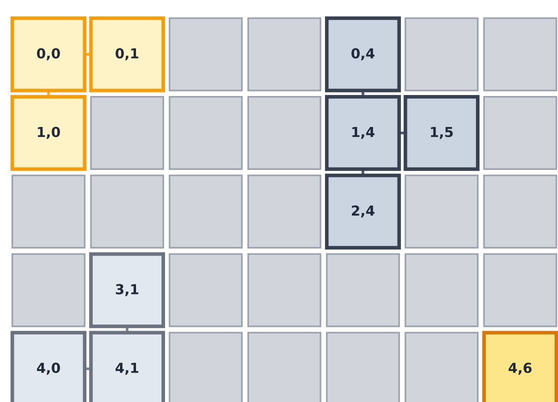
i Problem 1 (Blob sayma — bağlı bileşenler): n×m beyaz/siyah piksel grid; kenar paylaşan iki beyaz piksel aynı blobda. Blob sayısını O(n·m)'de bul.

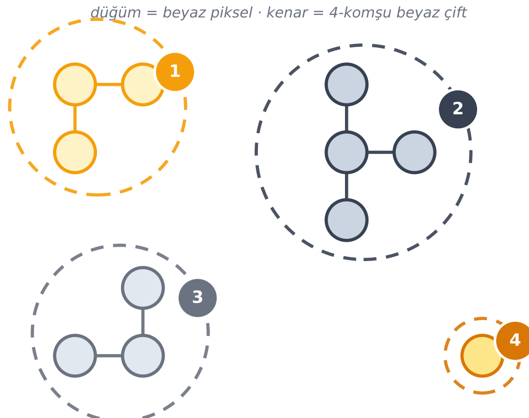
Çözüm.
"Blob" tanımı transitiftir (a–b, b–c bitişikse a, c aynı blob) $\to$ bu bir **bağlı bileşen** problemidir. Çizge kur: **düğüm = her beyaz piksel** (x,y koordinatıyla kimliklendir); **kenar = bitişik (kenar paylaşan) iki beyaz piksel**. $V \leq n \cdot m$, $E \leq 4 \cdot n \cdot m$ (her piksel ≤ 4 komşu).
Doğruluk cümlesi: görüntüdeki **blob sayısı = bu çizgedeki bağlı bileşen sayısıdır**.

> *"the number of blobs in the image corresponds to the number of connected components in this graph."* — Ku, 31:10

Full BFS/DFS ile bağlı bileşenleri say. (Girdi karesel görünse de — $n \cdot m$ piksel — girdi *boyutu* $n \cdot m$ olduğundan bu **doğrusaldır**.)
Karmaşıklık. $V = O(n \cdot m)$, $E = O(n \cdot m)$ $\to$ full BFS/DFS $O(n \cdot m)$.

Şekil 29.3 Problem 2'nin imza fikrini gösterir: $E = V$ olduğunda çizge ağaç + bir kenardır (tam bir çevrim); s 'nin iki çevrim kenarını teker teker silerek iki aday yol elde edilir.





Şekil 29.2: Blob sayma = bağlı bileşen (QR2 Problem 1, Ku 31:10): n×m beyaz/siyah piksel grid → çizge; düğüm = beyaz piksel (r,c), kenar = 4-komşu beyaz çift. Doğruluk cümlesi: görüntüdeki blob sayısı = bu çizgedeki bağlı bileşen sayısı. SOL panel: 5×7 grid, 11 beyaz piksel dört bileşene göre renkli (amber + üç slate tonu); siyah pikseller açık gri. SAĞ panel: çizge modeli — bileşen başına kesikli hale + numara rozeti; toplam 4 bileşen = 4 blob. Motordan: V = 11 beyaz piksel, nm = 35, bileşenler {(0,0),(0,1),(1,0)}, {(0,4),(1,4),(1,5),(2,4)}, {(3,1),(4,0),(4,1)}, {(4,6)}. E = O(nm) (her piksel en çok 4 komşu) → full BFS/DFS O(nm); girdi nm piksel olduğundan DOĞRUSAL.

ℹ Problem 2 (Ağaç + bir kenar): bağlı yönsüz çizge, pozitif ağırlık, E = V. s→t en kısa yolu O(V)'de bul.

Çözüm.
$E = V$ gözlemi kilit: bir ağaç $V - 1$ kenarlıdır, demek ki bu çizge **ağaç + bir fazladan kenar** = tam olarak **bir çevrim** içerir.

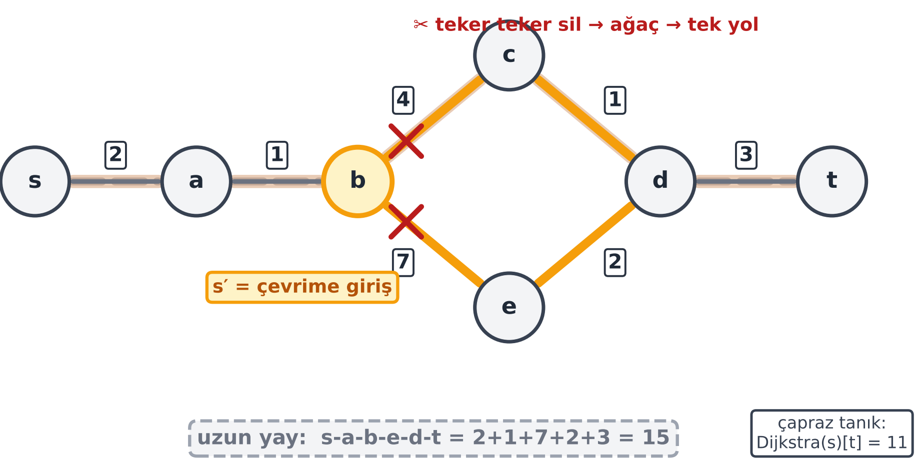
> *"this graph is a tree plus an extra edge."* — Ku, 37:21

Pozitif ağırlık → en kısa yollar basit. Çevrim yoksa (ağaçta) iki düğüm arasında **tek** basit yol vardır → ağırlıksız erişilebilirlik (BFS/DFS) ağacı o yolu verir. Çevrim varsa s→t için **iki** olası basit yol (çevrimin iki yarısı). Algoritma: BFS/DFS ile bir yayılma ağacı kur; ağaçta olmayan kenar (u, v) çevrimi belirler; s'ten u ve v'ye yolların **son ortak düğümü** s (çevrimde s'e en yakın). s 'nin çevrim kenarlarını teker teker kaldırıp her seferinde s→t erişilebilirliğini çalıştır, en kısasını seç. Sabit sayıda ($\leq$ derece) BFS/DFS.
Karmaşıklık. Sabit sayıda $O(V)$ erişilebilirlik araması + ön-ek bulma $O(V) \to O(V)$.

Problem 3'ün figürü yoktur — süpernode kalıbı, PS6'daki sensör problemi görselinde zaten ayrıntılı görselleştirildi (donut shop = sensör, aynı süpernode + tek Dijkstra şeması). Aşağıda doğrudan çözüme geçiyoruz.

Ağaç + 1 kenar (E = V): tek çevrim → s′'nin iki çevrim kenarını teker teker sil → her seferinde ağaç

KAZANAN: s-a-b-c-d-t = 2+1+4+1+3 = 11



E = V → TAM bir çevrim (Ku 37:21). s′'nin 2 çevrim kenarı için sabit sayıda O(V) tarama → toplam O(V).

Şekil 29.3: Ağaç + bir kenar (QR2 Problem 2, İMZA figür, Ku 37:21): bağlı yönsüz çizge, pozitif ağırlık, E = V → ağaç (V−1 kenar) + bir fazladan kenar = TAM BİR çevrim. Çevrim b-c-d-e (motordan); s kuyruğu s-a-b, t kuyruğu d-t. s = b (s'in çevrime bağlandığı düğüm). İmza fikir: s ’nin iki çevrim kenarını (b-c ve b-e) teker teker sil → her silme çevrimi kırar → ağaç → tek basit yol; iki adayın min'i. KAZANAN kısa yay s-a-b-c-d-t = 2+1+4+1+3 = 11 (amber); kaybeden uzun yay s-a-b-e-d-t = 2+1+7+2+3 = 15 (kesikli slate). Dijkstra çapraz tanık = 11 (sağ alt). Sabit sayıda O(V) tarama → O(V).

Problem 3 (Donut — süpernode filtreleme): n konum, derece ≤ 5, d donut shop, mesafe k. p→h, donut'a k'dan yakın olmayan en kısa yol, O(n log n).

Çözüm.
Çizge: düğüm = konum (n tane), kenar = yol (ağırlık = pozitif sürüş mesafesi); derece $\leq 5 \rightarrow E = O(n)$. Bir donut'a $\leq k$ mesafedeki *her* düğüm yasaktır. d donut'tan teker teker Dijkstra $O(d \cdot n \log n)$ — d sınırsız, yavaş. **Süpernode** ile paralelleştir.

"filter forbidden vertices by using supernode plus one run of Dijkstra." — Ku, 1:01:30

[illegible] mesafesi $\leq k$ olan düğümleri **çıkar** (filtrele). (3) Kalan grafta p→h Dijkstra → istenen en kısa yol. (Genelleme: her donut farklı yarıçap → giriş kenar ağırlığını maks_yarıçap − yarıçap yap.)

Bu, PS6'daki sensör problemiyle **aynı kara kutudur** — orada "her an sensörlerden $\geq k$ uzakta kal", burada "donut'lardan $> k$ uzakta kal". Motorda da `nearest_sensor_dist` aynen yeniden kullanılır: bu dersteki deterministik örnek (donut q, b'ye 2 uzakta, $k = 3$ ile b yasak) en yakın mesafeleri a=4, c=4, p=6, h=6, d=9, e=9 verir; üst koridor 8 kapanır → alt dolambaç **9**. $k = 1$ ile b serbest (2 > 1) → üst koridor açılır → **8**.

Karmaşıklık. İki Dijkstra + filtreleme → $O(n \log n)$.

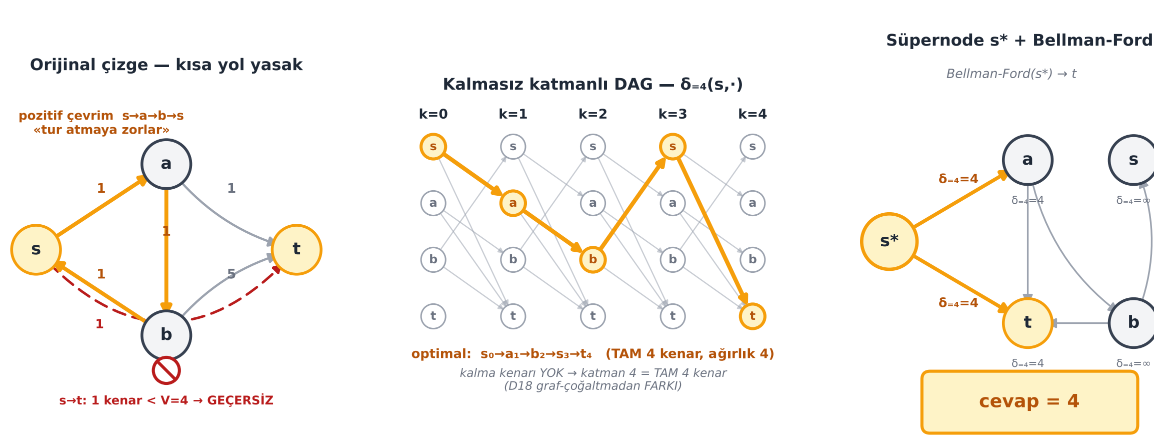
Problem 4 (Uzun-kısa yol): yönlü, keyfi ağırlık; en az V kenarlı en küçük ağırlıklı s→t yolunu $O(V^3)$'te bul.

Çözüm.
"En az V kenar" → yol **basit olamaz** ($V + 1$ düğüm gerekir). Önce **Bellman-Ford**: negatif çevrim s→t yolunda ise cevap $-\infty$ (sonsuz kenarlı yol mevcut). Yoksa: "tam V kenarlı" mesafeyi düşün. **Graf çoğaltma**: $V + 1$ katman, kenarları yalnız bir alt katmana yönlendir (kalma-kenarı *yok*) → katman 0'dan katman V'ye yol = orijinalde **tam V kenarlı** yol; çizge bir **DAG** → DAG relaxation $O(V^3)$ (G boyutu $V \cdot (V + E) = O(V^3)$).

"En az V" için: tam-V-kenarlı mesafeleri (DAG relaxation çıktısı) bir **süpernode** ile alt katmana bağla, sonra orijinal graf üzerinde **Bellman-Ford** çalıştır (kalan kısım basit yol — negatif çevrim yok). Üst kısım (DAG) ucuz, alt kısım (çevrimli) küçük → karmaşıklık ayrıştırılır. Deterministik örnek bunu doğrular: $\delta_4 = \{s : \infty, a : 4, b : \infty, t : 4\}$, optimal yol s→a→b→s→t (tam 4 kenar, ağırlık 4); doğrudan s→t ağırlık 1 ama tek kenar olduğu için **geçersiz**. Negatif çevrim varyantında (a ile b arası [illegible] −2) cevap $-\infty$; kaba kuvvet k_{max} 16'dan 32'ye çıkınca değer −12'den ≥ −28'e düşer — ıraksamanın sayısal izi.

Karmaşıklık. Bellman-Ford $O(V \cdot E)$ + DAG relaxation $O(V^3)$ + son Bellman-Ford $O(V \cdot E)$ = $O(V^3)$.

Şekil 29.4 Problem 4'ün üç adımını gösterir: orijinal çizgede kısa yolun [illegible] geçersiz olduğu, kalmasız katmanlı DAG ve süpernode + Bellman-Ford sonlu [illegible]



Şekil 29.4: En az V kenarlı en küçük ağırlıklı yol (QR2 Problem 4, Ku): yönlü, keyfi ağırlık; en az V kenarlı en küçük ağırlıklı s⟶t yolu. SOL panel: orijinal çizge (V=4); doğrudan s⟶t ağırlık 1 ama yalnız 1 kenar < V ⟶ GEÇERSİZ (kesikli kırmızı + yasak işareti); pozitif çevrim s⟶a⟶b⟶s yolu tur atmaya zorlar. ORTA panel: kalmasız katmanlı DAG (k=0..4, kalma kenarı YOK) ⟶ katman 4 = TAM 4 kenar; optimal $s_0 \longrightarrow a_1 \longrightarrow b_2 \longrightarrow s_3 \longrightarrow t_4$ (amber rota, ağırlık 4). SAĞ panel: süpernode s* ⟶ δ_4 sonlu düğümler (a=4, t=4; s ve b sonsuz) + orijinal çizgede Bellman-Ford soneki ⟶ cevap = 4. Motordan: δ_4 = {s sonsuz, a=4, b sonsuz, t=4}, optimal yol s⟶a⟶b⟶s⟶t (tam 4 kenar, ağırlık 4); doğrudan s⟶t (ağırlık 1) tek kenar olduğu için geçersiz. Negatif çevrim varyantında (a ile b arası çift −2) cevap $-\infty$; brute kmax 16⟶32 hâlâ düşer (−12⟶−28), iraksamanın sayısal izi.

29.10 Quiz Hazırlığı Egzersizleri

Egzersiz 1. Bir sözel problemi (örneğin labirent, ağ, oyun) çizgeye modelle: düğüm/kenar/ağırlık/yön tanımla, çözeceğin soyut problemi (en kısa yol / bileşen / topolojik) ifade et.

Egzersiz 2. SSSP hiyerarşisi tablosunu (BFS/DAG/Dijkstra/Bellman-Ford) ezberden çıkar; her biri için "hangi kısıt + hangi süre" yaz.

Egzersiz 3. Üç graf-değiştirme stratejisini (düğüm çoğaltma, süpernode, ön işleme) birer örnek problemle eşle.

Egzersiz 4. Bir "durum izleme" problemini düğüm çoğaltmayla çöz (örneğin kapasite/mod katmanları); V ve E'nin nasıl büyüdüğünü hesapla.

Egzersiz 5. Johnson'ın 5 adımını ezberden anlat; her adımın süresini ve neden $V\times$ Dijkstra'nın baskın olduğunu açıkla.

29.11 Quiz 3 Öncesi Kapsam Genişlemesi

Quiz 2 buraya kadar; sıradaki blok (Ders 23+, **Quiz 3** kapsamı) **dinamik programlamaya (DP)** geçer:

- **Ders 23-27 (L15-L18; araya Ders 25 = PS8 girer):** DP temelleri — alt problem tanımı, ilişki (recurrence), topolojik sıra, taban durum, çözüm kurma.
- DP, "kendi algoritmanı tasarla" ünitesidir; en kısa yolların **optimal alt yapı** ve **örtüşen alt problem** sezgileri DP'nin habercisidir.

Bağ: çizge bloğu "kara kutuya indirge" iken, DP "alt problemleri birleştir" der — ama ikisi de aynı optimal-alt-yapı omurgasını paylaşır.

29.12 Ders 13-21 Toplu Cheat Sheet (L9-L14 + PS5-6)

Konu	Özü	Kaynak (L/PS)
Çizge G = (V, E)	Komşuluk listesi; $O(1)$ kenar sorgu	L9
BFS	Seviye kümeleri; ağırlıksız en kısa yol; $O(V+E)$	L9
DFS / full DFS	Erişilebilirlik $O(E)$; bileşen / topolojik / çevrim $O(V+E)$	L10
DAG relaxation	Topolojik sırada gevşet; herhangi ağırlık; $O(V+E)$	L11

Konu	Özü	Kaynak (L/PS)
Bellman-Ford	Genel SSSP; negatif çevrim $-\infty$; $O(V \cdot E)$	L12
Dijkstra	Ağırlık ≥ 0; öncelik kuyruğu; $O(V \log V + E)$	L13
Johnson (APSP)	Potansiyel ile yeniden ağırlıklandır $\rightarrow V \times$ Dijkstra; $O(V^2 \log V + V \cdot E)$	L14
Graf değiştirme	Düğüm çoğaltma / süpernode / ön işleme	PS5-6

Sonraki Ders

Ders 23 (L15): Dinamik Programlama 1 — SRTBOT
Erik Demaine ile **dinamik programlama (DP)** ünitesine geçiyoruz: SRTBOT çerçevesi (Subproblem, Relation, Topological order, Base case, Original problem, Time). Çizge bloğunun "kara kutuya indirge" disiplini biter; DP "alt problemleri tanımla ve birleştir" disiplinine geçer — ama optimal-alt-yapı omurgası aynı kalır.

29.13 Builder ve OMSCS Bağlantıları

6 köprü

Bu tekrar oturumu, "problemi doğru bir çizgeye modelle, kara kutuya indirge, sakladığını ve aradığını yaz" disiplinini kurar — köprülerin özeti:

1. **Quiz 2 = çizge midterm** $\rightarrow$ OMSCS CS 6515: çizge algoritmaları + indirgeme, graduate dersin giriş varsayımıdır.
2. **Modelleme** $\rightarrow$ **gerçek dünya** $\rightarrow$ **çizge:** sosyal graf, bağımlılık grafı, ağ topolojisi, durum-uzayı.
3. **Süpernode** $\rightarrow$ çok-kaynak/çok-hedef indirgemesi; veritabanı, ağ akışı, kümeleme.
4. **Düğüm çoğaltma** $\rightarrow$ durum-augmentasyonu: zaman-genişletilmiş çizge, mod/kapasite katmanları.
5. **SSSP hiyerarşisi** $\rightarrow$ "en kısıtlı uygulanabilir algoritmayı seç": pratik performans bilinci.
6. **Doğru ama yavaş > yanlış ama hızlı** $\rightarrow$ mühendislikte önce doğruluk, sonra optimizasyon.

! Tek bir şey alıp gideceksen

Quiz 2 senden çizge algoritması icat etmeni değil, **problemi doğru bir çizgeye modellemeni** ister. Düğüm/kenar/ağırlığı açıkça tanımla, çözeceğin soyut problemi (en kısa yol, bileşen, topolojik sıra) söyle, BFS/Dijkstra/Bellman-Ford/Johnson kara kutusuna **indirge** — algoritmayı asla modifiye etme — ve süreyi grafın boyutundan analiz et. Karmaşıklık, algoritmadan değil, "doğru grafı görmekten" gelir.

30 Dinamik Programlama 1: SRTBOT

Özyineleme + memoization: alt problemleri SRTBOT çerçevesiyle tanımla, bir recurrence ile ilişkilendir, topolojik sırada bir memo tablosunda sakla — Fibonacci üstelden polinoma iner, bowling ise suffix-DP'de yerel kaba kuvvetle O(n)'e; kursun yepyeni bölümünün, kendi algoritmanı tasarlama bölümünün açılış dersi

i Oturum bilgisi

- **Demaine'in videosu:** YouTube — Lecture 15: Dynamic Programming, Part 1 (≈57 dk)
- **OCW sayfası:** MIT 6.006 Lecture 15: Dynamic Programming, Part 1
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 23 (L15)
- **Hoca:** Erik Demaine (dinamik programlama; **DP ünitesinin AÇILIŞ dersi** — 4 dersin 1.si)
- **Okuma süresi:** ≈27 dk

  Bu, kursun **yepyeni bölümünün** açılışıdır: hazır algoritma uygulamaktan **kendi algoritmanı tasarlamaya** geçiyoruz. Demaine ile başlayan bu blok, en güçlü algoritmik tasarım paradigmasını öğretir: **dinamik programlama (DP)**. Özü iki kelimedir — özyineleme + memoization — ve tasarımı bir akronime bağlanır: **SRTBOT**.

30.1 Bu Derste Ne Var?

Kursun **yepyeni bölümü** başlıyor (Erik Demaine). Şimdiye dek hazır algoritmaları (sıralama, çizge, veri yapısı) *uyguluyorduk*; bundan sonra **kendi algoritmamızı sıfırdan tasarlayacağız** — özellikle **dinamik programlama (DP)**, en güçlü algoritmik tasarım paradigması.

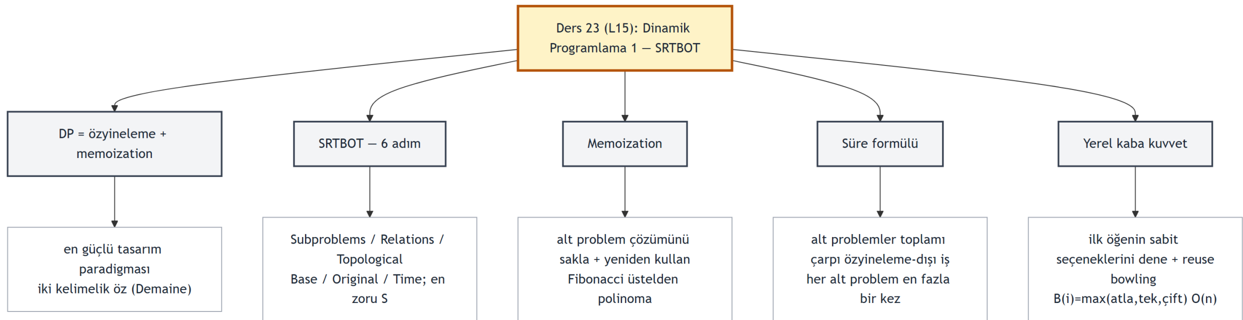
> *"dynamic programming… It's probably the most powerful algorithmic design paradigm."* — Demaine, 1:00

DP'nin özü iki kelimedir: **özyineleme + memoization**. Tasarım için bir akronim: **SRTBOT**.

> *"when I say dynamic programming, I mean recursion with memoization."* — Demaine, 27:13

Üç ana fikir:

1. **SRTBOT** — özyinelemeli algoritma tasarımının 6 adımı: **S**ubproblems, **R**elations, **T**opological order, **B**ase case, **O**riginal problem, **T**ime.
2. **Memoization** — alt problem çözümlerini sakla/yeniden kullan; Fibonacci'yi üstelden **polinoma** indirir.
3. **Yerel kaba kuvvet (local brute force)** — "ilk öğeye ne yapabilirim?" sorusunun polinom seçeneği varsa, polinom bir DP elde edilir.



Şekil 30.1: Ders 23'ün (L15) kavram haritası: kök = Dinamik Programlama 1 ve SRTBOT (Demaine) — DP ünitesinin açılış dersi; algoritma uygulamaktan algoritma TASARLAMAYA geçiş. Beş dal — (1) DP = özyineleme + memoization: en güçlü tasarım paradigması, iki kelimelik öz. (2) SRTBOT 6 adım: Subproblems, Relations, Topological order, Base case, Original problem, Time; en zoru S (doğru alt problemi bulmak). (3) Memoization: alt problem çözümünü sakla, yeniden kullan; Fibonacci üstelden (phi üzeri n) polinoma (n alt problem). (4) Süre formülü: alt problemler toplamı çarpı özyineleme-dışı iş; her alt problem en fazla bir kez. (5) Yerel kaba kuvvet: ilk öğenin sabit sayıda seçeneğini dene plus reuse; bowling suffix-DP B(i) = max(atla, tek, çift) O(n). Sonuç: DAG en kısa yol da bir DP'dir, memo gizli bir DFS'tir; dizi girdide önek/sonek/alt-dizi seç, alt-dizilim ASLA (2 üzeri n).

Builder Notu — Memoization = caching

DP'nin tek değiştirdiği şey, saf özyinelemeye "hesaplananı sakla, gerektiğinde yeniden kullan" cümlesini eklemektir. Bu desen gerçek sistemlerde **önbelleklemenin (caching)** ta kendisidir: pahalı bir hesap (HTTP isteği, veritabanı sorgusu, embedding çıkarımı) bir kez yapılır, sonucu bir tabloya yazılır, aynı girdi tekrar gelince tablodan döner. Fibonacci'nin üstelden polinoma inişi, bir memcache/Redis katmanının neden bu kadar büyük bir hız farkı yarattığının minyatür ispatıdır.

- **İleriye → DP her yerde:** dizi hizalama (DNA, diff), Viterbi/DTW (ses/ML), en kısa yol, knapsack, metin akıllı-kırpma — hepsi DP.
- **İleriye → memoization = caching:** "hesaplananı sakla/yeniden kullan" deseni, gerçek sistemde önbellekleme.
- **Geriye → DAG shortest paths (Ders 16):** DAG relaxation aslında bir DP; SRTBOT içinde gizli bir DFS çalışır.
- **İleriye → DP 2-4 (Ders 24-27):** string, ağaç, pseudopolinom alt problemler.

Tek cümle: *DP = özyineleme + memoization; SRTBOT ile alt problemleri tanımla, bir recurrence ile ilişkilendir, topolojik sırada çöz, çözümleri bir memo tablosunda sakla — böylece "yerel kaba kuvvet" üstel aramayı polinom zamana indirir.*

30.2 1. Yeni Bölüm: Algoritmik Tasarım ve DP

Şimdiye dek "ver(il)en algoritmaya indirge" yaptık. Artık **sıfırdan polinom-zaman algoritma** tasarlıyoruz. DP, özyinelemeli bir tasarım türüdür ve tümevarımla kanıt tekniğimizle çok iyi uyuşur.

> *"Today we start a totally new section of the class."* — Demaine, 0:18

DP'nin temeli: alt problemleri tanımla, aralarındaki ilişkiyi (recurrence) yaz, çevrimsiz (DAG) bir alt-problem grafı kur, çözümleri sakla. Tüm dersi tek bir disiplinde toparlayan akronim **SRTBOT**'tur.

30.3 2. SRTBOT Çerçevesi

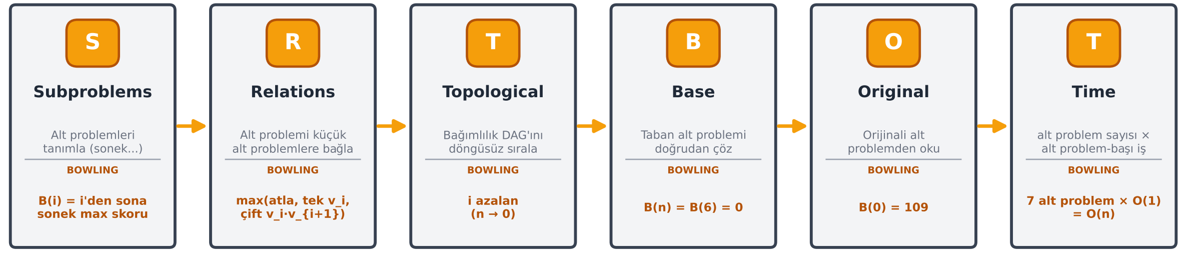
Özyinelemeli algoritma tasarımının altı adımı (akronim **SRTBOT**):

> *"SRTBOT… sub-problems, relations, topological order, base case, original problem, and time."* — Demaine, 2:33

- **S — Subproblems (alt problemler):** problemi polinom sayıda alt probleme böl. *En zor adım* — doğru alt problemi bulmak.
- **R — Relations (ilişki):** bir alt problemi daha küçüklerinden çöz (recurrence).
- **T — Topological order:** alt-problem grafı **DAG** olmalı (çevrim = sonsuz döngü); topolojik sıra.
- **B — Base case (taban durum):** özyinelemenin durağı.
- **O — Original problem:** çözmek istediğin (genelde alt problemlerden biri).
- **T — Time (süre):** $\sum$ (alt problemler) $\times$ özyineleme-dışı iş.

DP = SRTBOT + **memoization**.

Şekil 30.2 altı adımı tek bir yatay zincirde toplar; her kutuda adımın adı, Türkçe tek-satır açıklaması ve altında bu dersin **bowling** örneğindeki karşılığı yer alır (S: sonek $B(i)$, R: max(atla, tek, çift), …, O: $B(0) = 109$ — bu sayı motordan canlı hesaplanır, T: 7 alt problem $\times$ $O(1) = O(n)$).



Şekil 30.2: SRTBOT — özyinelemeli tasarımın 6 adımı yatay zincir (Demaine L15 §2): S→R→T→B→O→T. Her kutuda adım adı (İngilizce) + Türkçe tek-satır açıklama + altında BOWLING karşılığı (amber). S: Subproblems — B(i) = i'den sona sonek max skoru. R: Relations — max(atla, tek v_i, çift v_i·v_{i+1}). T: Topological — i azalan (n → 0). B: Base — B(n)=B(6)=0. O: Original — B(0)=109 (motordan canlı). T: Time — 7 alt problem × O(1) = O(n). Amber oklar zinciri kutudan kutuya bağlar. Alt not: Demaine 2:33 — her DP çözümü bu altı adımın bir örneğidir. DP = SRTBOT + memoization. Veri MOTORDAN: bowling_bottom_up([1,9,9,2,−5,−5])[0] == 109 (assert); B(n)=B(6)=0 taban (assert); n=6 → T kutusu '7 alt problem × O(1) = O(n)'.

30.4 3. Örnek: Merge Sort SRTBOT ile

Bildiğimiz merge sort'u SRTBOT'a oturtalım. **S:** $S(i,j) = A[i..j-1]$'i sırala. **O:** $S(0,n)$. **R:** $S(i,j) =$ merge$(S(i,m), S(m,j))$ (m = orta). **T:** $j-i$ artan sırada. **B:** boş dizi. **Time:** $O(n \log n)$.

(Merge sort memoization'dan *yararlanmaz* — alt diziler ayrık, tekrar yok. Ama çerçeve uygulanabilir; SRTBOT yalnızca DP'ye değil, her özyinelemeli tasarıma uyar.)

30.5 4. Fibonacci: Memoization'sız Üstel

Fibonacci: $f_n = f_{n-1} + f_{n-2}$, $f_1 = f_2 = 1$. **S:** $f(i) = i$. Fibonacci. **R:** $f(i) = f(i-1) + f(i-2)$. Doğal özyineleme — ama özyineleme ağacı çizilince $f(n-3), f(n-2)$ gibi alt problemler **defalarca** hesaplanır.

$T(n) = T(n-1) + T(n-2) + 1 \to$ çözüm yaklaşık φ^n (altın oran, üstel). Üstel = kötü; yalnız çok küçük n çözülür.

Şekil 30.3 bu dersin imza karşılaştırmasını verir: solda $f(6)$'nın tam özyineleme ağacı (memoization YOK) — motordan sayılan tekrarlar amber tonlarıyla işaretli ($f(3)$ tam 3 kez, $f(2)$ 5 kez, $f(1)$ 3 kez; ağaç 15 düğüm = naif 15 çağrı); sağda aynı alt problemler memoization ile **tek kopya** zincir-DAG ($f(1)$ … $f(6)$) ve büyük sayılar paneli ($n = 20$: naif 13.529 çağrı, memo TAM 20 alt problem; oran calls$(20)/$calls$(15) = 11{,}10 \approx \varphi^5 = 11{,}09$ — büyüme $\approx \varphi^n$ üstel).

30.6 5. Memoization: Üstelden Polinoma

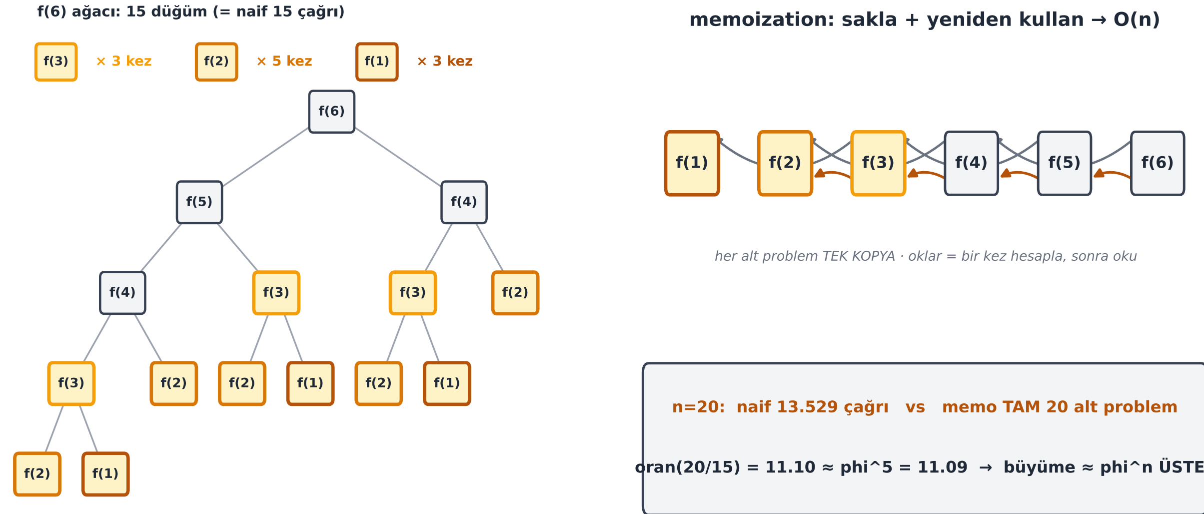
Küçük bir kurnazlık her şeyi değiştirir: **memoization** — alt problem çözümlerini bir "memo" tablosuna yaz, gerektiğinde yeniden kullan.

> *"remember and reuse solutions to sub-problems."* — Demaine, 14:14

Generic DP yapısı (her DP böyledir):

Memo tablosu genelde bir **doğrudan-erişim dizisi** veya hash tablosu. Fibonacci için: her alt problem en fazla bir kez çözülür $\to$ **n alt problem** $\times O(1) = O(n)$. (Bir DFS'in "ziyaret edildi mi" kontrolünü oynar memo tablosu.)

Şekil 30.4 bu generic iskeleti bir akış diyagramı olarak çizer: çağrı $\to$ "alt problem memoda mı?" (EVET $\to$ tabloyu döndür, tekrar hesaplama YOK) $\to$ "taban durum mu?" (EVET $\to$ taban değer; HAYIR $\to$ recurrence ile özyinele) $\to$ sonucu **sakla** $\to$ döndür. Sağ alt köşede, memonun bir DFS'in "ziyaret edildi mi" kontrolünü oynadığı not edilir. Motor tanığı: bowling_memo_counted$([1, 9, 9, 2, -5, -5]) = (109, 7)$ — $n + 1 = 7$ alt problem, her biri tam bir kez.



Şekil 30.3: Fibonacci: özyineleme ağacından memoization'a — üstelden polinoma (Demaine L15 §4-5 İMZA). SOL: f(6) tam özyineleme ağacı (memoization YOK); tekrarlanan alt problemler aynı amber tonuyla — f(3) 3 kez, f(2) 5 kez, f(1) 3 kez (motordan sayıldı); ağaç 15 düğüm = naif 15 çağrı. SAĞ: aynı düğümler memo ile TEK KOPYA zincir-DAG (f(1)..f(6)); her f(i)'den f(i−1) ve f(i−2)'ye geri-ok (bir kez hesapla, sonra oku); büyük sayılar paneli — n=20: naif 13.529 çağrı vs memo TAM 20 alt problem; oran(20/15) = 11.10 ≈ phi^5 = 11.09 → büyüme yaklaşık phi^n ÜSTEL. Veri MOTORDAN (assert): fib_naive_counted(20) = (6765, 13529); fib_naive_counted(15)[1] = 1219; fib_naive_counted(6) = (8, 15); fib_memo_counted(20) = (6765, 20); ratio 13529/1219 = 11.10 ≈ phi^5 = 11.09; f(6) ağaç düğüm 15, f(3) tekrar 3.

Şekil 30.4: Generic DP iskeleti — f(alt problem) akış diyagramı (Demaine L15 §5 pseudocode birebir): memo kontrolü → taban → recurrence → sakla. SOL DİKEY ANA AKIŞ: çağrı → karar ‘ap ∈ memo?’ → (HAYIR) karar ‘taban durum?’ → (EVET) result = taban değer / (HAYIR) result = relation(f, ap), özyinele → her iki dal SAKLA kutusunda birleşir (memo[ap] = result, amber). SAĞ: memo HIT kısa-devresi — ‘ap ∈ memo?’ EVET → memo[ap] döndür, TEKRAR HESAPLAMA YOK (amber kalın) → return result. SAĞ ALT KÖŞE: memo = bir DFS’in ziyaret-edildi-mi kontrolü; DP içinde GİZLİ DFS çalışır (Demaine §7). ALT NOT (motor tanıklı): her alt problem EN FAZLA BİR KEZ çözülür → T = Σ (alt problem) × (özyineleme-dışı iş); tanık bowling([1,9,9,2,−5,−5]) → skor 109, 7 alt problem (n+1). Veri MOTORDAN (assert): bowling_memo_counted([1,9,9,2,−5,−5]) == (109, 7); fib_memo_counted(10) == (55, 10); ikisi de AYNI iskelet.

30.7 6. Çalışma Süresi Formülü

Memoization'lı bir DP'nin süresi:

T = Σ (tüm alt problemler) × (recurrence'ın özyineleme-dışı işi)

> *"the time it takes is at most the sum over all sub problems of the relation time."* — Demaine, 23:07

Çünkü her alt problem **en fazla bir kez** çözülür; özyinelemeli çağrılar zaten toplama dahildir. (Fibonacci: $n \times O(1) = O(n)$.)

Word-RAM inceliği. Fibonacci sayıları n-bit büyüklüğüne ulaşır; bir w-bit makine sözcüğünde toplama, $\lceil n/w \rceil$ sözcük işine mal olur. Dolayısıyla daha hassas süre $O(n + n^2/w)$'dir — ama bu da w sabitken **polinom** kalır. Asimptotik tablo değişmez: memoization, Fibonacci'yi üstelden polinoma indirir.

30.8 7. DAG En Kısa Yol DP Olarak

DAG tek-kaynak en kısa yol (Ders 16) aslında bir DP'dir. **S:** $\delta(s, v)$ tüm v. **O:** hepsi. **R:**

$$\delta(s, v) = \text{min}(\{\delta(s, u) + w(u, v) : (u, v) \text{ gelen kenar}\} \cup \{\infty\})$$

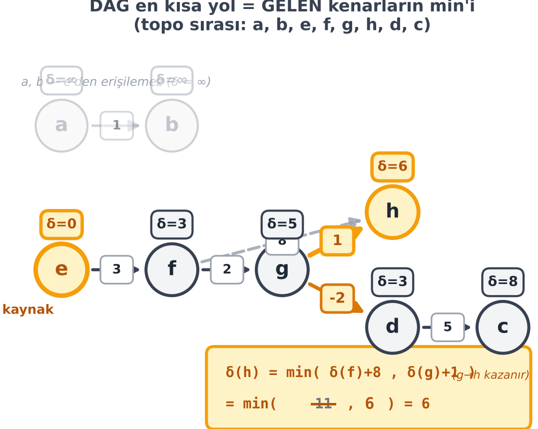
T: alt-problem grafı = G'nin kendisi $\to$ G'nin topolojik sırası. **B:** $\delta(s, s) = 0$. **Time:** $\sum$ (gelen kenar sayısı $+1) = O(V + E)$.

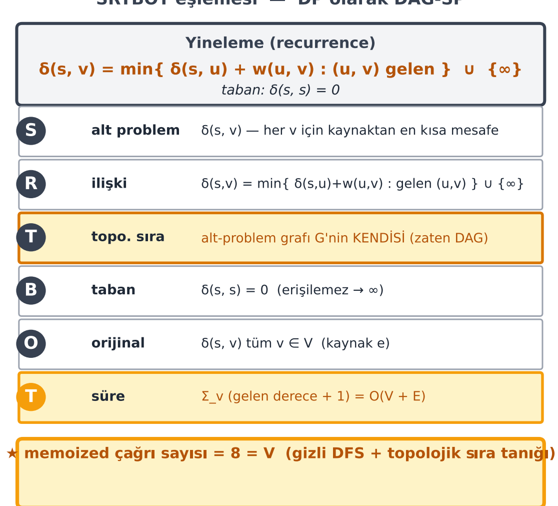
İlginç: generic DP algoritması (memo-kontrol $\to$ özyinele) **alt-problem grafının tersinde bir DFS** çalıştırır; memo tablosu "ziyaret edildi mi" kontrolüdür. Yani DP, içinde DFS + topolojik sıra barındırır — "bedavaya".

Şekil 30.5 bu denkliği Ders 16 örneğiyle kanıtlar: solda 8 düğümlü DAG (topolojik sıra a, b, e, f, g, h, d, c; kaynak e), her düğümün üstünde motordan hesaplanan δ rozeti (e:0, f:3, g:5, h:6, d:3, c:8; $a = b = \infty$ soluk, erişilmez), ve h için gelen-kenar min recurrence'ı vurgulu ($\delta(h) = \text{min}(\delta(f) + 8, \delta(g) + 1) = \text{min}(11, 6) = 6$; kazanan $g \to h$ amber, kaybeden $f \to h$ kesikli + "11" üstü çizik); sağda recurrence kutusu, SRTBOT eşleme tablosu (S/R/T/B/O/T) ve alt tanık rozeti — memoized çağrı sayısı $= 8 = V$ (gizli DFS tanığı), dag_sp_memoized ile dag_relaxation aynı δ'yı BİREBİR verir (D16 çapraz tanık).

Builder Notu — DAG = build sistemi

Alt-problem grafının çevrimsiz (DAG) olması bir tesadüf değil, bir zorunluluktur: çevrim olsaydı bir alt problem kendini beklerdi (sonsuz döngü). Bu yapı gerçek mühendislikte her gün karşına çıkar — **build sistemleri** (Make, Bazel, webpack), paket bağımlılık çözücüler, elektronik tablo hücre yeniden-hesaplaması ve görev zamanlayıcıları hep bir bağımlılık DAG'ını topolojik sırada işler. Memoization'lı DP'nin "her düğümü bir kez çöz, sonucu sakla" mantığı, Make'in "değişmemiş hedefi yeniden derleme" mantığının ta kendisidir; ikisi de aynı gizli DFS'i koşturur.





Şekil 30.5: DAG en kısa yol = Dinamik Programlama (Demaine L15 §7; D16 köprüsü): delta(s,v) = min{ gelen δ(s,u)+w(u,v) } ∪ {∞}. SOL: D16 çizgesi (8 düğüm; topo sırası a,b,e,f,g,h,d,c soldan sağa); her düğümün üstünde δ değeri rozetli (e=0 amber kaynak; a,b=∞ soluk erişilemez). h için GELEN-kenar min recurrence vurgulu: δ(h) = min(δ(f)+8, δ(g)+1) = min(11, 6) = 6 — kazanan g→h amber kalın, kaybeden f→h kesikli + '11' üstü çizik. SAĞ: recurrence kutusu + SRTBOT eşleme tablosu (S alt problem / R ilişki / T topo sıra / B taban / O orijinal / T süre = O(V+E)) + alt rozet (memoized çağrı sayısı = 8 = V gizli DFS tanığı; dag_relaxation ile BİREBİR). Veri MOTORDAN (assert): topo == [a,b,e,f,g,h,d,c]; delta == {a:∞,b:∞,e:0,f:3,g:5,h:6,d:3,c:8}; nsub == 8 == V; delta == dag_relaxation (D16 çapraz tanık); δ(f)+8=11, δ(g)+1=6, δ(h)=min(11,6)=6.

30.9 8. Alt-Problem Tasarım Aracı: Prefix/Suffix/Substring

Girdi bir **dizi** ise, doğal alt problem adayları:

> *"if your input is a sequence, here are some good sub-problems… prefixes… suffixes… substrings."*
> — Demaine, 43:03

- **Önekler (prefixes):** $x[: i]$, her i için — $\Theta(n)$ tane.
- **Sonekler (suffixes):** $x[i :]$, her i için — $\Theta(n)$ tane.
- **Alt diziler (substrings):** $x[i : j]$ — $\Theta(n^2)$ tane.

Alt-dizilim (subsequence) KULLANMA — 2^n tane (üstel). Önek/sonek genelde denktir ve yeterlidir; yetmezse substring'e geç.

Şekil 30.6 bu dört adayı $n = 6$ örnek dizide ("ABCDEF") yan yana sıralar: önek satırı soldan büyür (A, ABC, ABCDEF), sonek sağdan büyür (F, DEF, ABCDEF) — ikisi de yeşil "$\Theta(n)$ — 7 tane" rozetiyle; substring ortadan bir aralık seçer (CD, BCDE, …) amber "$\Theta(n^2)$ — 28 tane"; subsequence atlamalı seçim (A_C_E, …) kırmızı üstü-çizik "$2^n = 64$ — YASAK". Sayılar figür içinde gerçek enumerasyondan hesaplanır ($n + 1/n + 1/(n + 1)(n + 2)/2/2^n$).

30.10 9. Bowling Problemi: SRTBOT Uygulaması

Problem. n pin, pin i değeri v_i. Bir pin vurursan v_i puan; bitişik iki pini $(i, i + 1)$ birlikte vurursan $v_i \cdot v_{i+1}$ puan. Skoru maksimize et (pinleri atlamak serbest).

Çalışılan Örnek — suffix DP. S: $B(i) = $ pin $i \ldots n - 1$ ile elde edilebilecek maksimum skor. **O:** $B(0)$. **R:** ilk pin i'ye üç şey yapabilirim — atla, tek vur, ya da $i + 1$ ile çiftle:

$$B(i) = \max\bigl(B(i+1),\ B(i+1) + v_i,\ B(i+2) + v_i \cdot v_{i+1}\bigr)$$

T: i azalan (for $i = n - 1 \ldots 0$). **B:** $B(n) = 0$. **Time:** n alt problem $×, O(1) = $ $O(n)$.

30.11 10. Bottom-Up DP ve Yerel Kaba Kuvvet

SRTBOT'u doğrudan bir **for-döngüsü** algoritmasına çevirebiliriz (bottom-up): taban durumu yaz, topolojik sırada döngü kur, recurrence'ı uygula, sonunda orijinali döndür.

Bu, **yerel kaba kuvvettir (local brute force)**: pin i için *tüm* seçenekleri (3 tane) dene, en iyisini al. Normalde $3 \times 3 \times \ldots$ üstel olurdu; ama alt problemleri yeniden kullandığımızdan **doğrusal**.

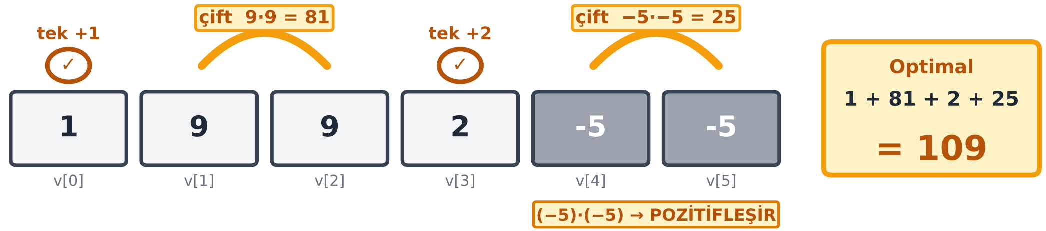
> *"dp is essentially an idea of using local brute force."* — Demaine, 54:47

DP sezgisi: "**çözümün hangi özelliğini bilsem işim biterdi?**" Bu özelliğin seçenek sayısı polinomsa, polinom bir DP elde edersin. (Sonek $\to$ ilk öğeyi düşün; önek $\to$ son öğe; substring $\to$ ortadaki öğe.)

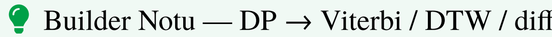
> *"identify some feature of the solution that if we knew that feature we would be done."* — Demaine, 56:00

Şekil 30.6: Alt-problem tasarım aracı: dizi girdide hangi alt problemi seçeyim? (Demaine L15 §8, 43:03; n=6 'ABCDEF'). 4 satır: PREFIX (önekler x[:i]) soldan büyüyen A / ABC / ABCDEF + yeşil 'Θ(n) — 7 tane'; SUFFIX (sonekler x[i:]) sağdan büyüyen F / DEF / ABCDEF + yeşil 'Θ(n) — 7 tane'; SUBSTRING (alt diziler x[i:j]) ortadan CD / BCDE / D + amber 'Θ(n^2) — 28 tane'; SUBSEQUENCE (alt dizilim, atlamalı) A C E / B D F / A F üstü-çizik kırmızı + '2^n = 64 — YASAK'. ALT NOT: dizi girdide önce prefix/suffix dene → yetmezse substring → subsequence ASLA (Demaine 43:03); n+1 / n+1 / (n+1)(n+2)/2 polinom çözülebilir, 2^n üstel DP tablosu patlar. Veri figür içinde GERÇEK enumerasyondan (assert): prefix = n+1 = 7; suffix = n+1 = 7; substring = (n+1)(n+2)/2 = 28; subsequence = 2^n = 64; sıralama prefix=suffix < substring < subsequence.

Şekil 30.7 bu dersin ikinci imza figürüdür ve bowling'i uçtan uca çözer: üstte pin dizisi $v = [1, 9, 9, 2, -5, -5]$ ve optimal plan izi (tek vuruşlar halka, çift kemerler amber yay; "$(-5) \cdot (-5) \rightarrow$ POZİTİFLEŞİR" rozeti) — toplam $1 + 81 + 2 + 25 = 109$; altta suffix-DP tablosu B doldurma sırası $i = 6 \rightarrow 0$ (topolojik sıra, sağdan sola amber ok), her hücrede $B[i]$ değeri ve kazanan seçenek. Motor bunu üç bağımsız yoldan teyit eder: bowling_bottom_up$(v) = [109, 108, 43, 27, 25, 0, 0]$, bowling_brute_pairs$(v) = 109$ (bitmask-çift bağımsız tanık), bowling_memo_counted$(v) = (109, 7)$ $(n + 1$ alt problem).



Şekil 30.7: Bowling suffix-DP — pin dizisi + plan izi + B tablosu (Demaine L15 §9-10 İMZA): B[n]=0 taban; i azalan topolojik sıra; her pin 3 seçenek (atla / tek vuruş +v_i / çift kemer v_i·v_{i+1}); negatif çift çarpımla pozitifleşir. ÜST PANEL: pin dizisi v=[1,9,9,2,–5,–5] (negatifler slate dolgu); optimal plan izi — tek vuruşlar halka (✓ +1, +2), çift kemerler amber yay (9·9=81, (–5)·(–5)=25) + '(–5)·(–5) → POZİTİFLEŞİR' rozeti; sağ toplam kutusu 1 + 81 + 2 + 25 = 109. ALT PANEL: suffix-DP tablosu B 7 hücre i=0..6, doldurma yönü i=6→0 (topolojik sıra, sağdan sola amber ok); her hücre B[i] değeri + kazanan seçenek (← çift/tek/atla) + formül; B[0]=109 CEVAP amber vurgu, B[6]=0 taban (boş sonek). ALT NOT: her pin 3 seçenek, alt problemler YENİDEN KULLANILIR → 3^n kaba kuvvet yerine O(n); yerel kaba kuvvet sabit ≤3 seçenek/pin (Demaine 54:47). Veri MOTORDAN (assert): v=[1,9,9,2,–5,–5]; bowling_bottom_up(v)=[109,108,43,27,25,0,0]; plan=[(0,tek,1),(1,çift,81),(3,tek,2),(4,çift,25)]; bowling_brute_pairs(v)=109 (bağımsız); bowling_memo_counted(v)=(109,7); plan toplam=B[0]=109.


Builder Notu — DP → Viterbi / DTW / diff

"İlk öğeye ne yapabilirim?" + memoization şablonu, bowling'in çok ötesine geçer. Aynı yerel-kaba-kuvvet refleksi gerçek ML ve sistem araçlarının çekirdeğidir: **Viterbi** (gizli Markov modellerinde en olası durum dizisi — her adımda "bu gözlem hangi durumdan geldi?"), **DTW** (dynamic time warping

— ses/zaman serisi hizalama), **diff** ve **git merge** (en uzun ortak alt-dizilim üzerinden satır eşleme), **sequence alignment** (Needleman-Wunsch, Smith-Waterman — DNA/protein). Hepsi bir dizide ilk/son öğenin sabit sayıda seçeneğini deneyip alt problem çözümlerini yeniden kullanır. Bu dersin bir sonraki adımı (Ders 24, LCS/LIS) tam da bu köprünün ilk taşıdır.

30.12 Bu Dersin Özeti

1. **DP = özyineleme + memoization**; en güçlü tasarım paradigması.
2. **SRTBOT**: Subproblems / Relations / Topological / Base / Original / Time.
3. **Memoization**: alt problem çözümünü sakla $\to$ her biri bir kez $\to$ Fibonacci $O(n)$.
4. **Süre formülü**: $\sum$ alt problemler $\times$ özyineleme-dışı iş.
5. **DAG shortest path** = DP; içinde DFS + topolojik sıra gizli.
6. **Alt-problem aracı**: prefix/suffix $\Theta(n)$, substring $\Theta(n^2)$; subsequence (2^n) YASAK.
7. **Bowling**: suffix DP, $B(i)$ = max(atla, tek, çift) → $O(n)$; yerel kaba kuvvet.

! Tek Bir Cümle

DP, "ilk öğeye ne yapabilirim?" sorusunu polinom sayıda alt problem üzerinde memoization'la çözer: SRTBOT ile alt problemleri tanımla, recurrence ile ilişkilendir, topolojik sırada sakla — yerel kaba kuvvet üstel aramayı polinom zamana indirir.

30.13 Kontrol Soruları

i Soru 1: Fibonacci'nin saf özyinelemesi neden üstel, memoization neden polinom yapar?

Cevap: Saf özyinelemede $f(n)$, $f(n-1)$ ve $f(n-2)$'yi çağırır; bunlar da kendi alt çağrılarını yapar — özyineleme ağacında $f(n-3)$, $f(n-2)$ gibi alt problemler **defalarca** yeniden hesaplanır. $T(n) = T(n-1) + T(n-2) + 1 \approx \varphi^n$ (üstel). Memoization, her alt problemi çözünce bir memo tablosuna yazar; aynı alt problem ikinci kez istendiğinde **yeniden hesaplanmaz**, tablodan döner. Böylece yalnız n farklı alt problem, her biri $O(1) \to O(n)$. Tek değişiklik "hesaplananı sakla/yeniden kullan"dır.

i Soru 2: SRTBOT'un altı adımı nedir, ve hangisi en zordur?

Cevap: **S**ubproblems (alt problemleri tanımla), **R**elations (recurrence ile ilişkilendir), **T**opological order (DAG + çözüm sırası), **B**ase case (taban durum), **O**riginal problem (çözmek istediğin), **T**ime ($\sum$ alt problem $\times$ özyineleme-dışı iş). En zoru genelde **S** (ve onunla bağlı R): doğru alt problemleri bulmak. Demaine'in sezgisi: "çözümün hangi özelliğini bilsem işim biterdi?" — o özelliğin polinom seçeneği varsa, alt problemler doğru kurulur.

Soru 3: Neden dizi problemlerinde alt-dizilim (subsequence) alt problem olarak seçilmez?

Cevap: n öğeli bir dizinin 2^n alt-dizilimi vardır (her öğe ya alınır ya alınmaz) — üstel. Alt problemleri alt-dizilimlerle parametreleştirirsek, alt problem sayısı garantili üstel olur $\rightarrow$ DP'nin tüm avantajı kaybolur. Oysa **önek** ($\Theta(n)$), **sonek** ($\Theta(n)$) ve **alt dizi/substring** ($\Theta(n^2)$) polinom sayıdadır. DP'nin işe yaraması için alt problem sayısı polinom olmalı; bu yüzden dizilerde prefix/suffix/substring tercih edilir, subsequence asla.

Soru 4: Bowling'de B(i) = max(B(i+1), B(i+1)+v_i, B(i+2)+v_i·v_{i+1}) neden doğru? 'Yerel kaba kuvvet' ne demek?

Cevap: İlk pin i'ye yapılabilecek **tüm** şeyler: (1) atla $\rightarrow$ kalan sonek $B(i+1)$; (2) tek vur $\rightarrow$ $B(i+1)+v_i$; (3) $i+1$ ile çiftle $\rightarrow$ $B(i+2)+v_i \cdot v_{i+1}$. Bu üç seçenek tüm olasılıkları kapsar; max alarak en iyisini seçeriz. Geri kalan her durumda kalan **bir sonektir** (daha küçük alt problem) $\rightarrow$ özyineleme geçerli. "Yerel kaba kuvvet": her adımda yalnız ilk öğenin seçeneklerini (sabit sayıda) dene; alt problemler yeniden kullanıldığından, normalde 3^n olacak arama $O(n)$'e iner.

30.14 Egzersizler

Egzersiz 1. Fibonacci'yi hem memoizasyonlu (top-down) hem bottom-up Python'da yaz; her ikisinin de $O(n)$ toplama yaptığını doğrula.

Egzersiz 2. Bowling'i verilen bir örnek dizide (örn. $[1, 9, 9, 2, -5, -5]$) elle $B(i)$ tablosuyla çöz; optimal stratejiyi göster.

Egzersiz 3. DAG en kısa yolu SRTBOT olarak yaz (S/R/T/B/O/T); recurrence'ın gelen kenarlar üzerinden min olduğunu açıkla.

Egzersiz 4. Bir dizi problemi için prefix, suffix ve substring alt problem sayılarını ($\Theta(n), \Theta(n), \Theta(n^2)$) ve subsequence'in neden 2^n olduğunu yaz.

Egzersiz 5. Bowling kuralını değiştir (örn. üç bitişik pin de vurulabilsin); recurrence'a yeni seçeneği ekle ve sürenin hâlâ $O(n)$ kaldığını göster.

30.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 24 (L16) — Dinamik Programlama 2 (Erik Demaine)

Ders 24 (L16): Dinamik Programlama 2 (string alt problemleri, Erik Demaine). DP'yi **dizi/string** problemlerine uyguluyoruz: en uzun ortak alt-dizilim (LCS), en uzun artan alt-dizilim (LIS) gibi klasikler, ve oyun ağaçları. SRTBOT aynı kalır; alt problemler önek/sonek/substring olur, recurrence "son karakteri eşleştir/eşleştirme" gibi yerel kaba kuvvetle kurulur. DP ünitesi burada devam eder (Ders 24-27 = L16-L18; araya **Ders 25 = PS8** girer); blok, Quiz 3 kapsamının (DP) belkemiğidir ve **Ders 30 = PS10/Quiz 3 Gözden Geçirme** ile özetlenir.
Ders 24 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (Fibonacci iki yöntem) ve 2 (bowling) çöz.
- SRTBOT'un altı adımını ezberden say; her birine bowling'den örnek ver.
- Ana cümleyi tekrar oku: *"Çözümün hangi özelliğini bilsem işim biterdi? — polinom seçenek → polinom DP."*

30.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Dinamik programlama	Özyineleme + memoization; polinom tasarım	Böl. 1
SRTBOT	Subproblems/Relations/Topological/Base/Original/Time	Böl. 2
Memoization	Alt problem çözümünü sakla/yeniden kullan	Böl. 5
Süre formülü	$\sum$ alt problem $\times$ özyineleme-dışı iş	Böl. 6
Alt-problem aracı	prefix/suffix $\Theta(n)$, substring $\Theta(n^2)$; subsequence 2^n YASAK	Böl. 8
Bowling recurrence	$B(i)$ = max(atla, tek v_i, çift $v_i \cdot v_{i+1}$)	Böl. 9
Bottom-up DP	Base $\to$ topolojik for $\to$ relation $\to$ original	Böl. 10
Yerel kaba kuvvet	İlk öğenin seçeneklerini dene + max/min; reuse	Böl. 10

30.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu ders, hazır algoritma uygulamaktan kendi algoritmanı tasarlamaya geçişin kapısıdır; SRTBOT disiplini ve memoization sezgisi, ML ve sistem mühendisliğindeki çok sayıda araca doğrudan bağlanır — köprülerin özeti:

1. **DP** → dizi hizalama (DNA/diff), Viterbi/DTW (ses, ML), metin akıllı-kırpma, knapsack, en kısa yol.
2. **SRTBOT** → algoritma tasarım disiplini; her DP'yi altı adımla yazma alışkanlığı (OMSCS CS 6515 — her DP probleminde "önce S/R/T/B/O/T çıkar" refleksi).
3. **Memoization** → caching/önbellekleme; saf hesaplamayı tekrar etmeden sakla.
4. **Yerel kaba kuvvet** → "her adımda sabit seçenek + reuse" → üstel aramayı polinoma indirme.
5. **Alt-problem grafı = DAG** → bağımlılık çözümü, build sistemi; DP içinde gizli DFS.
6. **Optimal alt yapı** → en kısa yol + DP köprüsü; alt-problem çözümlerini birleştirme.

❗ Tek bir şey alıp gideceksen

Dinamik programlama = özyineleme + memoization. SRTBOT'la alt problemleri tanımla, bir recurrence ile ilişkilendir, topolojik sırada çöz, çözümleri bir memo tablosunda sakla. Sihir "yerel kaba kuvvettedir": her adımda yalnız ilk öğenin sabit sayıda seçeneğini dener, en iyisini alırsın — alt problemleri yeniden kullandığından, üstel görünen arama polinom zamana iner. Tek soru: "çözümün hangi özelliğini bilsem işim biterdi?" Bu, kendi algoritmanı tasarlama bölümünün açılışıdır; sırada DP'yi string ve oyunlara taşıyan Ders 24 var.

31 Dinamik Programlama 2: LCS, LIS, Oyunlar

DP'yi tek diziden ötesine taşıyan üç teknik: çoklu girdide alt problem uzaylarını çarp (LCS), naif tanım çökerse bir kısıt ekleyip alt problemi genişlet (LIS), oyunlarda durumu koordinata taşı ve ben max rakip min oyna (para oyunu) — hepsi yerel kaba kuvvetle ve ebeveyn işaretçileriyle çözümün kendisini de kurtararak $O(n^2)$

Oturum bilgisi

- **Demaine'in videosu:** YouTube — Lecture 16: Dynamic Programming, Part 2 (≈58 dk)
- **OCW sayfası:** MIT 6.006 Lecture 16: Dynamic Programming, Part 2
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 24 (L16)
- **Hoca:** Erik Demaine (dinamik programlama; **DP serisinin 2/4'ü**)
- **Okuma süresi:** ≈27 dk

Bu, DP ünitesinin **ikinci dersidir**. Ders 23 SRTBOT + memoization'ı tek dizide kurdu; bu ders çerçeveyi zorlayan üç klasik problemle yeni teknikleri tanıtır: **LCS** (en uzun ortak alt-dizilim), **LIS** (en uzun artan alt-dizilim) ve **değişen para oyunu** (alternating coin game). Temel sezgi sabit kalır: "çözümün hangi özelliğini bilsem işim biterdi?" — o özelliği **yerel kaba kuvvetle** dene.

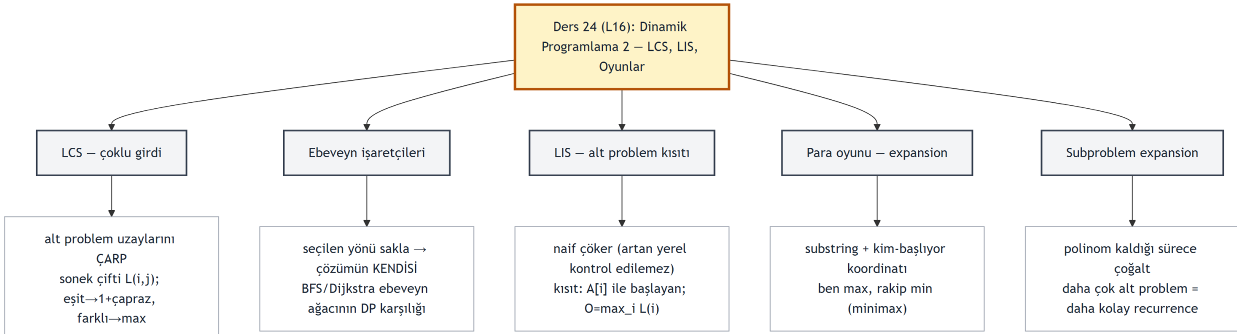
31.1 Bu Derste Ne Var?

DP serisinin **2/4'ü** (Erik Demaine). Üç klasik örnekle dört yeni DP fikri tanıtılır: **çoklu dizi** (tek değil), **substring** alt problemleri, **ebeveyn işaretçileri** (çözümü kurtarmak) ve **alt problem kısıtı/genişletmesi** (subproblem constraint/expansion).

> *"We're now in step two out of four."* — Demaine, 0:19

Üç ana fikir:

1. **Çoklu girdi → alt problem çarpımı** — iki dizi varsa, alt problem uzaylarını çarp (A'nın sonekleri × B'nin sonekleri).
2. **Alt problem kısıtı** — naif tanım recurrence vermiyorsa, alt probleme bir koşul ekle (örn. "A[i] ile başla").
3. **Subproblem expansion** — kısıt için alt problem sayısını çoğalt (polinom kalmak şartıyla); oyunlarda max ile min yer değiştirir.



Şekil 31.1: Ders 24'ün (L16) kavram haritası: kök = Dinamik Programlama 2 (Demaine) — DP serisinin 2/4'ü; DP'yi tek diziden çoklu girdiye, string'e ve oyunlara taşır. Dört dal — (1) LCS = çoklu girdi: iki dizi varsa alt problem uzaylarını ÇARP, her alt problem bir sonek çifti L(i,j); eşit harf 1 artı çapraz değiştirme argümanıyla, farklı harf max iki seçenek; O(n kare). (2) Ebeveyn işaretçileri: her alt problemde seçilen yönü sakla, L(0,0)'dan tabana yürü, çapraz adımlarda topla; DEĞER değil çözümün KENDİSİ; BFS Dijkstra ebeveyn ağacının DP karşılığı. (3) LIS = alt problem kısıtı: naif tanım çöker çünkü artan kısıtı yerel kontrol edilemez; kısıt ekle L(i) eşittir A[i] ile başlayan en uzun artan, O eşittir max i L(i); O(n kare). (4) Para oyunu = subproblem expansion: substring alt problemleri artı kim başlıyor koordinatı; ben max rakip min minimax; O(n kare). Birleştirici tema: bariz alt problemler yetmezse uzayı polinom kaldığı sürece çoğalt — daha çok alt problem eşittir daha çok kısıt eşittir daha kolay recurrence.

Builder Notu — LCS = diff / git'in kalbi

LCS soyut bir bulmaca değil; her gün kullandığın araçların çekirdeğidir. `diff` ve `git merge`, iki dosyayı satır dizileri olarak alıp **en uzun ortak alt-dizilimi** bulur — eşleşen satırlar "ortak", eşleşmeyenler "eklendi/silindi" olur. Aynı recurrence, **DNA/protein hizalamasında** (Needleman-Wunsch, Smith-Waterman), **sürüm karşılaştırmasında** ve **plagiarism tespitinde** yaşar. "Eşit harf → eşleştir, farklı harf → birini dışarıda bırakmayı dene" sezgisi, üç-yol birleştirmenin (three-way merge) ve edit-script üretiminin temelidir.

- **İleriye → LCS:** `diff`/git üç-yol birleştirme, DNA/protein hizalama, sürüm karşılaştırma.
- **İleriye → LIS:** patience sorting, zamanlama, kayıt-defteri analizi.
- **İleriye → minimax:** değişen para oyunu = iki-oyunculu minimax; oyun yapay zekâsı (satranç, alpha-beta budama).
- **Geriye → parent pointers (BFS/Dijkstra):** DP'de de çözümü ebeveyn işaretçileriyle geri kur.

Tek cümle: *İki dizi varsa alt problem uzaylarını çarp; naif tanım recurrence vermiyorsa bir kısıt ekleyip alt problem sayısını genişlet (polinom kalsın); oyunlarda "ben max, rakip min" — hepsi SRTBOT + yerel kaba kuvvetle* $O(n^2)$.

31.2 1. DP 2/4: Üç Örnek ve Yeni Fikirler

İlk DP dersi SRTBOT + memoization'ı kurdu. Bu ders, çerçeveyi zorlayan üç problemle yeni teknikleri tanıtır: **çoklu dizi**, **substring**, **ebeveyn işaretçileri** ve **alt problem kısıtı/genişletmesi**. Temel sezgi sabit kalır:

"çözümün hangi özelliğini bilsem işim biterdi?" — o özelliği **yerel kaba kuvvetle** dene.

> *"Whenever there's something I don't know, I'll just brute force it."* — Demaine, 31:58

Her problemde aynı refleks işler: bir alt problemde bilmediğin küçük bir şey varsa (hangi harf eşleşir, ikinci öğe ne, sıra kimde), o şeyin **sabit veya polinom sayıda** seçeneği varsa hepsini dener, alt problem çözümlerini yeniden kullanırsın.

31.3 2. SRTBOT Hatırlatma

Her DP aynı altı adımı izler: alt problemleri (dizi için prefix/suffix/substring) **tanımla**, bir recurrence ile **ilişkilendir** (alt-problem grafı **DAG** olmalı), **topolojik** sırada çöz, **taban** durumu ekle, **orijinali** kur, **süreyi** ($\sum$ alt problem $\times$ özyineleme-dışı iş) hesapla. Yeni bir alt problem türüne ihtiyaç doğdukça **genişlet** — polinom kaldığı sürece.

Bu derste S adımı her seferinde sınanır: tek dizinin prefix/suffix/substring'i yetmediğinde, alt problem uzayını ya **çarparız** (LCS), ya **kısıtlarız** (LIS), ya da bir **koordinatla genişletiriz** (para oyunu).

31.4 3. LCS: Çoklu Girdi → Alt Problem Çarpımı

Problem. İki dizi A, B verilir; ikisinin de **alt-dizilimi** olan en uzun diziyi bul (substring değil — aradan öğe atlanabilir). Klasik örnek: "hieroglyphology" ve "Michelangelo".

Bowling'de tek dizi vardı; burada **iki**. Çözüm — çoklu girdi için genel numara:

> *"subproblems for multiple inputs… We just take the product, multiply the subproblem spaces."*
> — Demaine, 7:04

S: $L(i,j) = A[i:]$ ve $B[j:]$ soneklerinin LCS'i (her alt problem bir **sonek çifti**); $0 \le i \le |A|, 0 \le j \le |B|$. Alt problem sayısı $(|A|+1)\cdot(|B|+1)$ — iki dizi için polinom; ama **n dizi** olsaydı n^n (üstel) olurdu.

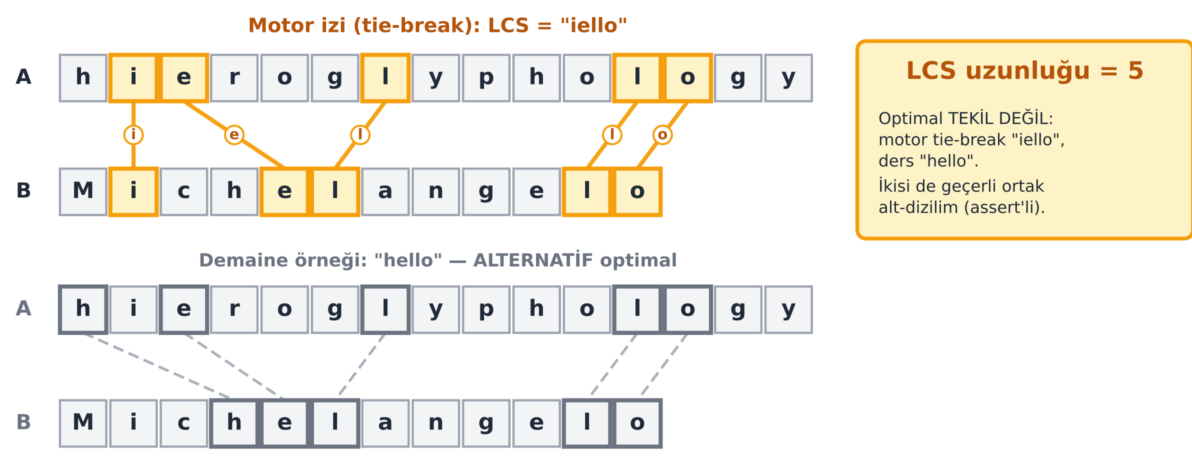
Şekil 31.2 ders örneğini motor üzerinde gösterir: $\text{LCS}(\text{hieroglyphology}, \text{Michelangelo})$ uzunluğu **5**'tir, ama optimal çözüm **tekil değildir** — motorun tie-break izi "iello" iken Demaine'in derste verdiği "hello" da geçerli bir ortak alt-dizilimdir (ikisi de uzunluk 5, `is_subsequence` ile doğrulanır). Bu önemli bir CS nüansıdır: DP'nin pin'lediği şey **değerdir** (uzunluk 5), tek bir temsilci dizgi değil; çözüm kümesi birden çok elemanlı olabilir.

31.5 4. LCS Recurrence: Eşit / Farklı Durumlar

Çalışılan Örnek — iki durum. İlk harflere $A[i]$ ve $B[j]$ bakarız:

- **Farklı ($A[i] \neq B[j]$):** ikisi birden LCS'in ilk harfi olamaz; **en az biri** dışarıda. Hangisi bilinmez, ikisini de dene:

$$L(i,j) = \max\big(L(i+1,j),\ L(i,j+1)\big)$$



Şekil 31.2: LCS(‘hieroglyphology’, ‘Michelangelo’) — optimal TEKİL değil, pin’lenen DEĞER (Demaine L16 §3). ÜST blok: A=hieroglyphology ve B=Michelangelo büyük harf kutuları arası ‘iello’ eşleşmeleri (motor tie-break izi, amber bağlantı çizgileri, soldan-açgözlü konumlar). ALT blok: aynı kelimeler arası ‘hello’ eşleşmeleri (Demaine’in derste verdiği ALTERNATİF optimal, soluk slate kesik çizgiler). SAĞ kutu: LCS uzunluğu = 5; optimal tekil değil — motor ‘iello’, ders ‘hello’, ikisi de geçerli ortak alt-dizilim. ALT NOT: çözüm kümesi geniş olunca pin’lenen şey DEĞER (uzunluk 5), temsilci dizgi değil. Veri MOTORDAN (assert): lcs_reconstruct(‘hieroglyphology’,‘Michelangelo’) == (‘iello’, 5); is_subsequence(‘hello’, her ikisi) True; brute_lcs_length == 5; ‘iello’ != ‘hello’ ama eşit uzunluk.

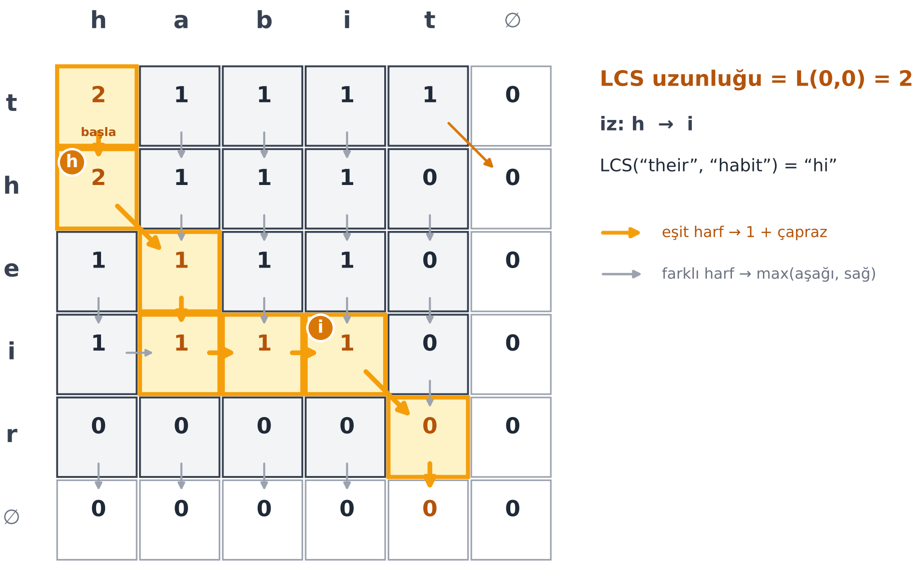
"at least one of Ai and Bj is not in the LCS." — Demaine, 11:24

- **Eşit ($A[i] = B[j]$):** bu harfi **eşleştirmek** her zaman optimaldir (değiştirme argümanı: çaprazlamayan bir eşleştirmede i'yi j ile eşlemek kaybettirmez), 1 puan al ve ikisinden de ilerle:

$$L(i, j) = 1 + L(i + 1, j + 1)$$

T: i, j azalan. **B:** bir dizi tükenirse $L = 0$. **O:** $L(0, 0)$. **Süre:** $\Theta(|A| \cdot |B|)$ alt problem $\times O(1) = O(n^2)$.

Şekil 31.3 bu recurrence'ı küçük bir örnek üzerinde tam DP-grid olarak gösterir (A = their, B = habit): her hücre $L(i, j)$ değerini taşır; **çapraz** (amber) oklar eşit-harf eşleşmesini ($1 + $ çapraz), **aşağı/sağ** (slate) oklar farklı-harf max seçimini gösterir. $L(0, 0) = 2$ ve ebeveyn izi LCS'in kendisini — "hi" — verir; bu §5'in konusu olan ebeveyn işaretçilerinin somut hâlidir.



Şekil 31.3: LCS sonek-çifti DP tablosu + ebeveyn izi (Demaine L16 §4-5 İMZA). A='their' (satır), B='habit' (sütun); her hücre L(i,j) = A[i:] ile B[j:] LCS uzunluğu. ÇAPRAZ amber ok = eşit harf (1+çapraz, değiştirme argümanı: eşleştirmek hep güvenli); AŞAĞI/SAĞ slate ok = farklı harf max(A[i] dışarıda ↓ / B[j] dışarıda →). Amber yol L(0,0)'dan tabana ebeveyn izi; çapraz adımların harf rozetleri (amber daireler) LCS karakterleridir. Sağ üst: LCS uzunluğu = L(0,0) = 2, iz 'h → i', LCS('their','habit') = 'hi'. Veri MOTORDAN (assert): lcs_table + lcs_reconstruct; L(0,0) == 2; lcs_str == 'hi'; is_subsequence('hi', her ikisi); brute_lcs_length == 2; izden toplanan == lcs_str. Alt not: ≠ → max(2 seçenek), = → 1+çapraz; Θ(|A|·|B|).

31.6 5. Parent Pointers ile Çözüm Kurtarma

DP yalnız **uzunluğu** değil, **LCS'in kendisini** de verir. Her alt problemde "hangi seçeneği aldım?" yönünde bir **ebeveyn işaretçisi** (kırmızı/amber ok) çiz; sonda $L(0,0)$'dan tabana izleyip **çapraz kenarları** (harf eşleşmeleri) toplarsan LCS çıkar.

> *"we just follow these pointers backward... and we get our answer."* — Demaine, 21:08

Bu, BFS/Dijkstra'daki ebeveyn ağacının DP karşılığıdır; bugünkü tüm DP'lerde kullanılabilir. Kritik fark: ebeveyn işaretçisi **değeri** (uzunluk) değil, **çözümün kendisini** (dizinin/yolun kendisini) geri kurar.

Şekil 31.4 fikri iki seviyede gösterir: solda kavramsal bir 3×3 DP-şeması (amber çapraz = topla, slate aşağı/sağ = atla; "geriye yürü, çapraz adımlarda topla"); sağda **iki gerçek iz** motordan — LCS "their"×"habit"'in çapraz harfleri "hi"yi, LIS "carbohydrate"'in ebeveyn zinciri (indeks 1→3→4→8→10) "abort"'u verir. Rozet altını çizer: kaydedilen yön DEĞER değil, çözümün KENDİSİDİR.

Şekil 31.4: Ebeveyn işaretçisi: çözümü geri kurma (Demaine L16 §5, 21:08). SOL panel KAVRAM: küçük 3×3 DP-tablo şeması; her hücrede seçilen yönün ebeveyn oku (amber çapraz = TOPLA, slate aşağı/sağ = atla); L(0,0)'dan tabana iz 'geriye yürü, çapraz adımlarda topla'. SAĞ panel İKİ GERÇEK İZ (motordan): (a) LCS 'their'×'habit' çapraz-adım harfleri → 'hi'; (b) LIS 'carbohydrate' ebeveyn zinciri indeksleri 1→3→4→8→10 → 'abort'. Rozet: BFS/Dijkstra ebeveyn ağacının DP karşılığı — kaydedilen yön DEĞER değil, çözümün KENDİSİ. Veri MOTORDAN (assert): lcs_reconstruct('their','habit') == ('hi', 2); çapraz harfleri 'hi'; lis_reconstruct('carbohydrate') == (['a','b','o','r','t'], 5); zincir indeksleri [1,3,4,8,10].

31.7 6. LIS: Naif Tanım Neden Çöker

Problem. Tek dizi; en uzun **kesin artan** alt-dizilim. Örnek: "carbohydrate" → "abort".

Naif **S:** $L(i) = A[i:]$'nin en uzun artan alt-dizilimi. **R denemesi:** $L(i) = \max\big(L(i+1),\ 1 + L(i+1)\big)$ — **çöker**, çünkü "artan" kısıtı hiç uygulanmıyor; $A[i]$'yi eklediğimde $L(i+1)$ zincirinin **ikinci öğesinin** ne olduğunu ($A[i]$'den büyük mü?) bilmiyoruz. "Artan" koşulunu yerel olarak kontrol edemediğimiz için recurrence yazılamaz.

Şekil 31.5 bu çöküşü ve §7'deki çözümü tek figürde toplar: üst panel naif tanımın neden recurrence vermediğini (üstü çizik deneme + "ikinci öğeyi bilmiyoruz") gösterir; alt panel kısıtlı tanımın "carbohydrate" üzerinde nasıl çalıştığını — her harfin altında $L(i)$ değeri ve kazanan zincir a→b→o→r→t (motordan) — verir.

31.8 7. LIS: Alt Problem Kısıtı

Çalışılan Örnek — kısıt ekleme. Çözüm: alt probleme bir **koşul** ekle.

> *"this is the idea of subproblem constraints or conditions."* — Demaine, 28:25

S (kısıtlı): $L(i) = A[i]$ **ile başlayan** (yani $A[i]$'yi içeren) en uzun artan alt-dizilim. **O:** $\max_i L(i)$ (LIS nerede başlar bilmediğimizden hepsinin maksimumu). **R:** $A[i]$ kesin var; ikinci öğe j'yi bilmiyoruz, kaba kuvvetle dene ($i < j$ ve $A[i] < A[j]$ olan tüm j):

$$L(i) = 1 + \max\big(\{\, L(j) : i < j,\ A[i] < A[j] \,\} \cup \{0\}\big)$$

T: i azalan. **B:** $L(|A|) = 0$. **Süre:** n alt problem $\times\, O(n)$ özyineleme-dışı iş $= O(n^2)$. Artan kısıtı, j seçimindeki $A[i] < A[j]$ koşuluyla **yerel** olarak garanti edilir; gerisi tümevarımla artan kalır.

31.9 8. Değişen Para Oyunu: Substring + Genişletme

Problem. $v_0 \dots v_{n-1}$ değerli paralar dizisi; iki oyuncu sırayla **baştaki veya sondaki** parayı alır. Ben (önce başlarım) toplam değerimi maksimize etmek istiyorum. Örnek: $[5, 10, 100, 25]$ — 25'i hemen almak hata (rakip 100'ü kapar); doğrusu 5 ile başlamak, 100'ün sonunda bende kalmasını sağlamaktır.

İki uçtan silme → **substring** alt problemi (prefix de suffix de yetmez; iki uç da değişiyorsa neredeyse her zaman substring gerekir).

Şekil 31.6 oyunu motor üzerinde çözer: ilk hamle **baş** ($v_0 = 5$) almaktır; bu durumda toplam payım $5 + x(1, 3, \text{sen}) = 5 + 100 = 105$, oysa son (25) almak yalnız $25 + x(0, 2, \text{sen}) = 25 + 10 = 35$ verir. Bağımsız bir tanık — üçüncü koordinatsız **fark formülasyonu** — aynı 105'i bambaşka yoldan doğrular: $(\Sigma v + d)/2 = (140 + 70)/2 = 105$.

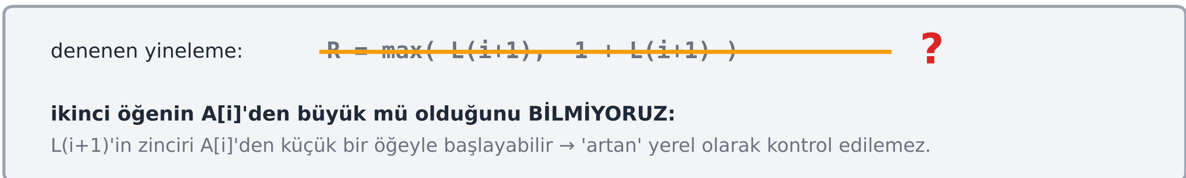
31.10 9. İki Oyuncu: Max/Min Recurrence

Çalışılan Örnek — subproblem expansion. Alt probleme **üçüncü** bir koordinat ekle: $x(i, j, p) = $ "paralar $v_i \dots v_j$ üzerinde, p **oyuncusu başlarsa** benim alacağım maksimum toplam değer" ($p \in \{\text{ben}, \text{sen}\}$). Bir hamle yapınca sıra döner — bu yüzden iki oyuncu durumunu izlemeliyiz.

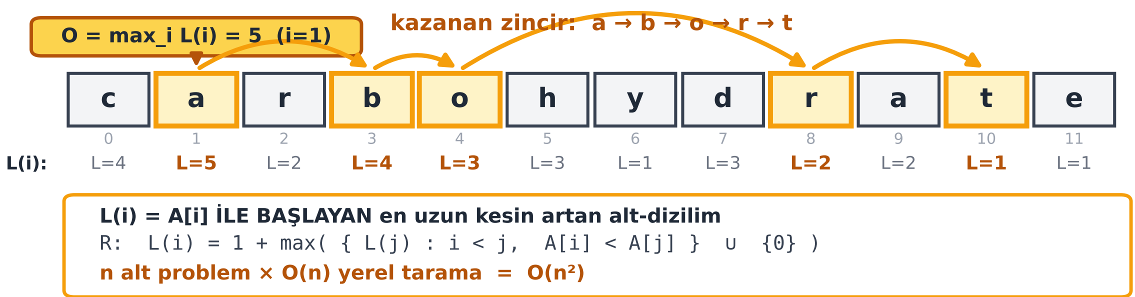
LIS — naif tanım çöker, kısıtlı alt problem kurtarır

Naif tanım çöker

x(i) = A[i:]'in LIS'i deseydik...



Kısıtlı tanım — alt problem genişletmesi



Şekil 31.5: LIS — naif tanım çöker, kısıtlı alt problem kurtarır (Demaine L16 §6-7 İMZA, 28:25). ÜST panel: naif tanım çöküşü — x(i)=A[i:] LIS'i deseydik denenen R = max(L(i+1), 1+L(i+1)) ÜSTÜ ÇİZİK; çünkü ikinci öğenin A[i]'den büyük mü olduğunu BİLMİYORUZ, 'artan' yerel kontrol edilemez. ALT panel: kısıtlı tanım — 'carbohydrate' 12 harf, her harf altında L(i) (motordan: [4,5,2,4,3,3,1,3,2,2,1,1]); kazanan zincir a→b→o→r→t amber oklarla (ebeveyn izi idx 1→3→4→8→10), O = max_i L(i) = 5 rozeti (i=1). Kısıt kutusu: L(i)=A[i] İLE BAŞLAYAN en uzun artan; R = 1 + max({L(j):i<j, A[i]<A[j]} ∪ {0}); n alt problem × O(n) yerel tarama = $O(n^2)$. Veri MOTORDAN (assert): lis_table('carbohydrate') L birebir; lis_reconstruct == (['a','b','o','r','t'], 5); brute_lis_length == 5; zincir [1,3,4,8,10].

- $x(i,j,\textbf{ben})$**:** baştaki (i) veya sondaki (j) parayı al, sonra sıra sende → **max**:

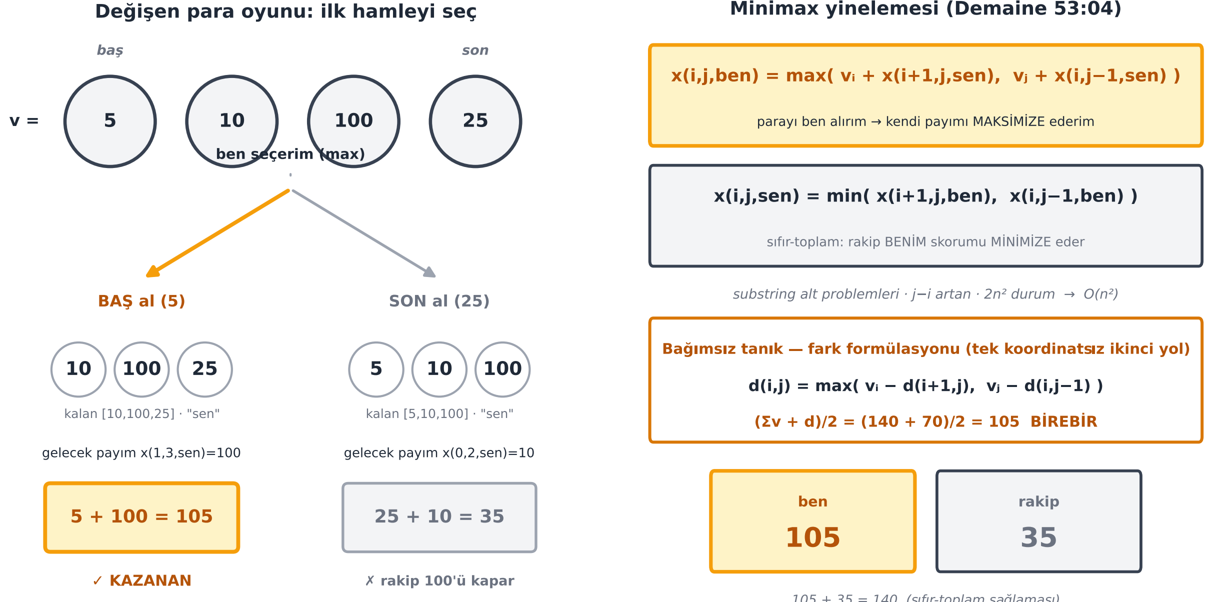
$$x(i,j,\text{ben}) = \max\big(v_i + x(i+1,j,\text{sen}),\ v_j + x(i,j-1,\text{sen})\big)$$

- $x(i,j,\textbf{sen})$**:** sen alırsın (ben puan almam, sıfır-toplam); rakip benim skorumu **minimize** eder:

$$x(i,j,\text{sen}) = \min\big(x(i+1,j,\text{ben}),\ x(i,j-1,\text{ben})\big)$$

> *"when you choose, we end up minimizing, because that's the saddest thing that could happen to us."* — Demaine, 53:04

T: $j-i$ artan. **B:** $x(i,i,\text{ben}) = v_i$; $x(i,i,\text{sen}) = 0$. **O:** $x(0,n-1,\text{ben})$. **Süre:** $\Theta(n^2)$ substring $\times\ O(1) = O(n^2)$ (üçüncü koordinat alt problem sayısını yalnız 2 katına çıkarır).



Şekil 31.6: Değişen para oyunu: minimax (Demaine L16 §8-9 İMZA, 53:04). SOL panel karar ağacı: v=[5,10,100,25]; ilk hamle BAŞ al (5) → 5 + x(1,3,sen)=100 = 105 KAZANAN (amber); SON al (25) → 25 + x(0,2,sen)=10 = 35 HATA (rakip 100'ü kapar, slate). SAĞ panel recurrence kutuları: x(i,j,ben) = max(v_i + x(i+1,j,sen), v_j + x(i,j−1,sen)) parayı ben alır MAKSİMİZE; x(i,j,sen) = min(x(i+1,j,ben), x(i,j−1,ben)) sıfır-toplam rakip MİNİMİZE. Bağımsız tanık — fark formülasyonu d(i,j)=max(v_i−d(i+1,j), v_j−d(i,j−1)); (Σv+d)/2 = (140+70)/2 = 105 BİREBİR. Alt rozet: ben 105 + rakip 35 = 140 (sıfır-toplam sağlaması). substring · j−i artan · $2n^2$ durum → O(n^2). Veri MOTORDAN (assert): coin_game([5,10,100,25]) → x(0,3,ben)=105, first='baş'; x(1,3,sen)=100; x(0,2,sen)=10; coin_game_diff_witness == 105 BİREBİR; d(0,3)=70.

31.11 10. Subproblem Expansion İlkesi

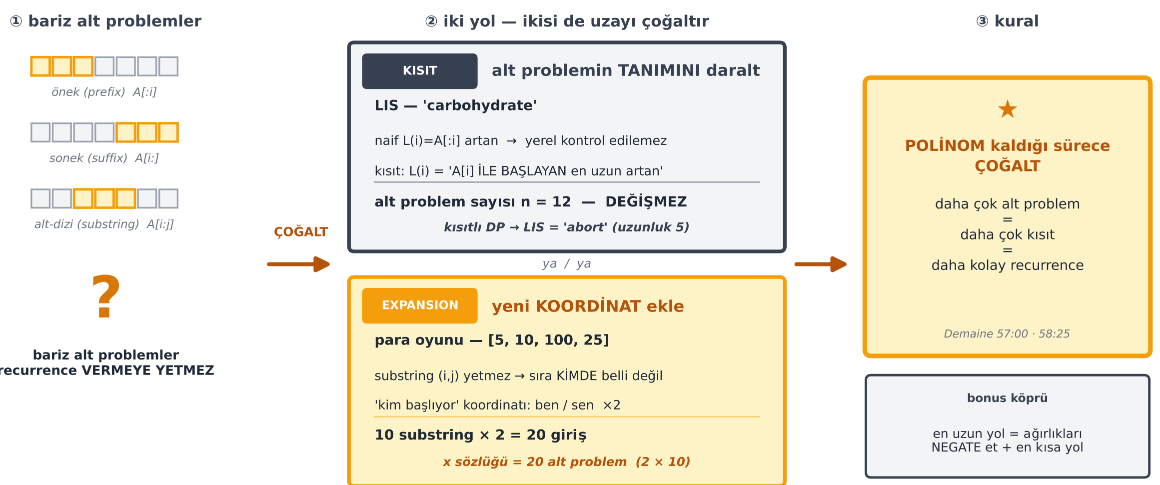
> *"What we're really doing is subproblem expansion."* — Demaine, 57:00

Bariz alt problemler (prefix/suffix/substring) yetmezse, **alt problem sayısını çoğalt** (polinom kaldığı sürece): bir kısıt veya durum koordinatı ekle. Para oyununda "kim başlıyor"u ekleyerek alt problemi 2 katına çıkardık; LIS'te "A[i] ile başla" kısıtını ekledik.

> *"The more subproblems we have, we can consider more constraints."* — Demaine, 58:25

Daha fazla alt problem = daha fazla kısıt = daha kolay recurrence. (Bir bonus: çoğu DP, bir DAG kurup üzerinde en kısa/en uzun yol çalıştırarak da çözülebilir — *en uzun yol = ağırlıkları negate et + en kısa yol* — ama recurrence yazmak çok daha basittir.)

Şekil 31.7 dersin birleştirici temasını tek panelde toplar: solda bariz alt problemlerin (prefix/suffix/substring) recurrence vermeye yetmediği; ortada iki çözüm yolu — **kısıt ekle** (LIS: "A[i] ile başlayan"; alt problem sayısı $n = 12$ DEĞİŞMEZ) ve **expansion** (para: kim-başlıyor koordinatı; 10 substring × 2 = 20 giriş, motordan); sağda kural rozeti ("polinom kaldığı sürece çoğalt") ve en uzun yol bonus köprüsü.



Şekil 31.7: Alt problem EXPANSION — dersin birleştirici teması (Demaine L16 §10, 57:00 + 58:25). SÜTUN 1 'bariz alt problemler YETMEZ': prefix/suffix/substring üç mini ikon + büyük soru işareti (recurrence kapanmıyor). SÜTUN 2 iki çözüm yolu: ÜST (slate) KISIT EKLE → LIS 'carbohydrate', naif L(i)=A[:i] yerel kontrol edilemez, kısıt 'A[i] ile başlayan'; alt problem sayısı n=12 DEĞİŞMEZ → LIS='abort'(5). ALT (amber) EXPANSION → para oyunu [5,10,100,25], substring (i,j) yetmez sıra kimde belli değil, 'kim başlıyor' koordinatı ben/sen ×2; 10 substring × 2 = 20 giriş. SÜTUN 3 kural rozeti: POLİNOM kaldığı sürece ÇOĞALT — daha çok alt problem = daha çok kısıt = daha kolay recurrence; bonus köprü: en uzun yol = ağırlıkları NEGATE et + en kısa yol. Veri MOTORDAN (assert): len(lis_table('carbohydrate')[0])==12; lis_reconstruct=='abort'(5); len(coin_game([5,10,100,25]))==20; distinct (i,j)==10; 10×2==20.

31.12 Bu Dersin Özeti

1. **Çoklu girdi**: alt problem uzaylarını çarp (LCS: sonek çifti); 2 dizi polinom, n dizi üstel.
2. **LCS recurrence**: $A[i] \neq B[j] \to$ max(2 seçenek); $A[i] = B[j] →1 + $ çapraz; $O(n^2)$.

3. **Ebeveyn işaretçileri**: DP'de çözümü (uzunluk değil, dizinin kendisi) geri kur.
4. **LIS naif çöker**: "artan" kısıtı uygulanamaz → alt problem kısıtı şart.
5. **Alt problem kısıtı**: $L(i) = $ "$A[i]$ ile başlayan LIS"; $O = \max_i L(i)$; $O(n^2)$.
6. **Para oyunu**: substring + 3. koordinat (kim başlıyor); ben max, rakip min; $O(n^2)$.
7. **Subproblem expansion**: kısıt için alt problemi çoğalt (polinom kalsın).

! Tek Bir Cümle

DP 2 üç teknik ekler: çoklu girdide alt problem uzaylarını **çarp** (LCS), naif tanım çökerse bir **kısıt** ekleyip alt problemi **genişlet** (LIS), oyunlarda "ben max, rakip min" durumunu koordinata taşı (para oyunu) — hepsi yerel kaba kuvvetle $O(n^2)$, ve ebeveyn işaretçileriyle yalnız değeri değil çözümün kendisini de kurtarırsın.

31.13 Kontrol Soruları

i Soru 1: İki dizili (LCS) bir problemde alt problemler nasıl kurulur, ve neden n dizide bu çalışmaz?

Cevap: Tek dizide prefix/suffix/substring alt problemdi; iki dizide **alt problem uzaylarını çarparız** — örn. A'nın sonekleri × B'nin sonekleri, yani her alt problem bir **sonek çifti** $L(i, j)$. Alt problem sayısı $(|A| + 1) \cdot (|B| + 1) = O(n^2)$, polinom. Sabit sayıda dizi (2, 3, …) için çarpım hâlâ polinom; ama **n dizi** olursa alt problem sayısı n^n (üstel) olur — bu yüzden değişken sayıda dizinin LCS'i için polinom algoritma muhtemelen yoktur.

i Soru 2: LCS'te A[i]=B[j] olduğunda neden 'eşleştir' her zaman doğru, hiçbir max gerekmez?

Cevap: Değiştirme (exchange) argümanı: bir optimal LCS düşün; çaprazlamayan harf-eşleştirmeleridir. $A[i]$ ve $B[j]$ eşitse, optimal çözümde (a) ikisi de eşleşmemişse, onları eşleyip daha uzun bir çözüm elde ederiz (çelişki); (b) $A[i]$ başka bir şeyle eşleşmişse, onu $B[j]$ ile eşlemeye çevirebiliriz — kayıp olmaz. Yani $A[i]$'yi $B[j]$ ile eşleyen bir optimal çözüm **daima vardır**; o hâlde 1 puan alıp $L(i + 1, j + 1)$'e inmek güvenlidir, başka seçeneği denemeye (max almaya) gerek yoktur.

i Soru 3: LIS'in naif tanımı L(i) = A[i:]'nin LIS'i neden recurrence vermez? Kısıt nasıl kurtarır?

Cevap: Naif tanımda $A[i]$'yi çözüme katmak istediğimizde, kalan $L(i + 1)$'in **ilk öğesinin** ne olduğunu bilmeyiz — $A[i] < $ (o öğe) mi? "Artan" koşulunu yerel olarak kontrol edemeyiz, recurrence yazılamaz. Çözüm: alt probleme **kısıt** ekle — $L(i) = $ "$A[i]$ **ile başlayan**" LIS. Artık $A[i]$ kesin dahildir; ikinci öğe j'yi kaba kuvvetle ($i < j$ ve $A[i] < A[j]$ olan tüm j) deneriz, böylece artan koşulu *yerel* olarak garanti edilir. Orijinal problem artık $L(0)$ değil, $\max_i L(i)$ (LIS nerede başlar bilinmez).

i Soru 4: Para oyununda x(i,j,sen) neden min? 'Subproblem expansion' burada ne yaptı?

Cevap: Oyun sıfır-toplamdır: rakip kendi skorunu maksimize ederken **benim** skorumu minimize eder. $x(i, j, \text{sen}) = $ "sen başlarsan benim alacağım max değer" olduğundan, rakibin hamlesini kontrol edemem; en kötü (benim için en az) durumu varsaymalıyım → **min**. Ben seçtiğimde max, rakip

seçtiğinde min — klasik minimax. “Subproblem expansion”: alt probleme **üçüncü koordinat** (kim başlıyor: ben/sen) ekleyerek alt problem sayısını 2 katına çıkardık; bu, “hamle sonrası sıra döner” durumunu temsil edip recurrence’ı çok sadeleştirdi (polinom kaldı: $2 \cdot n^2$ alt problem).

31.14 Egzersizler

Egzersiz 1. LCS’i bir grid (DAG) olarak çiz (örn. “their” vs “habit”); çapraz kenarların eşleşme, diğerlerinin max olduğunu ve ebeveyn izinin LCS’i verdiğini göster.

Egzersiz 2. LIS’i Python’da yaz (kısıtlı alt problem); “carbohydrate” gibi bir örnekte $L(i)$ tablosunu doldur.

Egzersiz 3. LCS recurrence’ının eşit-harf durumunu ($1 + $ çapraz) değiştirme argümanıyla kanıtla.

Egzersiz 4. Para oyununu $[5, 10, 100, 25]$ üzerinde $x(i, j, \text{ben}/\text{sen})$ matrisiyle elle çöz; optimal stratejiyi ebeveyn izinden çıkar.

Egzersiz 5. LIS’i DAG en kısa yola indirge: çapraz kenarlara -1, diğerlerine 0 ver; en kısa yolun en uzun artan alt-dizilime karşılık geldiğini göster.

31.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 25 (PS8) — Problem Oturumu 8 (DP uygulamaları)

Ders 25 (PS8): Problem Oturumu 8 (DP uygulamaları). Bir problem oturumuyla DP’yi pekiştiriyoruz: bu derste öğrenilen çoklu girdi, alt problem kısıtı ve genişletme tekniklerini gerçek problemlerde uygulayacağız. SRTBOT disiplini ve “yerel kaba kuvvet” sezgisi merkezde kalır. **Hoca değişimi notu:** problem oturumlarını **Justin Solomon** yürütür (DP ders bloğu Demaine’in; PS8 Solomon’un oturumudur). DP ünitesi Ders 24-27 (L16-L18) boyunca devam eder ve **Ders 30 = PS10/Quiz 3 Gözden Geçirme** ile özetlenir.
Ders 25 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 2 (LIS) ve 4 (para oyunu) çöz.
- LCS, LIS ve para oyununun SRTBOT’unu ezberden yaz.
- Ana cümleyi tekrar oku: *“Bilmediğin şeyi kaba kuvvetle dene; kısıt için alt problemi genişlet.”*

31.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Çoklu girdi	Alt problem uzaylarını çarp (sonek × sonek)	Böl. 3
LCS recurrence	$\neq \to$ max(2); $= \to 1 +$ çapraz; $O(n^2)$	Böl. 4

Kavram	Tanım	Sayfada
Ebeveyn işaretçileri	DP çözümünü (dizinin kendisi) geri kur	Böl. 5
Alt problem kısıtı	$L(i) = A[i]$ ile başlayan LIS; $O = \max_i L(i)$	Böl. 7
LIS recurrence	$1 + \max\{L(j) : i < j,\ A[i] < A[j]\}$; $O(n^2)$	Böl. 7
Para oyunu	$x(i, j, p)$; ben max, rakip min; substring	Böl. 9
Minimax	Ben seçince max, rakip seçince min	Böl. 9
Subproblem expansion	Kısıt için alt problemi çoğalt (polinom)	Böl. 10

31.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu dersin üç tekniği — çoklu girdi çarpımı, alt problem kısıtı, subproblem expansion — ML ve sistem mühendisliğindeki çok sayıda araca doğrudan bağlanır; köprülerin özeti:

1. **LCS** → `diff`/git üç-yol birleştirme, DNA/protein hizalama, plagiarism tespiti, edit-distance.
2. **LIS** → patience sorting (LIS'i $O(n \log n)$'e indiren kart-yığma algoritması), zamanlama, en uzun uyumlu kayıt dizisi.
3. **Minimax (para oyunu)** → oyun yapay zekâsı (satranç, go); **alpha-beta budama** minimax ağacını gereksiz dalları keserek hızlandırır — para oyununun max/min recurrence'ının doğrudan genişlemesi.
4. **Ebeveyn işaretçileri** → çözüm rekonstrüksiyonu: hizalama, rota, düzenleme betiği (edit script); değil değer, çözümün kendisi.
5. **Subproblem expansion** → durum-augmentasyonu; **OMSCS CS 6515'te DP tasarımının kalbi** — her DP probleminde "bariz alt problem yetmiyorsa hangi koordinatı eklerim?" refleksi.
6. **Çoklu girdi çarpımı** → çok boyutlu DP tabloları; sabit-boyut çarpım polinom, n-boyut üstel.

! Tek bir şey alıp gideceksen

DP 2, tek diziden ötesine geçer. İki dizi varsa alt problem uzaylarını **çarp** (LCS). Naif tanım recurrence vermiyorsa bir **kısıt** ekleyip alt problemi **genişlet** (LIS: "A[i] ile başla"). Oyunlarda durumu (kim başlıyor) koordinata taşı; ben **max**, rakip **min** (minimax). Bilmediğin her şeyi yerel kaba kuvvetle dener, alt problemleri yeniden kullanırsın — hepsi polinom ($O(n^2)$). Ve ebeveyn işaretçileriyle yalnız değeri değil, çözümün kendisini de kurtarırsın.

32 Problem Oturumu 8

Dinamik programlama oturumlarının ilki — SRTBOT ile dört problem: durum izleme, edit distance artı precomputation, LIS-benzeri blok kulesi ve yol sayma

Oturum bilgisi

- **Solomon'un videosu:** YouTube — Problem Session 8 (≈95 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 8
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 25 (PS8)
- **Hoca:** Justin Solomon
- **Okuma süresi:** ≈25 dk
- DP problem oturumlarının **İLKİ**; tema: DP bir algoritma değil bir yaşam tarzı (Solomon 2:35).

32.1 Bu Problem Oturumu Ne Hakkında?

Sekizinci problem oturumu (Justin Solomon), **dinamik programlama** üzerine iki oturumun ilki; dört problemi **SRTBOT** çerçevesiyle çözer (Şekil 39.1). Tema sabit: DP "bir algoritma değil, bir yaşam tarzı" — recurrence yaz, alt problemleri memoize et.

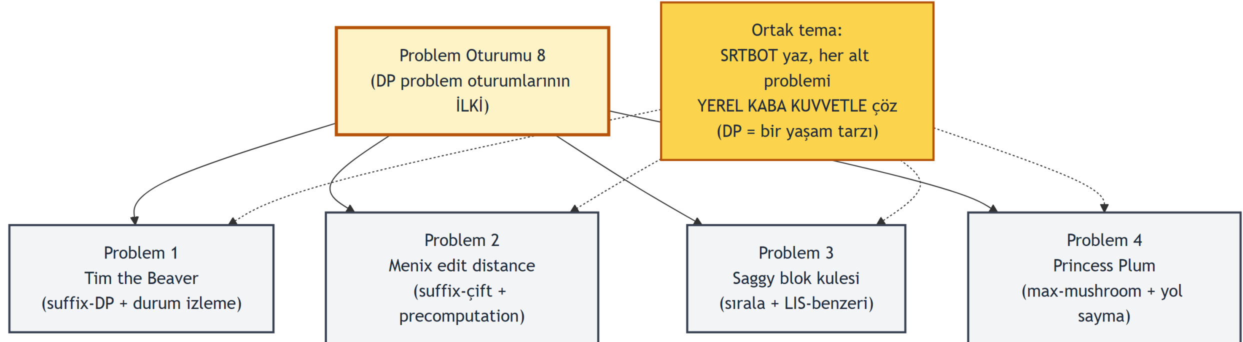
> *"dynamic programming, it's really more of a way of life than any particular algorithm."* — Solomon, 2:35

> *"when you have a recursive call and you repeat something… you might as well remember what you got the last time you saw that input."* — Solomon, 4:46

Dört problemde yeni kavramlar pekişir: alt problem **durum izleme** (kaç gün üst üste), **edit distance** (precomputation'ın çalışma süresine etkisi), **LIS-benzeri** sıralama-temelli DP, ve **yol sayma** (max yerine toplama). Her problem "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir.

Yaklaşım — ortak strateji: SRTBOT yaz, her alt problemi yerel kaba kuvvetle çöz

Dört problemin tamamı aynı refleksle başlar: önce alt problemi (Subproblem) tanımla — tek dizide prefix/suffix, iki dizide sonek çifti, ızgarada hücre. Sonra "bu alt problemde hangi yerel kararı versem geri kalanı bir küçük alt probleme iner?" diye **yerel kaba kuvvet** uygula; bu recurrence'ı (Relation) verir. Kısıt varsa (Tim'in "kaç gün üst üste", Saggy'nin "kesin destek") onu ya alt problem tanımına bir varsayım olarak göm, ya da bir sıralama doğurup LIS'e in. "Kaç yol" sorulduğunda max yerine topla. Bu oturum, DP kasını bu dört SRTBOT'la çalıştırır — Solomon'un deyimiyle DP bir algoritma değil



Şekil 32.1: Problem Oturumu 8'in kavram haritası: kök (PS8) dört probleme dallanır ve ortadaki ortak tema düğümü dördünü birden yönlendirir. Problem 1 Tim'in mutluluğunu suffix-DP ile maksimize eder ve 'kaç gün üst üste' kısıtını alt problem tanımına bir varsayım olarak gömerek durumu izler; Problem 2 Menix'in edit distance'ını suffix-çift DP ile çözer, satır karşılaştırmasının O(k) maliyetini precomputation'la çalışma süresine taşıyıp O(kn + n^2)'ye iner; Problem 3 Saggy'nin blok kulesini üç gözlemle (uzun kenarı hizala, yönelim başına bir kez, kesin destek sıralanabilir kılar) LIS-benzeri bir DP'ye indirger; Problem 4 Princess Plum ızgarasını önce max-mushroom DP'siyle, sonra max yerine toplama yapan ikinci bir yol-sayma DP'siyle çözer. Ortak tema — SRTBOT yaz, her alt problemi yerel kaba kuvvetle çöz — Solomon'un dört probleme de aynı kapıdan girmesini sağlar.

bir yaşam tarzıdır.

32.2 Problem 1: Tim the Beaver — Mutluluk DP

İfade. n gün, gün i sıcaklığı $t(i)$. Dışarı çık → mutluluk $+t(i)$; içeride kal → değişmez. **En fazla 2 gün üst üste** dışarı çıkılır. Mutluluğu maksimize et (ve planı kurtart).

Yaklaşım — durum izleme: kısıtı alt problem tanımına bir varsayım olarak göm

Tek indeks (gün) → **suffix DP**. Kilit incelik: "kaç gün üst üste dışarı çıktım" durumunu açıkça izlemek yerine, alt problemi *"i'de dışarı çıkma izni var"* varsayımıyla tanımla. Böylece "en fazla 2 gün üst üste" kısıtı recurrence'a değil, alt problemin TANIMINA gömülür; üç yerel seçeneğin (içeride / tek gün çık / iki gün çık) her biri ileriyi taze-izinli bir güne devreder, durum bilgisi i'de saklı kalır.

Şekil 32.2 bu suffix-DP'yi motordan **gerçek** verilerle gösterir: örnek $t = [5, -2, 8, 6, -4, 7, 3]$ için optimal plan gün 0'da tek (çık $+5$), gün 1 içeride (soğuk -2 atla), gün 2-3 çift ($8 + 6 = 14$), gün 4 içeride (-4 atla), gün 5-6 çift ($7 + 3 = 10$) → toplam $5 + 14 + 10 = 29$. Bağımsız bitmask tanığı (3 ardışık günü yasaklayan kaba kuvvet) de aynı 29'u verir.

Çözüm. S: $x(i) = $ i'de dışarı çıkma izniyle gün $i \ldots n$ maksimum mutluluk. **R:** üç seçenek:

- içeride kal → $x(i + 1)$ (yarın tam serbest);
- bugün çık, yarın içeride → $t(i) + x(i + 2)$;
- bugün + yarın çık, ertesi içeride → $t(i) + t(i + 1) + x(i + 3)$.

$x(i)$ bu üç seçeneğin maksimumudur. **T:** i azalan. **B:** $x(n+1) = x(n+2) = 0$. **O:** $x(1)$. Planı kurtarmak için her gün hangi seçeneği aldığını sakla, sonra izle.

(İndeks köprüsü: prose, kaynaktaki gibi 1-indekslidir — günler $1 \ldots n$, cevap $x(1)$. Aşağıdaki kod ve figür 0-indekslidir: cevap $x(0)$, günler $0 \ldots n-1$; motor dizisi güvenlik payıyla üç sıfır taban tutar, 1-indeksli gösterimde $x(n+1) = x(n+2) = 0$ yeterlidir.)

```
def tim_happiness(t):                              # O(n) — suffix DP
    n = len(t)
    x = [0] * (n + 3)                              # taban: x(n)=x(n+1)=x(n+2)=0
    for i in range(n - 1, -1, -1):                 # i azalan (suffix sırası)
        x[i] = max(x[i + 1], t[i] + x[i + 2])     # içeride  /  bugün çık
        if i + 1 < n:
            x[i] = max(x[i], t[i] + t[i + 1] + x[i + 3])  # bugün + yarın çık
    return x                                       # x[0] = cevap
```

Karmaşıklık. n alt problem $\times\, O(1) = O(n)$.

32.3 Problem 2: Menix Edit Distance — Precomputation

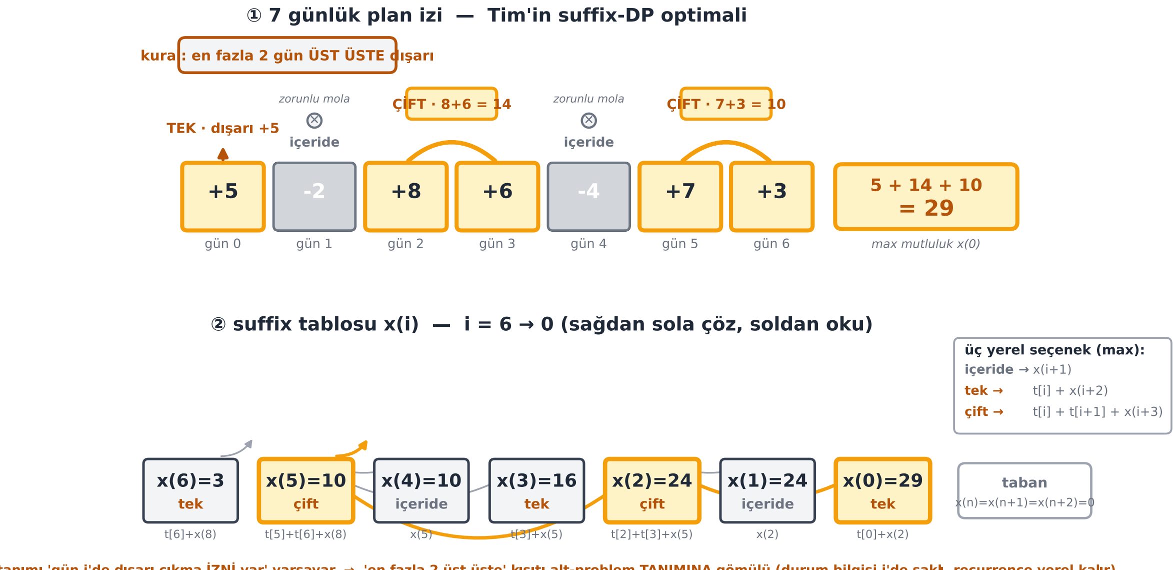
İfade. İki dosya A, B (n satır; her satır uzunluğu $\leq\ k$). Satır **ekle/sil** pahalı (1), bitişik **takas (swap)** bedava (yalnız faydalıysa). Yalnız A düzenlenir. A'yı B'ye çeviren minimum **takas-dışı** işlem sayısı; hedef $O(kn + n^2)$.

> Yaklaşım — klasik edit distance artı bir satır: precomputation'ı çalışma süresine taşı
>
> Klasik **edit distance** DP'si suffix çiftiyle kurulur, üstüne küçük bir ek. Dosyayı *satır satır, sırayla* düzenle (atlamadan) → iki uçtan değil, suffix çift. Asıl incelik karmaşıklıkta: her $A[i] = B[j]$ karşılaştırması iki satırın string eşitliğidir, yani $O(k)$. Naif DP n^2 alt problemin her birinde $O(k)$ iş yapar. Hile: satırları önce $O(k)$ ile bir kimliğe (hash) **ön-işle** (toplam kn), sonra DP'de $O(1)$ karşılaştır (n^2) → $O(kn+n^2)$. Çalışma süresindeki k, satır karşılaştırmasının ipucudur.

"the way that I would edit a file… would be linearly." — Solomon, 39:33

Çözüm. S: $x(i,j) = A[i:] \to B[j:]$ minimum iş. **R** (min, dört durum): $A[i] = B[j] \to x(i+1, j+1)$; $A[i]$ sil $\to 1 + x(i+1, j)$; satır ekle ($B[j]$ eşle) $\to 1 + x(i, j+1)$; takas ($A[i+1] = B[j] \wedge A[i] = B[j+1]$) $\to x(i+2, j+2)$. **T:** $i+j$ artan (2B tabloda sağ-aşağı). **B:** $x(n+1, j) = n+1-j$; $x(i, n+1) = n+1-i$. **O:** $x(1,1)$.

Örnek $A = [\alpha, \beta, \gamma, \delta]$, $B = [\beta, \alpha, \gamma, \varepsilon]$ için cevap 2: takas (α, β) bedava + γ eşle + δ sil + ε ekle. Bu DP'nin doğruluğunu **ikinci bir çerçeve** teyit eder: edit distance, durum (i,j)'leri düğüm, eşle/takas kenarları 0-ağırlık, sil/ekle kenarları 1-ağırlık olan bir **grid-DAG üzerinde en kısa yoldur** (bu, Ders 24'ün LCS-grid'inin ve Ders 23'te kurulan "her DP bir DAG'da en kısa yola indirgenebilir" köprüsünün doğrudan akrabasıdır). Motorda bu grid-DAG'ı $(0,0) \to (n,n)$ koşan bağımsız tanık bir **Dijkstra**'dır; aynı örnekte o da 2 verir, ve önce kimliklenmiş satırlarla koşulan precomputation sürümü de aynı cevabı üretir.



Şekil 32.2: Tim'in suffix-DP optimali — Problem 1 İMZA. Veri MOTORDAN (t=[5,−2,8,6,−4,7,3], x[0..6]=[29,24,24,16,10,10,3], plan [tek@0, çift@2, çift@5], bitmask brute = 29). ÜST: 7 gün şeridi — negatif sıcaklıklar slate dolgulu (soğuk, zorunlu mola), çıkılan günler amber çerçeveli; gün 0 TEK (dışarı +5), gün 2-3 ve 5-6 ÇİFT amber kemerleri (8+6=14, 7+3=10), gün 1 ve 4 içeride; 'en fazla 2 gün üst üste' kural rozeti; sağ toplam kutusu 5+14+10 = 29. ALT: x(i) suffix tablosu i=6→0 (sağdan sola çöz, soldan oku); her hücrede kazanan seçenek (içeride x(i+1) / tek t+x(i+2) / çift t+t'+x(i+3)) + amber ebeveyn okları; üç-seçenek lejantı; taban x(n)=x(n+1)=x(n+2)=0; 'izin VAR varsayımı kısıtı alt-probleme göm' notu. Alt not: Solomon 2:35 'way of life' — kısıtı durum değişkenine taşı, recurrence yerel kalsın; O(n).

```
def edit_distance_swaps(A, B):                    # O(n²) alt problem; precompute → O(kn + n²)
    na, nb = len(A), len(B)
    x = {}
    for j in range(nb + 1): x[(na, j)] = nb - j  # taban: kalanı ekle
    for i in range(na + 1): x[(i, nb)] = na - i  # taban: kalanı sil
    for i in range(na - 1, -1, -1):
        for j in range(nb - 1, -1, -1):
            best = min(1 + x[(i + 1, j)], 1 + x[(i, j + 1)])   # sil / ekle
            if A[i] == B[j]:                                   # eşle (bedava)
                best = min(best, x[(i + 1, j + 1)])
            if i + 1 < na and j + 1 < nb \
                    and A[i + 1] == B[j] and A[i] == B[j + 1]:  # takas (bedava)
                best = min(best, x[(i + 2, j + 2)])
            x[(i, j)] = best
    return x[(0, 0)]
```

Karmaşıklık. n^2 alt problem $\times$ $O(1)$ — *ama* her $A[i] = B[j]$ satır karşılaştırması $O(k)$. Satırları önce $O(k)$ ile bir kimliğe ön-işle (toplam kn), sonra DP'de $O(1)$ karşılaştır $(n^2) \to O(kn + n^2)$. (Bu, "alt problem $\times$ iş" formülüne ön-işlemenin eklendiği nadir bir durumdur.)

32.4 Problem 3: Saggy'nin Blok Kulesi — LIS-benzeri

İfade. n blok, her biri $w \times h \times l$ (istenildiği gibi döndürülebilir; her tipten ≥ 3 adet). Maksimum kule yüksekliği; her blok altındakinde **kesin (strict) destekli** olmalı.

> Yaklaşım — kesin destek bir sıralama doğurur: sırala, sonra LIS-benzeri DP koş
>
> 2^n olası alt küme gibi görünür; ama üç gözlemle **LIS'e** indirger. (1) Uzun-kenarı uzun-kenara hizala $\to$ her taban sıralı bir (küçük, büyük) çifti. (2) Kesin destek $\to$ her **yönelim** (hangi eksen yukarı) en fazla **bir kez** kullanılır $\to$ her bloktan en fazla 3 yönelim. (3) Destek koşulu, taban kenarlarını her katta **azaltır** $\to$ liste **sıralanabilir**. Sıraladıktan sonra "alt blok daha büyük taban" zinciri tam olarak en uzun artan alt-dizilim kalıbıdır; yerel kaba kuvvetle her yönelimi tepe yapıp altına gelebilecek en yüksek alt-kuleyi ara.

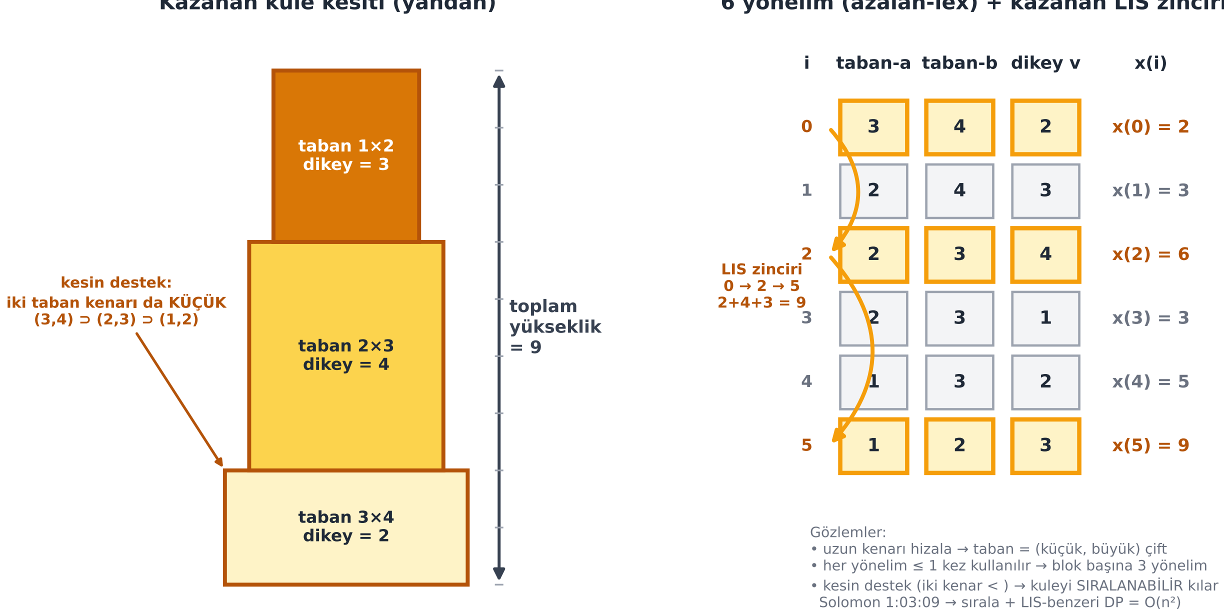
> *"we can sort by the edge lengths because we know that we have this support condition."* — Solomon, 1:03:09

Şekil 32.3 bu indirgemeyi motordan **gerçek** verilerle gösterir: iki blok tipi $\{1 \times 2 \times 3,\ 2 \times 3 \times 4\}$ altı yönelime açılır, taban azalan lexicographic sıralanır; LIS-benzeri DP en yüksek kuleyi 9 bulur — taban $(3, 4)$ dikey 2, üstüne $(2, 3)$ dikey 4, üstüne $(1, 2)$ dikey 3 $(2 + 4 + 3 = 9)$. Bağımsız zincir-DFS tanığı da 9 verir.

Çözüm. Her bloğu sıralı $(w \leq h \leq l)$ 3 yönelime aç, listeyi lexicographic sırala, tekrarları çıkar. **S:** $x(i) = $ blok i'yi **tepe** yaparak ilk i blokla kurulabilen maksimum yükseklik. **R:** $x(i) = v_i + \max(\{x(j) : j < i,\ a_i < a_j,\ b_i < b_j\} \cup \{0\})$, burada (a_i, b_i) yönelim i'nin tabanı ve v_i dikey kenarıdır; koşul $a_i < a_j \wedge b_i < b_j$, blok i'nin blok j'nin üstüne kesin destekle sığması demektir. **O:** $\max_i x(i)$. **B:** $x(1) = v_1$.

```
def tower_height(blocks):                    # O(n log n) + n×O(n) = O(n²)
    ori = block_orientations(blocks)         # sıralı 3 yönelim/blok, lex sıralı
    n = len(ori)
    x = [0] * n
    for i in range(n):
        a, b, v = ori[i]
        best = 0
        for j in range(i):                   # LIS-benzeri tarama
            aj, bj, _ = ori[j]
            if a < aj and b < bj:            # kesin destek: İKİ kenar da küçük
                best = max(best, x[j])
        x[i] = v + best
    return max(x), ori, x
```

Karmaşıklık. Ön sıralama $O(n \log n)$ + n alt problem $\times$ $O(n)$ iş = $O(n^2)$.



i	taban-a	taban-b	dikey v	x(i)
0	3	4	2	x(0) = 2
1	2	4	3	x(1) = 3
2	2	3	4	x(2) = 6
3	2	3	1	x(3) = 3
4	1	3	2	x(4) = 5
5	1	2	3	x(5) = 9

Şekil 32.3: Saggy'nin blok kulesi: sırala + LIS-benzeri DP — Problem 3. Veri MOTORDAN ({1×2×3, 2×3×4} → 6 yönelim, kule = 9, zincir [0,2,5], zincir-DFS brute = 9). SOL: kazanan 3-katlı kule kesiti (yandan) — taban (3,4) dikey 2 (en geniş), üstüne (2,3) dikey 4, üstüne (1,2) dikey 3; genişlikler taban-a ile orantılı KESİN küçülür ((3,4)⊃(2,3)⊃(1,2)); sağda yükseklik ölçek çubuğu toplam = 9; sol kenarda 'kesin destek: iki taban kenarı da küçük' notu. SAĞ: 6 yönelimin azalan-lex tablosu (taban-a, taban-b, dikey v, x(i)); kazanan satırlar (i=0,2,5) amber; sola yaylı LIS-zinciri okları 0→2→5 (2+4+3=9); altta üç gözlem + Solomon 1:03:09. Alt başlık: yükseklik = 9.

32.5 Problem 4: Princess Plum Izgarası — Yol Sayma

İfade. $n \times n$ ızgara; her hücre mushroom / ağaç / boş. Sol-üstten sağ-alta **hızlı yol** ($2n - 1$ hücre — yalnız aşağı/sağ; ağaca girilmez). (a) Toplanabilecek **maksimum mushroom** k; (b) k mushroom toplayan **hızlı yol sayısı**.

Yaklaşım — iki ardışık DP: önce hedefi hesapla, sonra max yerine toplayarak say

“Hızlı” = tam $2n - 1$ hücre → yalnız **aşağı ve sağ** (yukarı/sol path’i uzatır). Izgara zaten bir DAG (sağ-aşağı). İki ardışık DP gerekir. (a) max mushroom için klasik ızgara DP: her hücreye gelen iki komşudan en iyisini al. (b) “kaç yol” sorusu kritik bir değişiklik ister: recurrence’da **max yerine toplama** — bir komşu optimal mushroom sayısına ulaşıyorsa onun yol sayısını ekle. Pozitif sayılar yalnız **taban durumdan** doğar; 1’lerin kaynağı $x(0, 0) = 1$’dir.

> *“she can only move down and to the right because if she moved up or to the left, her path would no longer be called quick.”* — Solomon, 1:21:32

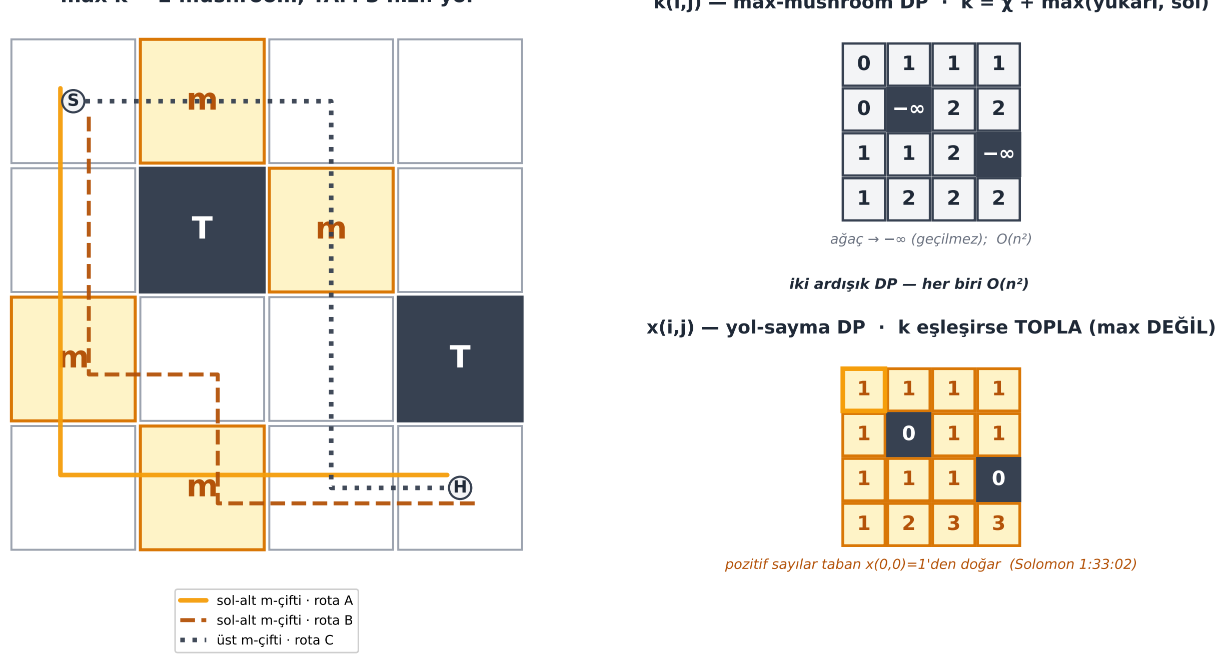
Şekil 32.4 bu iki DP’yi motordan **gerçek** verilerle gösterir: 4×4 örnekte max mushroom $k = 2$ ve onu toplayan **tam 3** hızlı yol vardır. Bağımsız tüm-monoton-yollar tanığı da $(2, 3)$ verir; ek bir kontrol olarak tamamen kapalı bir 2×2 ızgara $k = -\infty$ ve yol sayısı 0 döndürür (taban dışında pozitif sayı doğmaz).

Çözüm. (a) max mushroom: $k(i, j) = \chi(i, j) + \max(k(i-1, j),\ k(i, j-1))$; ağaç $\to -\infty$. ($ = $ hücrede mushroom varsa 1, yoksa 0.) Base: $k(1, 1) = 0$. **(b) yol sayısı:** $x(i, j) = (i, j)$’de biten, $k(i, j)$ mushroom toplayan hızlı yol sayısı. Soldan ve/veya yukarıdan gelen bir komşu için, o komşunun k değeri artı $\chi(i, j)$ tam olarak $k(i, j)$’ye eşitse, o komşunun yol sayısını $x(i, j)$’ye **ekle** (**max değil, toplama** — yol sayıyoruz). Base: $x(1, 1) = 1$. **O:** $x(n, n)$. Pozitif sayılar yalnız **taban durumdan** doğar.

> *“all of the reason why the positive numbers appear in this problem is from the base case.”* — Solomon, 1:33:02

```
def plum_count_paths(grid):                          # (b) yol-sayma DP
    n, m = len(grid), len(grid[0])
    k = plum_max_mushroom(grid)                      # (a) önce max mushroom (ayrı DP)
    x = {}
    for i in range(n):
        for j in range(m):
            if grid[i][j] == "T" or k[(i, j)] == -INF:
                x[(i, j)] = 0; continue
            if i == 0 and j == 0:
                x[(i, j)] = 1; continue              # taban: pozitiflerin TEK kaynağı
            chi = 1 if grid[i][j] == "m" else 0
            cnt = 0
            if i > 0 and k[(i - 1, j)] + chi == k[(i, j)]: cnt += x[(i - 1, j)]  # TOPLA
            if j > 0 and k[(i, j - 1)] + chi == k[(i, j)]: cnt += x[(i, j - 1)]  # max DEĞİL
            x[(i, j)] = cnt
    goal = (n - 1, m - 1)
    return k[goal], x[goal], x
```

Karmaşıklık. İki DP, her biri n^2 hücre $\times$ $O(1) \to O(n^2)$.



Şekil 32.4: Princess Plum: max-mushroom DP → yol-sayma DP — Problem 4 İMZA. Veri MOTORDAN (4×4 grid, max k=2, TAM 3 yol, tüm-monoton-yollar brute = (2,3)). SOL: 4×4 ızgara — mushroom (amber m), ağaç (koyu T, geçilmez), boş (beyaz); S başlangıç, H hedef; ÜÇ optimal yol farklı çizgi stiliyle (her biri 2 mushroom: üst m-çifti 1 rota + sol-alt m-çifti 2 rota). SAĞ-ÜST: k(i,j) max-mushroom DP tablosu ($k = \chi + \max(\text{yukarı, sol})$; ağaç → $-\infty$). SAĞ-ALT: x(i,j) yol-sayma DP tablosu (k eşleşirse TOPLA, max DEĞİL); pozitif sayılar amber, taban x(0,0)=1 amber çerçeveli — pozitiflerin TEK kaynağı (Solomon 1:33:02). Alt: iki ardışık DP, her biri $O(n^2)$.

32.6 Ne Öğrendik?

! Altı Taşınabilir Araç

Bu oturum, DP problem oturumlarının ilkiydi ve SRTBOT çerçevesini dört somut problemde uyguladı:

1. **Durum izleme (Tim):** "kaç gün üst üste" gibi bir kısıtı, alt problem tanımına bir varsayım ("i'de izin var") gömerek izle.
2. **Edit distance:** suffix-çift DP; ekle/sil/eşle/takas durumları min ile; $O(n^2)$.
3. **Precomputation runtime'a girer:** satır karşılaştırması $O(k)$ → önce hash (kn), sonra DP (n^2) = $O(kn + n^2)$; "alt problem × iş" formülüne ön-işleme eklenir.
4. **Sıralama-temelli DP (blok):** kısıt bir sıralama doğuruyorsa, sırala → problem LIS'e iner.
5. **Yol sayma (Plum):** "kaç yol" sorulduğunda recurrence'da **max yerine toplama**; pozitiflik taban durumdan gelir.
6. **İki ardışık DP:** önce hedefi (max mushroom k) hesapla, sonra onu kullanan ikinci DP (yol sayısı).

32.7 Sonraki

Ders 26 (L17): Dinamik Programlama 3 — Demaine

Sırada **Ders 26 (L17): Dinamik Programlama 3** var — Erik Demaine ile, DP'yi **ağaç** ve daha karmaşık alt problemlere taşıyoruz. Bu oturumda gördüğümüz subproblem expansion ve durum izleme, ağaç alt problemlerinde ve pseudopolinom örneklerde derinleşir.

33 Dinamik Programlama 3: Floyd-Warshall, Parantezleme

Subproblem expansion derinliği: alt probleme bir koordinat ekleyerek geçmişi/state'i hatırlamak — Floyd-Warshall vertex-prefix APSP ile ilk k düğümü kullan ve E terimini V'ye çevir O(V^3), parantezlemede son işlemi (kök) tahmin et ve negatifler için min/max genişlet O(n^3), parmaklamada hangi parmakla başladığını state olarak taşı O(n·F^2)

ℹ Oturum bilgisi

- **Demaine'in videosu:** YouTube — Lecture 17: Dynamic Programming, Part 3 (≈63 dk)
- **OCW sayfası:** MIT 6.006 Lecture 17: Dynamic Programming, Part 3
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 26 (L17)
- **Hoca:** Erik Demaine (dinamik programlama; **DP serisinin 3/4'ü**)
- **Okuma süresi:** ≈27 dk

Bu, DP ünitesinin **üçüncü dersidir**. Ders 24 çoklu girdi, alt problem kısıtı ve genişletmeyi tanıttı; bu ders **subproblem expansion**'ı derinleştirir: alt probleme bir kısıt/koordinat ekleyerek geçmişi/state'i hatırlamak. Dört örnek — Bellman-Ford (DP olarak), **Floyd-Warshall** (yeni APSP, $O(V^3)$), aritmetik parantezleme (kök tahmini + min/max), piyano parmaklama (state = parmak). **Tarihsel not:** DP'yi 1950'lerde Bellman icat etti; "programming" eski anlamıyla **optimizasyon**, "dynamic" ise her aşamada farklı davranan yerel kaba kuvvet.

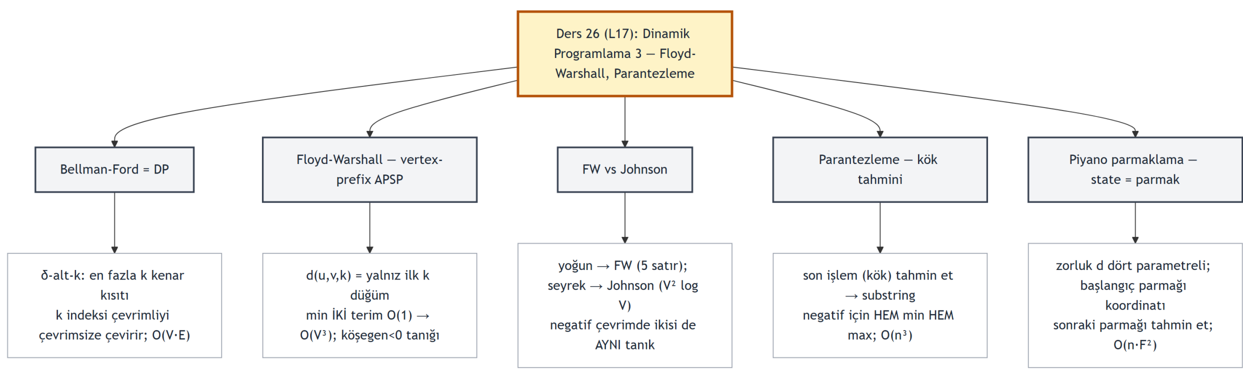
33.1 Bu Derste Ne Var?

DP serisinin **3/4'ü** (Erik Demaine). Tema: **subproblem expansion** (alt problem genişletme) — alt probleme bir kısıt/durum koordinatı ekleyerek "geçmişi/state'i hatırlamak".

> *"we can always add subproblems to make the next step easier."* — Demaine, 2:55

Üç ana fikir:

1. **Floyd-Warshall** — alt problemi "yalnız ilk k düğümü kullanarak" kısıtla → $O(V^3)$ (E faktörünü V'ye çevirir).
2. **Kökü tahmin et** — parantezleme/ifade ağacında **son işlem (kök)** en kolay tahmin edilebilir özelliktir → substring alt problemleri.
3. **State = koordinat** — piyano parmaklamada "hangi parmakla başladım" durumunu alt problem koordinatına taşı.



Şekil 33.1: Ders 26'nın (L17) kavram haritası: kök = Dinamik Programlama 3 (Demaine) — DP serisinin 3/4'ü; tek tema SUBPROBLEM EXPANSION, yani alt probleme küçük bir koordinat/kısıt ekleyerek geçmişi ve state'i hatırlamak. Dört dal — (1) Bellman-Ford DP olarak: δ-alt-k aslında en fazla k kenar kısıtlı bir DP'dir, k indeksi çevrimli grafı çevrimsize çevirir, son kenarı tahmin et, O eşittir V çarpı E; bu k_edge_distances motorunun ta kendisidir. (2) Floyd-Warshall vertex-prefix APSP: alt problem d(u,v,k) yalnız ilk k düğümü ara-düğüm kullanır, k. düğümü ekleyince yol ya geçmez ya BİR KEZ geçer yani min İKİ terim O(1) iş, V küp alt problem çarpı O(1) eşittir O(V küp), negatif çevrim tanığı köşegen d(v,v) sıfırdan küçük. (3) FW karşı Johnson: yoğun grafta FW basit beş satır kod, seyrek grafta Johnson V kare log V kazanır, ikisi de negatif çevrimde AYNI tanığı verir. (4) Parantezleme kök tahmini: son yapılan işlem en kolay tahmin edilen özellik, kökü seç sol ve sağ substring ayrışır; negatif sayılarda iki yanı maksimize çöker çünkü negatif çarpı negatif eşittir büyük pozitif, çözüm her substring için HEM min HEM max sakla, n küp. (5) Piyano parmaklama state eşittir parmak: zorluk d dört parametreli, alt problemi başlangıç parmağıyla kısıtla, sonraki parmağı tahmin et, n F kare. Birleştirici tema: küçük state'i alt problem sayısıyla çarp, geçişleri yerel kaba kuvvetle tara — neredeyse her problemin her yönünü yakalarsın.

Builder Notu — Floyd-Warshall = transitif kapanış / ağ analizi

Floyd-Warshall yalnız APSP mesafe matrisi vermez; ağırlıkları "var/yok" boole'e indirgersen aynı $O(V^3)$ üçlü döngü **transitif kapanışı** (reachability — "hangi düğümden hangi düğüme ulaşılır?") hesaplar. Bu yüzden ağ analizi, sosyal-ağ erişilebilirliği ve derleyici veri-akışı analizinde temel araçtır: dense çizgede 5 satır kod, in-place, basit. Asimptotik olarak Johnson kadar iyi değildir ($V \cdot E$ seyrek çizgede kazanır), ama dense ($E \sim V^2$) veya küçük çizgede pratik üstünlük FW'dedir.

- **İleriye → Floyd-Warshall:** APSP mesafe matrisi, transitif kapanış (reachability), ağ analizi; dense çizgede pratik.
- **İleriye → parantezleme:** derleyici ifade optimizasyonu, **matris-zincir çarpımı** (matrix chain), sözdizimi ağacı.
- **İleriye → parmaklama:** MIDI/müzik teknolojisi, dizilim hizalama, **durum-makinesi DP'leri**.
- **Geriye → Bellman-Ford (Ders 18):** δ_k aslında bir DP alt-problem kısıtıdır.

Tek cümle: *Alt probleme bir koordinat ekleyerek "state" hatırlarsın: Floyd-Warshall "ilk k düğümü kullan" der ($O(V^3)$), parantezleme "son işlem hangisi?" diye kökü tahmin eder (negatifler için min+max), parmaklama "hangi parmak" durumunu taşır — hepsi subproblem expansion.*

33.2 1. DP 3/4: Subproblem Expansion

DP 2'de para oyununda alt problemi 2 katına (kim başlıyor) çıkarmıştık. Bu ders, bu fikri derinleştirir: **alt probleme kısıt/koordinat ekleyerek geçmişi hatırlamak**. Hatırlatma: alt problemleri tanımla (prefix/suffix/substring; çoklu girdide çarpım), "çözümün hangi özelliğini bilsem işim biterdi?" sorusunu kaba kuvvetle dene, DAG + topolojik sıra, taban, orijinal, süre.

> *"we can always add subproblems to make the next step easier."* — Demaine, 2:55

Bu derste bu refleksi dört kez izleriz: Bellman-Ford'u bir DP olarak yeniden okuruz; Floyd-Warshall'da "yalnız ilk k düğüm" kısıtını ekleriz; parantezlemede "son işlem hangisi?" diye kökü tahmin ederiz; parmaklamada "hangi parmak" durumunu taşırız. Her seferinde alt problem sayısını küçük bir faktörle çarpar, recurrence'ı sadeleştiririz.

33.3 2. Bellman-Ford DP Olarak

Bellman-Ford (Ders 18) aslında bir **DP**'dir. **S:** $ _k(s, v) = $ en fazla k kenarlı en kısa $s \to v$ yol (kısıt: k). **R:**

$$\delta_k(s,v) = \min\big(\delta_{k-1}(s,v),\ \min\{\,\delta_{k-1}(s,u) + w(u,v) : (u,v) \text{ gelen kenar}\,\}\big)$$

Son kenar (u,v)'yi tahmin et; yol $\leq k$ kenarsa, kalanı $\leq k-1$ kenar → δ_{k-1}. k **indeksi** çevrimli grafı (G) çevrimsize çevirir: k artan sırada referanslar hep $k-1$'e → acyclic. **O:** $\delta_{V-1}(s,v)$. **Süre:** $\sum_k \sum_v (\text{gelen kenar}) = O(V \cdot E)$.

Şekil 33.2 bu DP'yi motor üzerinde somutlaştırır: D21 örneğinin (a,b,c,d) $\delta_k(a, v)$ tablosu — satırlar $k = 0 \ldots 3$, sütunlar v. Satırdan satıra **düşen** hücreler (amber ok) "bir kenar daha eklendi" demektir; son satır δ_3 tam olarak klasik Bellman-Ford sonucudur ($\{a{:}0, b{:}{-2}, c{:}1, d{:}0\}$). Kritik gözlem: motorun `k_edge_distances` fonksiyonu **bu tablonun ta kendisidir** — Bellman-Ford'un DP iskeleti.

Bellman-Ford bir DP'dir — δₖ(a, v) alt-problem tablosu

k \ v	a	b	c	d
k=0	0	∞	∞	∞
k=1	0	-2	4	∞
k=2	0	-2	1	0
k=3	0	-2	1	0

δ₃ = Bellman-Ford sonucu (sonlu)

SRTBOT eşlemesi

- **S** δₖ(s, v) = en fazla k kenarlı en kısa mesafe — *KISIT = k koordinatı (kenar bütçesi)*
- **R** δₖ(v) = min(δₖ₋₁(v), min_{u→v} δₖ₋₁(u) + w(u,v)) — *kal ya da bir kenar daha gevşet*
- **T** k artan (0 → V−1) — *k boyutu çevrimli grafı ACYCLIC yapar*
- **O** δ_{V−1}(s, v) = SSSP cevabı — *(negatif çevrim yoksa)*

amber ok: δₖ < δₖ₋₁ → bir kenar daha eklendi (yeni gevşeme)

alt-problem genişletmesi ×V → O(V·E)
D18 motoru k_edge_distances BU tablonun ta kendisi

Şekil 33.2: Bellman-Ford bir DP'dir — δ_k(a, v) alt-problem tablosu (L17 §2 + D18 köprüsü). Sol bölge: satırlar k=0..3, sütunlar v=a,b,c,d; her hücre δ_k (∞ soluk). Satırdan satıra DÜŞEN hücreler amber ok ('$\delta_k < \delta_{k-1}$ → bir kenar daha eklendi, yeni gevşeme'). Son satır δ_3 amber kesik çerçeve = klasik Bellman-Ford sonucu (sonlu). Sağ kutu: SRTBOT eşlemesi (S = δ_k en fazla k kenar kısıtı = k koordinatı; R = kal ya da bir kenar gevşet; T = k artan çevrimliyi acyclic yapar; O = δ_{V−1} SSSP cevabı). Alt rozet: alt-problem genişletmesi ×V → O(V·E); motor k_edge_distances BU tablonun ta kendisi. Veri MOTORDAN (assert): k_edge_distances(build_fw_example, 'a', 3); son satır == bellman_ford_classic; tab[0..3] birebir; düşüşler k=1 {b,c}, k=2 {c,d}, k=3 yok.

33.4 3. Floyd-Warshall: Vertex-Prefix APSP

Yeni bir **tüm-çiftler en kısa yol** (APSP) algoritması. Johnson kadar asimptotik iyi değil ama **çok basit** ve dense çizgede mükemmel: $O(V^3)$. Fikir — farklı bir subproblem expansion.

Düğümleri $1 \ldots V$ numarala. **S:** $d(u, v, k) = u \to v$ en kısa yol, **yalnız** $\{u, v, 1 \ldots k\}$ **düğümlerini ara-düğüm olarak kullanarak**.

Bu, "en fazla k kenar" kısıtından farklı bir kısıt: "yalnız ilk k düğüm". Bu, pahalı "tüm gelen düğümler üzerinde döngü"yü (E terimi) ortadan kaldırır.

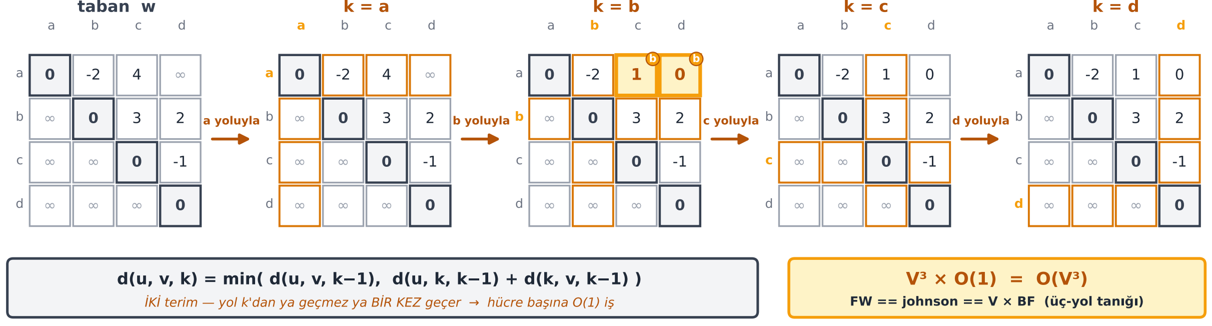
33.5 4. Floyd-Warshall Recurrence ve O(V^3)

Çalışılan Örnek. $d(u, v, k)$'yı hesapla: k. düğümü ekleyince iki seçenek — yol k'dan **geçmez** ya da **geçer**:

$$d(u, v, k) = \min\big(d(u, v, k-1),\ d(u, k, k-1) + d(k, v, k-1)\big)$$

İlk terim: k'yı kullanma. İkinci: $u \to k$ (ilk $k-1$ ile) + $k \to v$ (ilk $k-1$ ile) — basit yol olduğundan k bir kez geçilir. **Min yalnız iki terim** → $O(1)$ iş! **T:** k artan (üçlü iç içe döngü k, u, v). **B:** $d(u, v, 0) = 0$ $(u = v)$ / $w(u, v)$ (kenar varsa) / ∞ (yoksa). **O:** $d(u, v, V)$. **Süre:** V^3 alt problem $\times\, O(1) = O(V^3)$.

Şekil 33.3 recurrence'ı motorun 5 ardışık d-matrisi üzerinde gösterir (taban $\to k = a, b, c, d$): her adımda bir önceki matrise göre **değişen** hücreler amber vurgulanır. Örneğin $k = b$ adımında (a, c): $4 \to 1$ ve (a, d): $\infty \to 0$ — çünkü artık $a \to b \to c$ ve $a \to b \to d$ yolları açılır. Son matris üç bağımsız yoldan doğrulanır: FW = johnson = V × Bellman-Ford (üç-yol tanığı, 60 rastgele çizgede de tutar), ve köşegen $d(v, v) = 0$ (negatif çevrim yok).



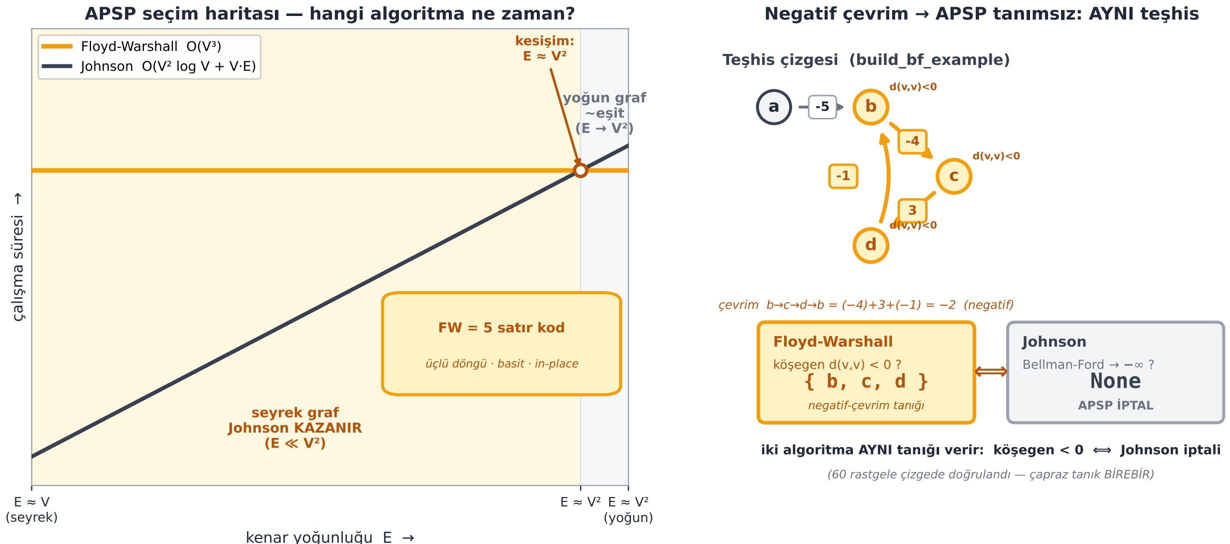
Şekil 33.3: Floyd-Warshall: d(u, v, k) evrimi — taban → k = a, b, c, d (Demaine L17 §3-4 İMZA). 5 mini-matris dizisi (4×4 d matrisi); her adımda bir önceki adıma göre DEĞİŞEN hücreler amber vurgulu + sağ-üst köşede 'k yoluyla' rozeti, köşegen 0 koyu, ∞ soluk; matrisler arası 'k yoluyla' okları. k=b adımında (a,c):4→1 ve (a,d):∞→0 (yol b'den geçer). Altta recurrence kutusu d(u,v,k) = min(d(u,v,k−1), d(u,k,k−1)+d(k,v,k−1)) — İKİ terim, yol k'dan ya geçmez ya BİR KEZ geçer → hücre başına O(1). Sağ rozet: V^3 × O(1) = O(V^3); FW == johnson == V × BF (üç-yol tanığı). Veri MOTORDAN (assert): floyd_warshall_steps(build_fw_example); 5 adım (taban + k=a..d); steps[2] (a,c)=1, (a,d)=0; final == floyd_warshall == johnson == brute_apsp; fw_negative_cycle(final) == [].

33.6 5. Floyd-Warshall vs Johnson

- **Floyd-Warshall:** her zaman $O(V^3)$. Dense çizgede ($E \sim V^2$) Johnson ile aynı ($V \cdot E \sim V^3$), ama daha **basit** (5 satır).
- **Johnson:** $O(V^2 \log V + V \cdot E)$. **Seyrek** çizgede çok daha iyi ($V^2 \log V$).

Kural: çizgenin **dense olduğunu önceden biliyorsan** (veya küçükse) Floyd-Warshall; seyrek/karışıksa Johnson (seyreklikten kazanır). Negatif olmayan ağırlıkta $V\times$ Dijkstra da bir seçenektir.

İki algoritma **negatif çevrim** karşısında da **aynı tanığı** verir: APSP, negatif çevrimi olan bir çizgede tanımsızdır (mesafe $-\infty$'a düşer). Floyd-Warshall bunu köşegende yakalar ($d(v,v)<0$); Johnson süpernode-Bellman-Ford adımında $-\infty$ görür ve `None` döndürerek iptal eder. Şekil 33.4 hem seçim haritasını (yatay FW $O(V^3)$ vs artan Johnson eğrisi) hem de bu ortak teşhisi gösterir: `build_bf_example` çizgesinde ($b\to c\to d\to b=-2$ çevrim) FW köşegeni $\{b,c,d\}$ düğümlerini negatif işaretler, Johnson `None` döner — **köşegen** $<0\Leftrightarrow$ **Johnson iptali**, 60 rastgele çizgede de birebir.



Şekil 33.4: APSP seçim: seyrek → Johnson, yoğun → FW (5 satır), negatif çevrimde ikisi de AYNI tanık (Demaine L17 §5; Johnson algoritmasının kendisi Ders 21/L14, Ku). SOL panel seçim haritası: x = kenar yoğunluğu E (seyrek V'den yoğun V^2'ye), y = süre; Floyd-Warshall yatay O(V^3) (amber, E'den bağımsız), Johnson eğri O(V^2 log V + V·E) (slate, E ile artar); kesişim E ≈ V^2; seyrek bölge 'Johnson KAZANIR', yoğun bölge '~eşit'; 'FW = 5 satır kod' rozeti. SAĞ panel negatif-çevrim teşhisi: build_bf_example çizgesi (çevrim b→c→d→b = (−4)+3+(−1) = −2), çevrim düğümleri {b,c,d} amber 'd(v,v)<0' rozetli; iki teşhis kutusu — Floyd-Warshall köşegen d(v,v)<0 → {b,c,d}; Johnson Bellman-Ford → −∞ → None (APSP İPTAL); ⟺ köprüsü 'köşegen < 0 ⟺ Johnson iptali' (60 rastgele çizgede çapraz tanık). Veri MOTORDAN (assert): floyd_warshall(build_fw_example) == johnson == brute_apsp; fw_negative_cycle == []; floyd_warshall(build_bf_example) köşegen<0 düğümleri == fw_negative_cycle == ['b','c','d']; johnson(build_bf_example) is None.

33.7 6. Aritmetik Parantezleme: Kökü Tahmin Et

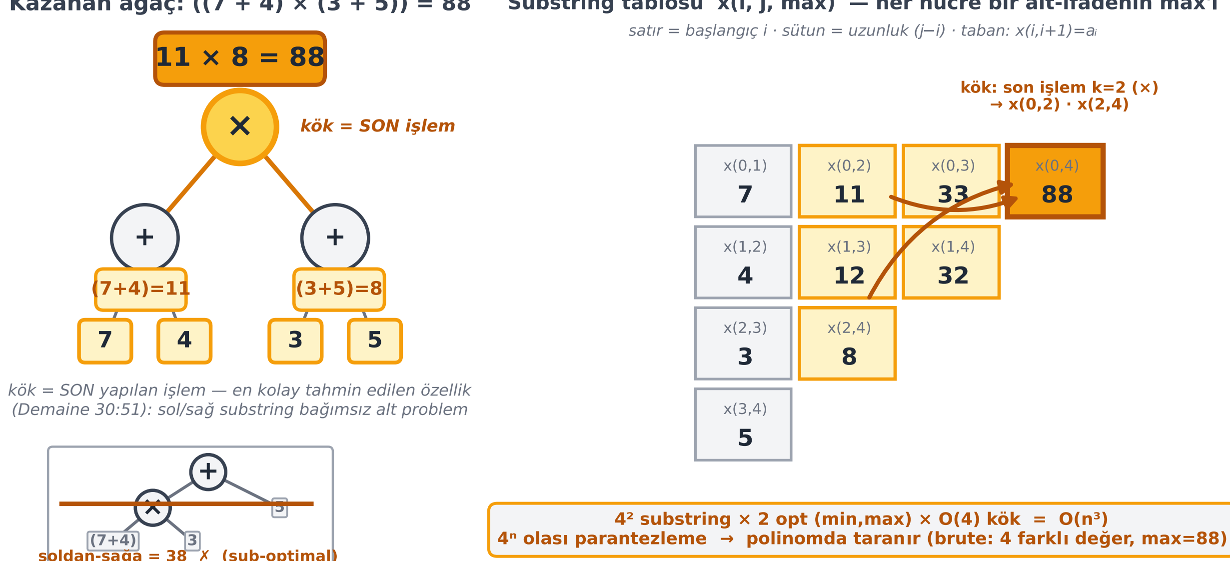
Problem. Formül $a_0 \star_1 a_1 \star_2 \dots a_{n-1}$ ($\star = +$ veya $\times$, a_i tamsayı). Parantezleri istediğin gibi yerleştirip sonucu **maksimize** et. Örnek: $7+4\times 3+5 \to ((7+4)\times(3+5))=88$.

İfade bir **ağaçtır**; en kolay tanımlanabilir özellik **kök** $=$ **son yapılan işlem**.

"guess which operation, star i, is evaluated last — or, in other words, at the root." — Demaine, 30:51

Kök $\star_k$'yı tahmin et → solu (prefix) ve sağı (suffix) ayrışır; ama prefix-of-suffix gerektiğinden alt problem **substring** olmalı (prefix+suffix karışımı → daima substring).

Şekil 33.5 bu fikri motor üzerinde gösterir: kazanan ağaç $((7+4)\times(3+5)) = 88$ — kök $k = 2$ (yani $\times$), sol substring $(7+4) = 11$, sağ substring $(3+5) = 8$, $11 \times 8 = 88$ (Demaine'le birebir). Soldan-sağa naif değerlendirme ($((7+4)\times 3)+5 = 38$) sub-optimaldir. Sağdaki substring tablosu $x(i, j, \max)$ üçgenini (10 hücre) ve kök-seçim oklarını gösterir; brute-force 4 farklı parantezleme değeri ($\{24, 38, 39, 88\}$) verir, $\max = 88$.



Şekil 33.5: Parantezleme: kök-tahmini (SON işlem) → O(n^3) DP (Demaine L17 §6 İMZA, 30:51). SOL panel kazanan ifade ağacı: kök × (amber, 'kök = SON işlem'), sol + = (7+4)=11, sağ + = (3+5)=8, üst rozet 11 × 8 = 88; alt köşede alternatif KÖTÜ ağaç (((7+4)×3)+5) üstü çizik 'soldan-sağa = 38 × sub-optimal'. SAĞ panel substring tablosu x(i,j,max): üçgen yerleşim 10 hücre (satır = başlangıç i, sütun = uzunluk j−i, taban x(i,i+1)=a_i), (0,4)=88 amber hedef; kök-seçim okları (0,4) ← (0,2) ve (2,4) (son işlem k=2 bölünmesi); alt rozet 4^2 substring × 2 opt × O(4) kök = O(n^3), 4^n olası parantezleme polinomda taranır (brute: 4 farklı değer, max=88). Veri MOTORDAN (assert): build_paren_example == ([7,4,3,5],[+,*,+]); paren_reconstruct == ('((7 + 4) × (3 + 5))', 88); choice[(0,4,max)] kök k=2 (×); xmax[(0,2)]=11, xmax[(2,4)]=8, 11×8=88; brute_paren_values max=88, 4 farklı değer; soldan-sağa=38 brute kümesinde.

33.8 7. Negatif Sayılar: Min/Max Genişletmesi

Çalışılan Örnek — neden min de gerek. Sayılar negatif olabilirse, "maksimize için iki yanı maksimize et" çöker: iki **negatif** sayının çarpımı **pozitif büyük** olur. Örn. $7 + (-4) \times 3 + (-5)$.

> *"when we take a product of two negative numbers, we get a positive number."* — Demaine, 34:50

Çözüm: **subproblem expansion** — hem **min** hem **max** sakla. **S:** $x(i, j, \text{opt})$, $\text{opt} \in \{\text{min}, \text{max}\} = a[i..j)$ alt-ifadesinin opt değeri.

> *"If I can solve max and min, I'll know the entire range that I could get."* — Demaine, 35:38

Şekil 33.6 bu işaret-çevirme mantığını motor üzerinde gösterir: kazanan ağaç $(7+((-4)\times(3+(-5)))) = 15$. Kritik adım, sağ alt-ifade $(3 + (-5)) = -2$'nin **minimize** edilmesidir; çünkü $(-4) \times (-2) = +8$ büyük pozitif verir ve $7 + 8 = 15$. "İki yanı da maksimize et" stratejisi burada çöker — onun yerine her substring için $[\text{min}, \text{max}]$ çifti (aralık aritmetiği) saklanır, çarpımda dört kombinasyon denenir.

33.9 8. Parantezleme Recurrence ve O(n³)

R: kök k'yı + sol/sağ için opt_L, opt_R'yi kaba kuvvetle dene:

$$x(i, j, \text{opt}) = \text{opt over } \{\, x(i, k, \text{opt}_L) \star_k x(k, j, \text{opt}_R) : i < k < j,\ \text{opt}_L, \text{opt}_R \in \{\text{min}, \text{max}\}\,\}$$

(Min/max için 4 kombinasyon — çarpımda işaretler karışık, hepsini dene; toplamda yalnız min-min/max-max gerekir.) **T:** $j - i$ artan (substring). **B:** $x(i, i + 1, \text{opt}) = a_i$. **O:** $x(0, n, \text{max})$. **Süre:** n^2 substring $\times$ 2 (opt) $\times\, O(n)$ $(k) = O(n^3)$. (4^n parantezlemeyi polinomda tarıyoruz — subproblem expansion sayesinde.)

Bu, Şekil 33.5 ve Şekil 33.6'in formel özetidir: kökü tahmin etmek substring alt problemlerini doğurur ($O(n^2)$ tane), her birinde min / max ikilemesi ($\times 2$) ve k taraması ($O(n)$) vardır.

33.10 9. Piyano Parmaklama: State = Parmak

Problem. Nota dizisi $t_0 \ldots t_{n-1}$, F parmak. Geçiş zorluğu $d(t, f, t', f')$ (notayı t'yi f parmağıyla, sonra t''yü f' ile çalmanın zorluğu). Toplam zorluğu minimize edip her notaya parmak ata.

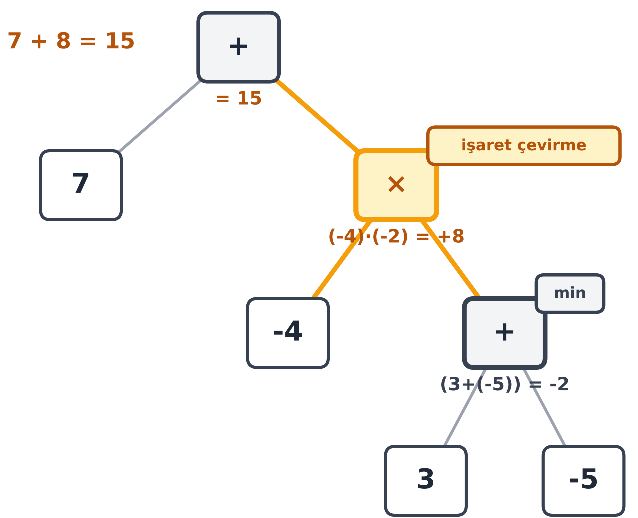
Çalışılan Örnek — state expansion. Naif suffix yetmez: d dört parametreli, "şu anki parmağı" bilmeden iki nota arası zorluğu hesaplayamayız. Çözüm: alt problemi **başlangıç parmağıyla** kısıtla.

S: $x(i, f) = t_i$'yi f **parmağıyla** başlayarak sonek $t_i \ldots t_{n-1}$'i çalmanın min toplam zorluğu (alt problem $\times F$ genişletmesi). **R:** bir sonraki parmak f''yü tahmin et:

$$x(i, f) = \text{min over } f' \ \{\, x(i + 1, f') + d(t_i, f, t_{i+1}, f')\,\}$$

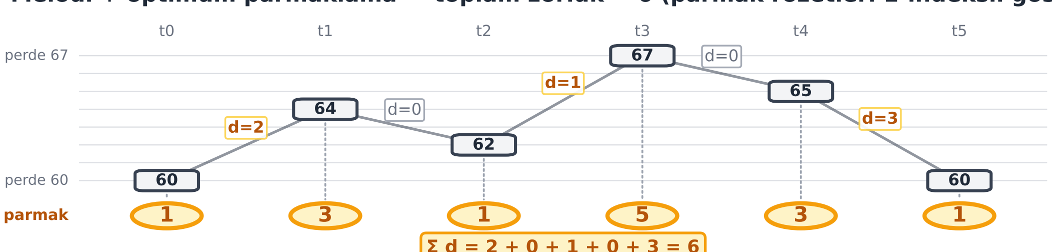
O: $\text{min}_f\, x(0, f)$. **Süre:** $n \cdot F$ alt problem $\times\, O(F) = O(n \cdot F^2)$ (F sabit → doğrusal).

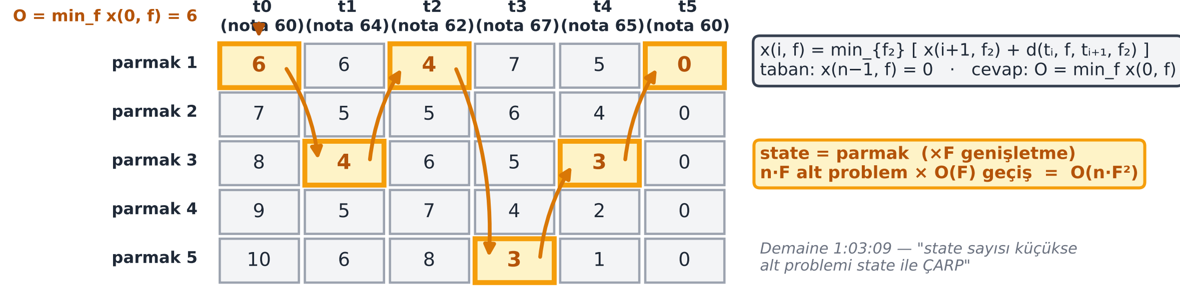
Şekil 33.7 örneği motor üzerinde çözer: 6 notalık melodi $[60, 64, 62, 67, 65, 60]$, $F = 5$. Min toplam zorluk 6, optimal atama $[0, 2, 0, 4, 2, 0]$ (0-indeksli; 1-indeksli gösterimde 1./3./1./5./3./1. parmak), F^n kaba



Şekil 33.6: Negatif sayılar: min/max genişletmesi (Demaine L17 §7-8, 34:50). Kazanan ağaç (7 + ((−4) × (3 + (−5)))) = 15: kök +, sol yaprak 7, sağ alt-ağaç × (‘işaret çevirme’, amber, (−4)·(−2) = +8), onun sağı + ‘min’ rozetli ((3+(−5)) = −2 MİNİMİZE edildi). KRİTİK SEZGİ: max’ı büyütmek için sağ alt-ifadeyi minimize et çünkü negatif × negatif = pozitif büyük; ‘7 + 8 = 15’. Sağ kutu ‘Neden HEM min HEM max?’: iki yanı da maksimize ÇÖKER (Demaine 34:50), pozitif büyük için EN KÜÇÜK (en negatif) çarpan gerekir → her substring için HEM min HEM max sakla (×2 expansion), çarpımda 4 kombinasyon (min·min, min·max, max·min, max·max). Alt not: aralık aritmetiği [min, max] çifti substring’in TÜM ulaşılabilir değerlerini sınırlar. Veri MOTORDAN (assert): build_paren_negative_example == ([7,−4,3,−5],[+,*,+]); paren_reconstruct == (‘(7 + ((-4) × (3 + (-5))))’, 15); xmin[(2,4)] == −2; (−4)·(−2) = 8; 7+8 = 15; choice[(0,4,max)]=(1,min,max); choice[(1,4,max)]=(2,min,min); brute_paren_values max=15.

kuvvet de 6 verir. Alt panel $x(i, f)$ tablosunu (6×5) ve kazanan zincir oklarını gösterir. Bu örnek **sentetik** bir zorluk fonksiyonu kullanır (`default_difficulty` $= | nota - parmak| + $ ters-yön cezası 2); dersin gerçek d'si soyuttur — buradaki amaç state-genişletmesini somutlaştırmaktır, gerçek parmaklama tavsiyesi değil.





	t0 (nota 60)	t1 (nota 64)	t2 (nota 62)	t3 (nota 67)	t4 (nota 65)	t5 (nota 60)
parmak 1	6	6	4	7	5	0
parmak 2	7	5	5	6	4	0
parmak 3	8	4	6	5	3	0
parmak 4	9	5	7	4	2	0
parmak 5	10	6	8	3	1	0

Şekil 33.7: Piyano parmaklama: state = parmak DP (Demaine L17 §9 İMZA, 1:03:09). ÜST panel: 6 nota zaman-çizgisi (y = perde 60-67) + seçilen parmak rozetleri (motor assign'dan, 0-indeksli +1 gösterim, yani 1./3./1./5./3./1. parmak) + ardışık geçiş maliyetleri d (sentetik default_difficulty), Σd = 2+0+1+0+3 = 6. ALT panel: x(i,f) tablosu (6 nota × 5 parmak) motor değerleriyle; kazanan hücre zinciri amber oklar; O = min_f x(0,f) = 6 işareti; recurrence kutusu x(i,f) = min_{f$_2$} [x(i+1,f$_2$) + d(t$_i$,f,t$_{i+1}$,f$_2$)], taban x(n−1,f)=0; O(n·F^2) rozeti (state = parmak ×F genişletme); Demaine 1:03:09 'state sayısı küçükse alt problemi state ile ÇARP'. NOT: d SENTETİK örnek-fonksiyon (dersin d'si soyut). Veri MOTORDAN (assert): build_piano_example == ([60,64,62,67,65,60], 5); piano_fingering → (6, [0,2,0,4,2,0]); brute_fingering == 6; geçiş maliyetleri trans == [2,0,1,0,3], Σ=6; tablo satırları motordan birebir.

33.11 10. Genelleme: Çoklu Nota, Gitar

State'i istediğin kadar zenginleştirebilirsin — yeter ki **state sayısı küçük** olsun.

> *"with subproblem expansion, I can capture almost any aspect of a problem that I want."* — Demaine, 1:03:03

Aynı anda çoklu nota → t^F state ($t = $ anlık maks nota); gitar → ayrıca "hangi tel" seçimi. Her durumda: state'i alt problem koordinatına ekle, geçişleri d ile yakala.

> *"As long as the number of states… is small, I can just multiply the number of subproblems by that state."* — Demaine, 1:03:09

(Bu, çizge çarpımıyla da yapılabilir ama DP metodik bir yol verir.) Şekil 33.7'in tek parmaklı state'i bu genelin en sade hâlidir: F küçük olduğundan $\times F$ genişletme doğrusal kalır; çoklu nota/gitar eklenince state çarpan büyür ama yine polinom kaldığı sürece DP çalışır.

33.12 Bu Dersin Özeti

1. **Subproblem expansion** $=$ alt probleme kısıt/koordinat ekleyip "state" hatırlamak.
2. **Bellman-Ford DP**: δ_k (en fazla k kenar) kısıtı → çevrimliyi çevrimsize çevirir; $O(V \cdot E)$.
3. **Floyd-Warshall**: $d(u,v,k) = $ yalnız ilk k düğüm; min iki terim → $O(V^3)$ (dense'te iyi).
4. **Floyd-Warshall vs Johnson**: dense → FW (basit); sparse → Johnson (hızlı).
5. **Parantezleme**: kökü (son işlem) tahmin et → substring; negatif için min+max; $O(n^3)$.
6. **Piyano parmaklama**: state = başlangıç parmağı; $\times F$ genişletme; $O(n \cdot F^2)$.
7. **Genel ilke**: küçük state'i alt problem koordinatına ekle, geçişleri kaba kuvvetle tara.

> **!** Tek Bir Cümle
>
> Subproblem expansion, alt probleme bir koordinat ekleyerek geçmişi/state'i hatırlatır: Floyd-Warshall "ilk k düğümü kullan" ($O(V^3)$), parantezleme "son işlem hangisi?" (min+max, $O(n^3)$), parmaklama "hangi parmak" ($O(n \cdot F^2)$) — küçük state'i çoğalt, geçişleri tara.

33.13 Kontrol Soruları

> **i** Soru 1: Floyd-Warshall'ın d(u,v,k) kısıtı nedir, ve neden recurrence yalnız iki terim (O(1)) olur?
>
> **Cevap:** $d(u, v, k) = $ "$u \rightarrow v$ en kısa yol, yalnız $\{u, v, 1 \ldots k\}$ düğümlerini ara-düğüm olarak kullanarak". k. düğümü eklerken yol ya k**'dan geçmez** ($= d(u, v, k-1)$) ya da **geçer** ($= d(u, k, k-1) + d(k, v, k-1)$, çünkü basit yolda k bir kez kullanılır ve iki parça da yalnız ilk $k-1$ düğümü kullanır). Bu iki olasılık tüm durumları kapsar → min **iki terimli** → $O(1)$ özyineleme-dışı iş. V^3 alt problem $\times\, O(1) = O(V^3)$. (Bellman-Ford'da "tüm gelen kenarlar üzerinde döngü" E terimi getirirdi; "ilk k düğüm" kısıtı bunu V'ye çevirir.)

> **i** Soru 2: Floyd-Warshall ne zaman Johnson'a tercih edilir, ne zaman edilmez?
>
> **Cevap:** Floyd-Warshall her zaman $O(V^3)$; Johnson $O(V^2 \log V + V \cdot E)$. **Dense** çizgede ($E \sim V^2$) ikisi de $\sim V^3$ ama Floyd-Warshall çok daha **basit** (5 satır) — onu tercih et (veya çizge küçükse). **Seyrek** çizgede ($E \sim V$) Johnson $V^2 \log V$'ye iner, Floyd-Warshall hâlâ V^3 harcar → Johnson çok daha iyi. Kural: dense/küçük biliyorsan FW; seyrek/karışıksa Johnson (seyreklikten kazanır).

Soru 3: Parantezleme'de neden 'kökü tahmin et', ve negatif sayılar için neden hem min hem max gerekir?

Cevap: İfade bir ağaçtır; "ilk işlem" karmaşık (ortada çarpım yapınca üç parça kalır) ama **kök = son işlem** kolayca tahmin edilir — kökü seçmek soldaki ve sağdaki alt-ifadeleri (substring'ler) doğal ayırır. Negatif sayılar için "maksimize = iki yanı maksimize" çöker: iki büyük **negatif** sayının çarpımı büyük **pozitif** olur. Yani bazen alt-ifadeyi *minimize* etmek (çok negatif yapmak) genel maksimumu artırır. Çözüm: her substring için hem **min** hem **max** sakla (subproblem expansion $\times 2$); çarpımda dört min/max kombinasyonunu dene.

Soru 4: Piyano parmaklamada neden alt problemi 'başlangıç parmağı f' ile genişletiyoruz?

Cevap: Zorluk fonksiyonu $d(t, f, t', f')$ **dört** parametreli — iki nota arası geçişin zorluğu, hem şu anki parmağa (f) hem sonraki parmağa (f') bağlı. Naif $x(i) = $ "sonek t_i ...'i en az zorlukla çal" tanımında, recurse ederken **şu anki parmağı** bilmediğimizden d'yi hesaplayamayız. Çözüm: alt problemi başlangıç parmağıyla kısıtla — $x(i, f) = $ "t_i'yi f ile başlayarak…". Artık f bilinir; sonraki parmak f''yü kaba kuvvetle dener, $d(t_i, f, t_{i+1}, f')$'yi hesaplayabiliriz. Alt problem sayısı $\times F$ (sabit) → $O(n \cdot F^2)$, F sabitse doğrusal.

33.14 Egzersizler

Egzersiz 1. Floyd-Warshall'ı küçük bir ağırlıklı çizgede elle çalıştır: $d(u, v, k)$ tablosunu $k = 0 \ldots V$ için doldur.

Egzersiz 2. Floyd-Warshall'ı Python'da yaz (üçlü iç içe döngü); $O(V^3)$ olduğunu ve dense'te $V \cdot E$'ye denk geldiğini göster.

Egzersiz 3. Aritmetik parantezlemeyi $7 + (-4) \times 3 + (-5)$ üzerinde $x(i, j, \min / \max)$ tablosuyla elle çöz; en büyük sonucu bul.

Egzersiz 4. Parantezleme recurrence'ında neden min/max'in 4 kombinasyonunu denemenin yeterli olduğunu (aralık aritmetiği) açıkla.

Egzersiz 5. Piyano parmaklamayı iki notalık akorlara genişlet: state'in t^F'ye çıktığını ve sürenin $O(n \cdot t^{2F})$ olduğunu göster.

33.15 Sonraki Ders İçin Hazırlık

Sonraki: Ders 27 (L18) — Dinamik Programlama 4 (pseudopolinom, DP FİNALİ)

Ders 27 (L18): Dinamik Programlama 4 (pseudopolinom). Erik Demaine ile, DP serisinin **finali**: **integer alt problemler** ve **pseudopolinom zaman**. Rod cutting ve subset sum örnekleriyle, $n \cdot T$ gibi sürelerin neden "polinom değil ama yeterince iyi" olduğunu görür, tüm DP tekniklerini diagonal bir bakışla toparlarız. DP ünitesi Ders 24-27 (L16-L18) boyunca sürer ve **Ders 30 = Quiz 3 Gözden Geçirme** ile özetlenir.

Ders 27 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 1 (Floyd-Warshall) ve 3 (parantezleme) çöz.
- Floyd-Warshall, parantezleme ve parmaklamanın SRTBOT'unu ezberden yaz.
- Ana cümleyi tekrar oku: *"Küçük state'i alt problem koordinatına ekle; geçişleri kaba kuvvetle tara."*

33.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Subproblem expansion	Alt probleme kısıt/koordinat ekle; state hatırla	Böl. 1
Bellman-Ford DP	δ_k (en fazla k kenar); çevrimli → çevrimsiz; $O(V \cdot E)$	Böl. 2
Floyd-Warshall	$d(u,v,k) = $ ilk k düğüm; min 2 terim; $O(V^3)$	Böl. 3-4
FW vs Johnson	dense → FW (basit); sparse → Johnson	Böl. 5
Parantezleme	Kökü (son işlem) tahmin et → substring	Böl. 6
Min/max genişletme	Negatif çarpım için hem min hem max sakla	Böl. 7-8
Parmaklama	$x(i, f)$; state = parmak; $O(n \cdot F^2)$	Böl. 9
Genel ilke	Küçük state → koordinat; geçişleri tara	Böl. 10

33.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu dersin üç tekniği — vertex-prefix APSP, kök tahmini + min/max, state genişletmesi — ML, derleyici ve sistem mühendisliğindeki çok sayıda araca bağlanır; köprülerin özeti:

1. **Floyd-Warshall** → APSP mesafe matrisi, **transitif kapanış** (reachability), ağ analizi; ağırlıkları boole'e indirgersen aynı üçlü döngü erişilebilirliği verir.
2. **Parantezleme** → derleyici ifade optimizasyonu, **matris-zincir çarpımı** (matrix chain multiplication — hangi sırada çarpınca en az skaler çarpma?), sözdizimi ağacı kurma.
3. **Min/max genişletme** → **aralık aritmetiği**; optimizasyonda alt sınır + üst sınır birlikte; soyut-yorumlama (abstract interpretation) ve sağlamlık (robustness) analizleri.
4. **Piyano/gitar parmaklama** → **MIDI/müzik teknolojisi**, dizilim hizalama, **durum-makinesi DP'leri** (her geçişte küçük bir state taşı).

5. **State = koordinat** → durum-augmentasyonu; **OMSCS CS 6515'te DP tasarımının kalbi** — "bariz alt problem yetmiyorsa hangi koordinatı eklerim?" refleksi (state = koordinat).
6. **DP = Bellman icadı** → optimizasyon tarihi; "programming" = optimizasyon (LP/IP ile aynı kök); "dynamic" = aşama-aşama yerel kaba kuvvet.

! Tek bir şey alıp gideceksen

Subproblem expansion, DP'nin en güçlü silahıdır: alt probleme küçük bir koordinat ekleyerek geçmişi/state'i hatırlarsın. Floyd-Warshall "yalnız ilk k düğümü kullan" der ve E terimini V'ye çevirip $O(V^3)$ verir. Parantezleme "son işlem (kök) hangisi?" diye sorar ve negatifler için hem min hem max saklar. Parmaklama "hangi parmakla başladım" durumunu taşır. Tek formül: küçük state'i alt problem sayısıyla çarp, geçişleri yerel kaba kuvvetle tara — neredeyse her problemin her yönünü yakalarsın.

34 Dinamik Programlama 4: Pseudopolinom, Subset Sum

DP finali — tamsayı alt problemler, rod cutting, OR-recurrence ile subset sum, pseudopolinom hiyerarşisi ve dört dersin toparlanması

i Oturum bilgisi

- **Demaine'in videosu:** YouTube — Lecture 18: Dynamic Programming, Part 4 (≈64 dk)
- **OCW sayfası:** MIT 6.006 Lecture 18: Dynamic Programming, Part 4
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 27 (L18)
- **Hoca:** Erik Demaine (dinamik programlama; **DP serisinin FİNALİ 4/4**)
- **Okuma süresi:** ≈27 dk

  Bu, DP ünitesinin **dördüncü ve son dersidir**. Ders 26 (L17) subproblem expansion'ı (Floyd-Warshall, parantezleme, parmaklama) derinleştirdi; bu ders **tamsayı (integer) alt problemleri** ele alır: Fibonacci'deki gibi bir tamsayı girdinin daha küçük sürümlerine bakmak. İki örnek — **rod cutting** ($O(L^2)$, gerçek polinom) ve **subset sum** ($O(n \cdot T)$, pseudopolinom) — ile yeni bir kavrama varırız: **pseudopolinom zaman**. Subset sum ayrıca bir **karar problemi**dir: recurrence'da min/max yerine **OR** kullanılır, "evet" certificate ile kolay kanıtlanır, "hayır" zordur — sonraki dersin (P/NP) habercisi. Ders dört DP dersinin **diagonal** (teknik-bazlı) toparlanmasıyla kapanır.

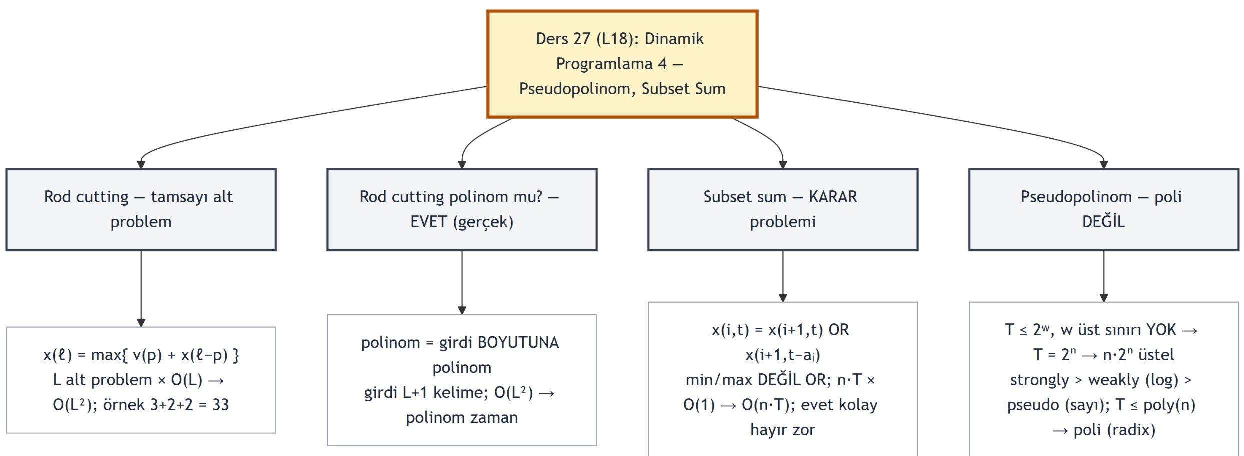
34.1 Bu Derste Ne Var?

DP serisinin **finali** (4/4, Erik Demaine). Odak: **tamsayı (integer) alt problemler** — Fibonacci'deki gibi, bir tamsayı girdinin daha küçük sürümlerine bakmak. Bu, yeni bir kavrama götürür: **pseudopolinom zaman**.

> *"pseudopolynomial is a pretty good running time."* — Demaine, 0:40

İki örnek (rod cutting, subset sum) + tüm DP'lerin **diagonal** (teknik-bazlı) toparlanması. Üç ana fikir:

1. **Tamsayı alt problem** — integer girdiyi $0 \dots n$ arası daha küçük tamsayılara böl (rod cutting, subset sum).
2. **Karar problemi** — "evet/hayır" çıktısı; recurrence'da min/max yerine **OR**.
3. **Pseudopolinom** — $n \cdot T$ gibi süre: girdi *boyutuna* değil girdi *sayılarına* polinom; "polinom değil ama yeterince iyi".



Şekil 34.1: Ders 27'nin (L18) kavram haritası: kök = Dinamik Programlama 4 (Demaine) — DP serisinin FİNALİ 4/4; tek tema TAMSAYI ALT PROBLEM, yani integer girdiyi 0 ile n arası daha küçük tamsayılara bölmek (Fibonacci'nin genellemesi). Dört dal — (1) Rod cutting tamsayı alt problem: L uzunluk çubuğu tamsayı parçalara böl, ilk parçanın boyu p tahmin et, x(ell) eşittir max p için v(p) artı x(ell eksi p), L alt problem çarpı O(L) seçim eşittir O(L kare); örnek L eşittir 7 değerler bir on onuc onsekiz yirmi otuzbir otuziki optimal uc arti iki arti iki eşittir onuc arti on arti on eşittir otuzuc. (2) Rod cutting polinom mu evet GERÇEK polinom: polinom zaman demek sürenin girdi BOYUTUNA polinom olması, girdi boyutu kelime cinsinden, rod girdisi L arti bir kelime, O(L kare) L arti bire polinom yani polinom zaman. (3) Subset sum KARAR problemi: n tamsayılık coklu küme A arti hedef T, herhangi alt küme T'ye toplanır mı, cıktı tek bit evet hayır, recurrence x(i,t) eşittir x(i arti bir, t) OR x(i arti bir, t eksi a-i) yani min max DEĞİL OR cünkü herhangi bir secenek evet verirse evet, n çarpı T alt problem çarpı O(bir) eşittir O(n çarpı T); evet'i kanıtlamak KOLAY certificate göster, hayır'ı kanıtlamak ZOR kısa yol yok NP habercisi. (4) Pseudopolinom poli DEĞİL: O(n çarpı T) girdi boyutu n arti bir kelime ama süre hem n hem T'ye bağlı, T bir kelimeye sığar w bit yani T küçük eşit iki üzeri w, w'nin üst sınırı YOK yani T eşittir iki üzeri n olabilir yani n çarpı T eşittir n çarpı iki üzeri n ÜSTEL; hiyerarşi strongly polinom sayıdan bağımsız büyük weakly polinom sayının LOG'una radix sort büyük pseudo polinom sayının KENDİSİNE subset sum; özel durum T küçük eşit poly(n) ise pseudo gerçek polinoma iner bu radix sort doğrusal koşulu. Birleştirici tema: dört DP dersi teknikleri kademeli tanıttı basit alt problem sonra coklu girdi ve kısıt sonra subproblem expansion sonra tamsayı alt problem ve pseudopolinom, DP ne bilsem işim biterdi sorusunu yerel kaba kuvvetle çözen güçlü bir tasarım çatısıdır.

Builder Notu — subset sum = knapsack / bütçe-dağıtımının temeli

Subset sum, **sırt çantası (knapsack)**, bütçe-dağıtım ve kaynak-tahsis problemlerinin çekirdeğidir: "verilen bir kapasite/hedefe hangi öğe alt kümesi sığar?" sorusu her yerde karşına çıkar. Aynı $O(n \cdot T)$ DP tablosu, hedef yerine kapasite koyduğunda 0/1 knapsack'e dönüşür. Ama dikkat: bu süre **pseudopolinom**dur — T (veya kapasite) büyükse patlar.

- **İleriye → knapsack/bütçe:** subset sum, knapsack ve bütçe-dağıtımının temelidir; çok boyutlu DP tablolarında kapasite/bütçe bir koordinattır.
- **İleriye → NP-tamlık:** subset sum bir karar problemidir; "evet"i kanıtlamak kolay (certificate), "hayır"ı zor — sonraki dersin (P/NP) habercisi.
- **Geriye → radix sort (Ders 7, L5):** pseudopolinom'un "poli olma" koşulu ($T \le \text{poly}(n)$), radix sort'un doğrusal çalışma koşuluyla **aynıdır** (sayılar poli-sınırlı) — aynı kavram iki yerde.
- **İleriye → OMSCS CS 6515:** "sayılar küçükse hızlı" bilinci; pseudopolinom algoritmaların hangi durumda pratik olduğunu (T küçük) ayırt etmek graduate algoritmalarının temel refleksidir.

Tek cümle: *Tamsayı girdiyi küçük sürümlerine bölerek DP yaparız; rod cutting $O(L^2)$ gerçek polinom, ama subset sum $O(n \cdot T)$ yalnız "pseudopolinom"dur ($T = 2^n$ olabilir) — yine de sayılar küçükse mükemmel.*

34.2 1. DP 4/4: Tamsayı Alt Problemler

İlk üç DP dersi sequence (prefix/suffix/substring), çoklu girdi (çarpım) ve subproblem expansion'ı kurdu. Bu ders, **tamsayı girdi** alt problemini derinleştirir: Fibonacci'de n verildi, $0 \dots n$ için çözdük. Aynı teknik rod cutting ve subset sum'da. SRTBOT aynı kalır; yeni soru: "bu süre **polinom** mu?"

34.3 2. Rod Cutting: Tamsayı Alt Problem

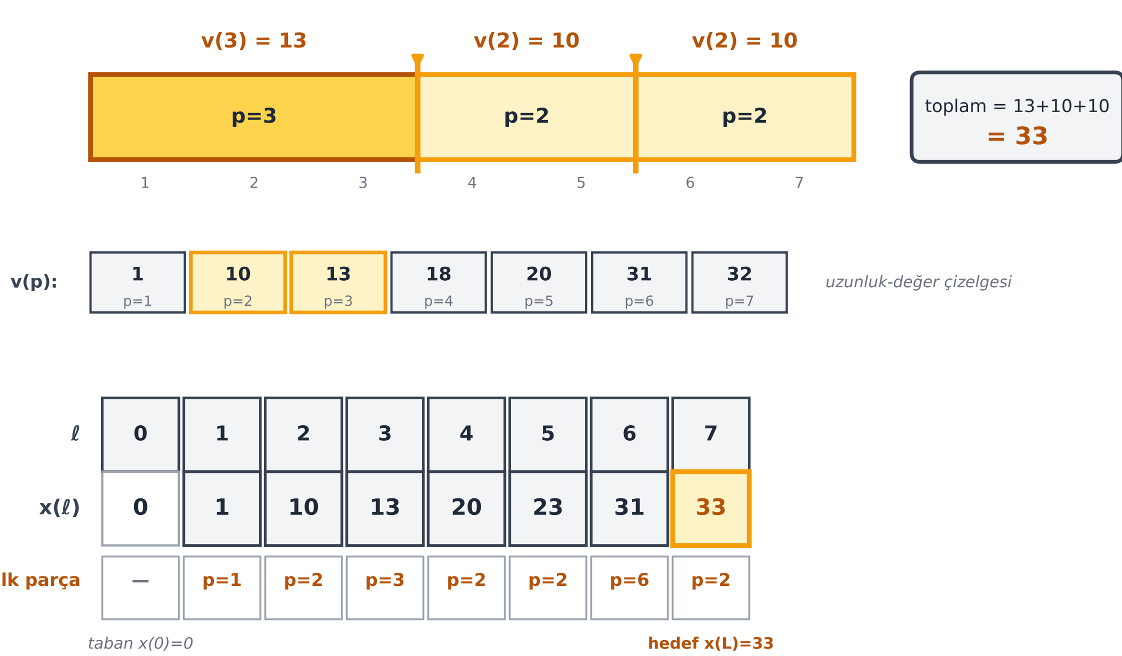
Problem. L uzunluğunda çubuk; $v(\ell) = \ell$ uzunluğunda parçanın satış değeri. Çubuğu tamsayı parçalara bölüp **toplam değeri maksimize** et. Örnek: $L = 7$, değerler $[1, 10, 13, 18, 20, 31, 32]$ → optimal $3+2+2 = 13+10+10 = 33$.

Çalışılan Örnek. S: $x(\ell) = \ell$ uzunluk çubuğunun maksimum-değer bölünmesi, $\ell = 0 \dots L$. **R:** ilk kesilecek parçanın boyu p'yi tahmin et:

$$x(\ell) = \max\{\, v(p) + x(\ell - p) : p = 1 \dots \ell \,\}$$

T: ℓ artan. **B:** $x(0) = 0$. **O:** $x(L)$. **Süre:** L alt problem $\times O(L)$ (p seçimi) $= O(L^2)$. (DAG-en-uzun-yol olarak da görülebilir: değerleri negatifleyip en kısa yol.)

Şekil 34.2 bu örneği motor üzerinde somutlaştırır: optimal bölünme $3 + 2 + 2$ (parça değerleri $v(3) = 13$, iki kez $v(2) = 10$), toplam 33 — `rod_plan` ve üstel `brute_rod` birebir aynı sonucu verir. Alt tablo $x(\ell)$'ı $\ell = 0 \dots 7$ için ve her hücrenin kazanan ilk parçasını ($\text{first}(\ell)$) gösterir; L alt problem $\times O(L)$ rozeti $O(L^2)$'nin **gerçek polinom** olduğunu (girdi $L + 1$ kelime) vurgular.



Şekil 34.2: Rod cutting: tamsayı alt problem DP (Demaine L18 §2-3 İMZA, 19:22). ÜST panel: 7-birim çubuk + optimal 3+2+2 kesim (amber ayraçlar), her parça v(p) değeriyle etiketli (v(3)=13, v(2)=10, v(2)=10), sağ kutuda toplam = 13+10+10 = 33; altta uzunluk-değer çizelgesi v(1..7) (kullanılan p=2,3 amber vurgulu). ALT panel: x(ℓ) DP tablosu ℓ=0..7 + kazanan ilk parça first(ℓ) satırı; taban x(0)=0 soluk, hedef x(7)=33 amber; recurrence kutusu x(ℓ) = max{ v(p) + x(ℓ−p) : p=1..ℓ }; O(L^2) rozeti 'L alt problem × O(L) = O(L^2), GERÇEK polinom (girdi L+1 kelime · Demaine 19:22)'. Veri MOTORDAN (assert): build_rod_example == (7, [0,1,10,13,18,20,31,32]); rod_plan == ([3,2,2], 33); brute_rod == 33; x == [0,1,10,13,20,23,31,33]; first == [0,1,2,3,2,2,6,2]; parça değerleri [13,10,10] toplam 33.

34.4 3. Rod Cutting Polinom mu?

Evet — **gerçek polinom** (strongly polynomial). "Polinom zaman" = sürenin **girdi boyutuna** polinom olması. Girdi boyutu **kelime (word)** cinsinden ölçülür.

> *"polynomial time means that the running time is polynomial in the size of the input."* — Demaine, 19:22

Rod cutting girdisi: bir sayı L + L sayılık değer dizisi $= L+1$ kelime. $O(L^2)$, $L+1$'e polinomdur $\rightarrow$ polinom zaman. ✓ (Şekil 34.2'in $O(L^2)$ rozeti bu sonucu birebir taşır: L alt problem $\times$ $O(L)$ seçim.)

34.5 4. Subset Sum: Karar Problemi

Problem. n tamsayılık çoklu-küme (multiset) A + hedef T. **Herhangi bir alt küme T'ye toplanır mı?** Bu bir **karar problemi** — çıktı tek bit: evet/hayır. Örnek: $A = \{2,5,7,8,9\}$, $T = 21 \rightarrow$ evet $(5+7+9)$; $T = 25 \rightarrow$ hayır.

> *"this is what we call a decision problem… we're just interested in a yes or no answer."* — Demaine, 27:10

İlginç: "evet"i kanıtlamak kolay — alt kümeyi göster; "hayır"ı kanıtlamak için bilinen kısa bir yol yok. Sonraki dersin habercisi.

> *"there's no succinct way as far as we know to prove to someone that the answer is no."* — Demaine, 26:49

Şekil 34.3 bu asimetriyi motor üzerinde gösterir: $T = 21$ için `subset_sum_certificate` certificate $\{5,7,9\}$ döndürür $(5+7+9 = 21)$; bir doğrulayıcı bunu **polinom zamanda** kontrol eder (`verify_subset_certificate` $\rightarrow$ True). Yanlış bir certificate $\{5,7,8\}$ $(= 20 \neq 21)$ de hızla reddedilir. Ama $T = 25$ için certificate `None`'dır — hiçbir alt küme 25 vermez ve bunu kısa yoldan kanıtlamanın bilinen yolu yok, tek kesinlik $2^5 = 32$ alt kümenin **hepsini** taramak. Bu asimetri, Ders 28'in (L19) **NP** sınıfının habercisidir.

34.6 5. Subset Sum Recurrence: OR

Çalışılan Örnek. S: $x(i,t) = A[i:]$ sonekinin herhangi bir alt kümesi t'ye toplanır mı? ($0 \leq i \leq n$, $0 \leq t \leq T$). a_i'yi kümeye **koy** ya da **koyma** — iki seçenek:

$$x(i,t) = x(i+1,t) \ \textbf{OR} \ \left(x(i+1,t-a_i) \ : \ a_i \leq t\right)$$

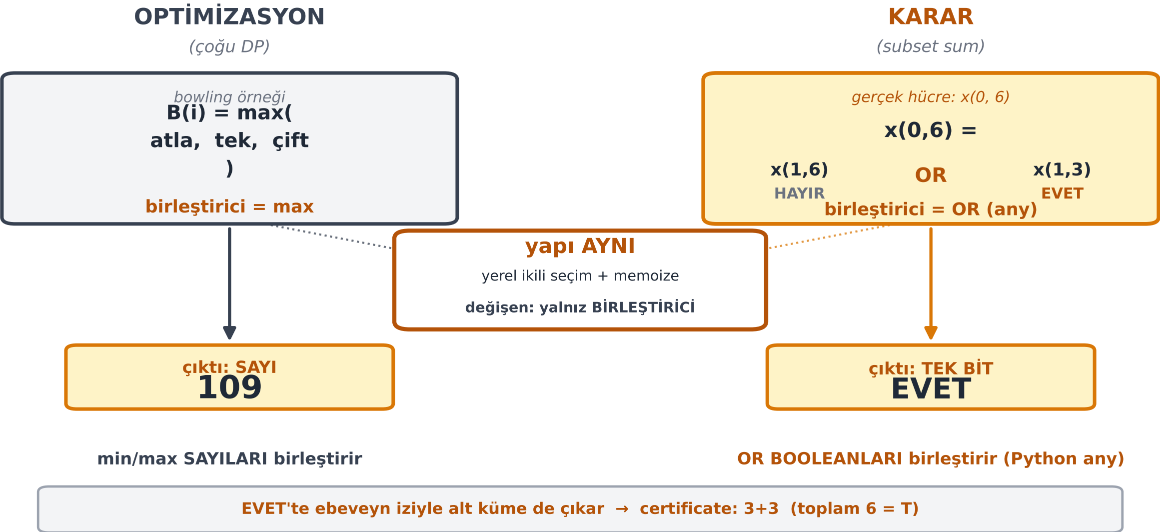
(İkinci terim yalnız $a_i \leq t$ ise geçerlidir.)

İlk: a_i dışarıda. İkinci: a_i içeride $\rightarrow$ kalan $t - a_i$'ye toplanmalı; $a_i \leq t$ koşulu negatif t'yi önler. Optimizasyon değil $\rightarrow$ min/max değil, **OR**. **T:** i azalan. **B:** $x(n, 0) = $ evet; $x(n, t ≠0) = $ hayır. **O:** $x(0,T)$. 2^n alt kümeyi, yerel ikili seçimlerle ve memoization ile tararız (alt problemler "toplamı t olan" tüm alt kümeleri tek değere çöker).

Şekil 34.3: Karar problemi asimetrisi: «EVET»i doğrulamak kolay, «HAYIR»ı zor (Demaine L18 §4, 27:10 + 26:49; NP habercisi → Ders 28/L19). SOL panel 'EVET kanıtı KOLAY': A = {2,5,7,8,9} 5 madalyon, certificate {5,7,9} amber seçili; doğrulayıcı kutusu '5+7+9 = 21 ✓ → polinom zamanda KONTROL (Demaine 27:10)'; yanlış-cert {5,7,8} → 20 ≠ 21 kırmızı REDDET (doğrulayıcı yine hızlı yakalar). SAĞ panel 'HAYIR kanıtı YOK': T=25 büyük kırmızı soru kutusu 'hiçbir alt küme 25 vermez, kısa ispat yok, certificate = None (Demaine 26:49)'; $2^5 = 32$ alt kümenin mini-ızgarası (hepsi ×); alt rozet → Ders 28 (L19): NP — EVET doğrulanabilir, HAYIR değil. Veri MOTORDAN (assert): build_subset_example == ([2,5,7,8,9], 21, 25); subset_sum(A,21)[0] is True, certificate == [5,7,9] (toplam 21); subset_sum(A,25)[0] is False, certificate is None; verify_subset_certificate(A,21,[5,7,9]) is True; verify_subset_certificate(A,21,[5,7,8]) is False (5+7+8=20); $2^5 = 32$.

Süre: $n \cdot T$ alt problem $\times O(1) = O(n \cdot T)$. ("Evet" durumunda ebeveyn işaretçileriyle alt küme de bulunur.)

Şekil 34.4 bu recurrence'ın özünü, bir optimizasyon DP'siyle (bowling) yan yana koyar: yapı **aynı** (yerel ikili seçim + memoize), değişen yalnız **birleştirici**dir — bowling `max` ile sayı (motor: $B(0) = 109$), subset sum `OR` ile tek bit ($x(0,6) = x(1,6)$ False OR $x(1,3)$ True = True) üretir.



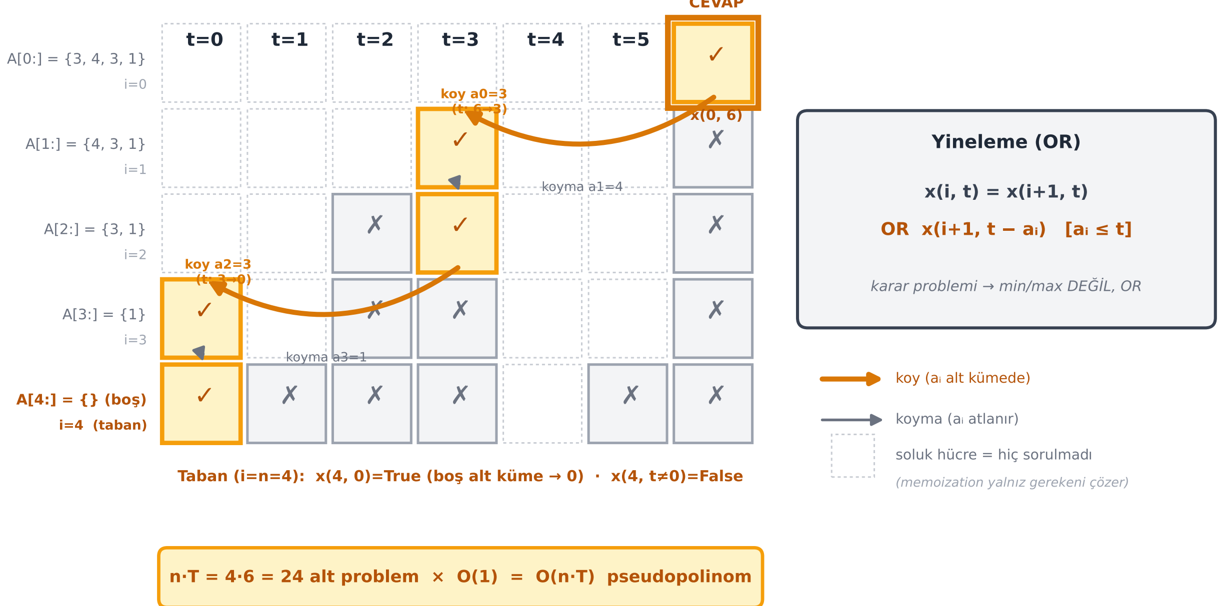
Şekil 34.4: Karar vs optimizasyon recurrence: subset sum OR-birleştirici tek bit, bowling max-birleştirici sayı (Demaine L18 §5). SOL sütun 'OPTİMİZASYON (çoğu DP)': bowling B(i) = max(atla, tek, çift); çıktı SAYI 109; min/max SAYILARI birleştirir. SAĞ sütun 'KARAR (subset sum)': gerçek hücre x(0,6) = x(1,6)=HAYIR OR x(1,3)=EVET; çıktı TEK BİT EVET; OR BOOLEANLARI birleştirir (Python any). ORTA köprü rozeti 'yapı AYNI: yerel ikili seçim + memoize; değişen yalnız BİRLEŞTİRİCİ'. Alt not: EVET'te ebeveyn iziyle alt küme de çıkar → certificate [3,3] (toplam 6 = T). Veri MOTORDAN (assert): build_subset_small_example == ([3,4,3,1], 6); subset_sum(A,6)[0] is True; x(0,6) = x(1,6)=False OR x(1,3)=True = True; a_0=3, t−a_0=3; certificate == [3,3] toplam 6; build_bowling_example == [1,9,9,2,–5,–5]; bowling_bottom_up B[0] == 109.

Şekil 34.5 aynı DP'yi `build_subset_small_example` ($A = \{3, 4, 3, 1\}$, $T = 6$) üzerinde tam tabloyla gösterir: satırlar $i = 0 \ldots 4$, sütunlar $t = 0 \ldots 6$. Kritik detay — **yalnız memoization'ın gerçekten sorduğu hücreler doludur** (motor `memo`'sundan): 16 hücre, $n \cdot T = 4 \cdot 6 = 24$ olası alt problemin ve 6×22 üst-sınırın çok altında — memoization yalnız gerekeni çözer (pseudopolinom tanığı). Certificate yolu $[3, 3]$ kalın amber "koy" / soluk "koyma" oklarıyla izlenir.

34.7 6. Subset Sum Polinom Değil: Pseudopolinom

$O(n \cdot T)$ **polinom değildir**. Girdi boyutu $= n + 1$ kelime, ama süre hem n'e hem T'ye bağlı. T bir kelimeye sığar (w bit) $\rightarrow T \leq 2^w$. Word-RAM'de $w \geq \log n$ garantilidir ama w**'nin üst sınırı yoktur** — $w = n$ bile makul (n sayı, her biri n-bit). O zaman $T = 2^n \rightarrow n \cdot T = n \cdot 2^n$ **üstel**.

Yine de "yeterince iyi": **pseudopolinom**.



Şekil 34.5: Subset Sum: x(i,t) OR-recurrence memoization tablosu (Demaine L18 §5 İMZA). A = {3,4,3,1}, T = 6. Satırlar i=0..4 (taban i=4 EN ALTTA, satır etiketi A[i:] soneki), sütunlar t=0..6. SADECE memoization'ın gerçekten sorduğu hücreler dolu (motor memo'sundan, 16 hücre); kalanlar soluk kesik 'hiç sorulmadı' — memoization yalnız gerekeni çözer (pseudopolinom tanığı). True = amber ✓, False = slate ×. Cevap x(0,6) büyük amber çerçeve = EVET. Certificate yolu [3,3]: koy adımları kalın amber ok (koy a_0=3 t:6→3, koy a_2=3 t:3→0), koyma adımları soluk slate ok (koyma a_1=4, koyma a_3=1). Recurrence kutusu: x(i,t) = x(i+1,t) OR x(i+1,t−a_i) [a_i≤t] — karar problemi → min/max DEĞİL OR. Alt rozet: n·T = 4·6 = 24 alt problem × O(1) = O(n·T) pseudopolinom. Veri MOTORDAN (assert): build_subset_small_example == ([3,4,3,1], 6); subset_sum True, memo[(0,6)] True, memo[(4,0)] True; certificate == [3,3] toplam 6; yol [(0,6)PUT3,(1,3)skip4,(2,3)PUT3,(3,0)skip1]; memo hücre sayısı 16.

34.8 7. Pseudopolinom Tanımı ve Hiyerarşi

Pseudopolinom zaman: girdi *boyutuna* **ve** girdi *sayılarına* polinom (sabit dereceli). $O(n \cdot T)$ böyledir: $n T = $ boyut $\times T$, sabit derece. Pseudopoli ama poli değil.

Önemli özel durum: girdi sayıları boyutun **polinomu** ise (örn. $T \le \text{poly}(n)$) $\to$ pseudopoli **polinoma** iner. Bu, **radix sort'un doğrusal çalışma koşuluyla aynıdır**.

> *"this is the condition when radix sort runs in linear time."* — Demaine, 50:50

Hiyerarşi (iyiden kötüye): **strongly polynomial** (girdi sayılarından bağımsız) $\to$ **weakly polynomial** (sayıların **log**'una polinom — radix sort) $\to$ **pseudopolynomial** (sayıların **kendisine** polinom — subset sum). "log of an exponential is polynomial" olduğundan weakly poli neredeyse poli kadar iyidir.

> *"log of an exponential is polynomial."* — Demaine, 53:41

Şekil 34.6 bu üç kademeyi hem merdiven şemasıyla hem de büyüme grafiğiyle gösterir: $n = 20, T = 2^{20}$ için strongly ($n^2 = 400$), weakly ($n \cdot \log_2 T = 400$) ve pseudo ($n \cdot T \approx 2{,}1 \times 10^7$) — pseudo, weakly/strongly'den ~ 52 bin kat büyük. Sağ panel w (bit) arttıkça pseudo'nun $n \cdot 2^w$ üstel patlamasını ($T \le 2^w$, w'nin üst sınırı yok $\to T = 2^n$), strongly'nin sabit ve weakly'nin doğrusal kaldığını çizer; radix koşulu ($T \le \text{poly}(n)$) rozetiyle.

34.9 8. DP Karakterizasyonu: Alt Problem Tipleri

Dört dersteki tüm DP'ler, **alt problem tipine** göre:

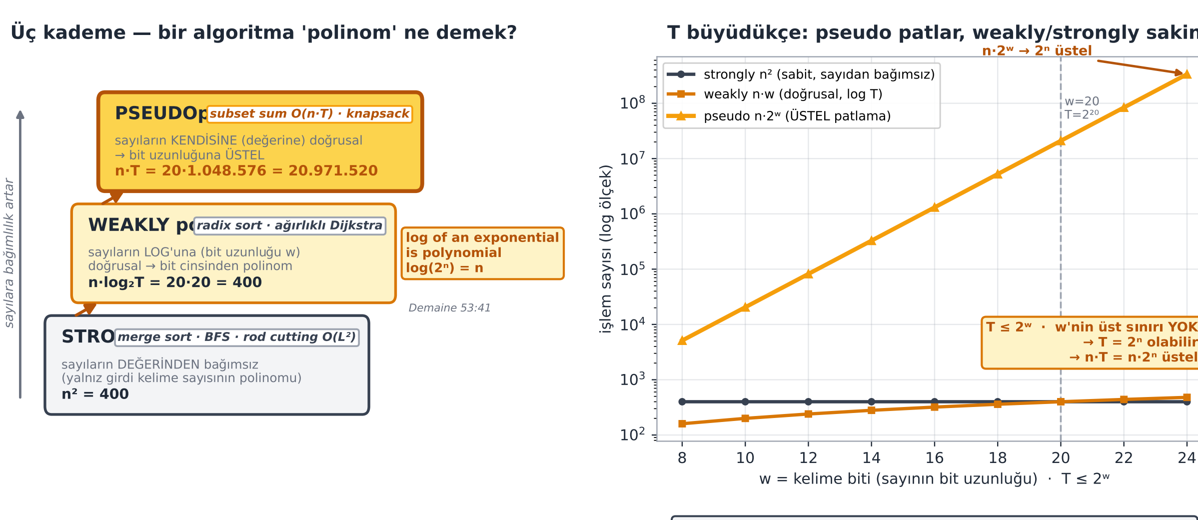
- **Prefix/Suffix:** bowling, LCS (iki sonek), LIS.
- **Substring:** değişen para oyunu (iki uçtan), parantezleme (ortadan).
- **Tamsayı:** Fibonacci, rod cutting, subset sum (+ Floyd-Warshall, vertex-prefix olarak).
- **Vertex:** DAG SP, Bellman-Ford, Floyd-Warshall (en kısa yol DP'leri).

Pseudopoli olanlar: **rod cutting** (aslında poli), **subset sum** (pseudopoli) — tamsayı alt problemli olanlar.

34.10 9. DP Karakterizasyonu: Constraint / Branching / Combination

- **Constraint/expansion:** non-expansive (LIS: kısıt ekledi ama alt problem sayısı sabit); expansive (para oyunu $\times 2$, parmaklama $\times F$, Bellman-Ford $\times V$).
- **Branching (özyineleme-dışı iş):** sabit (Fibonacci, bowling, LCS, para oyunu, Floyd-Warshall, subset sum); derece-bazlı (DAG SP, Bellman-Ford $\to E$ terimi); doğrusal (LIS, parantezleme, rod cutting).
- **Combination:** tek-en-iyi (çoğu $\to$ DAG en kısa yol); çoklu-birleştirme (Fibonacci toplar, Floyd-Warshall yol birleştirir, parantezleme iki parçayı çarpar/toplar).
- **Original problem:** tek alt problem (çoğu) vs çoklu (DAG SP, LIS, Bellman-Ford, Floyd-Warshall'da hepsinin min/max'ı).

Şekil 34.7 bu iki bölümün (§8 alt problem + §9 constraint/branching/combination) **diagonal** toparlanmasını tek tabloda dizer: 11 DP örneği satır, dört sınıflama ekseni sütun. Pseudopoli satırlar (rod cutting, subset sum) amber vurgulu; alt şerit dört dersin kademeli tanıtımını (DP1 = Ders 23 $\to$ DP4 = Ders 27) gösterir.



Şekil 34.6: Polinom hiyerarşisi: sayılar nasıl ölçülür? — strongly · weakly · pseudo (Demaine L18 §7 İMZA, 53:41 + 50:50). SOL panel üç-kademe merdiven (sayılara bağımlılık artar ↑): (1) STRONGLY polynomial — sayıların DEĞERİNDEN bağımsız, yalnız girdi kelime sayısının polinomu; merge sort · BFS · rod cutting $O(L^2)$; $n^2 = 400$. (2) WEAKLY polynomial — sayıların LOG'una (bit uzunluğu w) doğrusal; radix sort · ağırlıklı Dijkstra; $n{\cdot}\log_2 T = 20{\cdot}20 = 400$; yan not 'log of an exponential is polynomial, $\log(2^n)=n$' (Demaine 53:41). (3) PSEUDOpolynomial — sayıların KENDİSİNE doğrusal → bit uzunluğuna ÜSTEL; subset sum $O(n{\cdot}T)$ · knapsack; $n{\cdot}T = 20{\cdot}1.048.576 = 20.971.520$. SAĞ panel T büyüdükçe süre (log-y), x = w (bit) 8..24: strongly n^2 sabit, weakly $n{\cdot}w$ doğrusal, pseudo $n{\cdot}2^w$ ÜSTEL patlama (amber); '$T \leq 2^w$, w üst sınırı YOK → $T = 2^n$ → $n{\cdot}2^n$ üstel' kutusu + alt rozet '$T \leq poly(n)$ ise pseudo → gerçek polinom (radix koşulu, Demaine 50:50)'. Veri FORMÜLDEN (assert): n=20, $T=2^{20}$=1.048.576; strongly=400, weakly=400, pseudo=20.971.520; pseudo/weakly=52428.8; her w için pseudo=$n{\cdot}2^w$ üstel.

DP taksonomisi: dört dersin diagonal toparlanması — her örnek alt problem × constraint × branching × combination ekseninde

DP örneği	ders	alt problem tipi *prefix/suffix · substring · tamsayı · vertex*	constraint *non-exp · expansive ×k*	branching *sabit · derece · doğrusal*	combination *tek-en-iyi · çoklu*
Fibonacci	DP1	tamsayı	non-exp	sabit	çoklu-birleştirme
bowling	DP1	prefix/suffix	non-exp	sabit	tek-en-iyi
LCS	DP2	prefix/suffix	non-exp	sabit	tek-en-iyi
LIS	DP2	prefix/suffix	non-exp	doğrusal	tek-en-iyi
para oyunu	DP2	substring	expansive ×2	sabit	tek-en-iyi
parantezleme	DP3	substring	expansive ×F	doğrusal	çoklu-birleştirme
Floyd-Warshall	DP3	vertex	expansive ×V	sabit	çoklu-birleştirme
DAG SP	DP3	vertex	non-exp	derece (E)	tek-en-iyi
Bellman-Ford	DP3	vertex	expansive ×V	derece (E)	tek-en-iyi
rod cutting *GERÇEK poli O(L²)*	DP4	tamsayı	non-exp	doğrusal	tek-en-iyi
subset sum *PSEUDO poli O(n·T)*	DP4	tamsayı	non-exp	sabit	tek-en-iyi

DP1 · Ders 23 temel alt problem (Fibonacci, bowling) → DP2 · Ders 24 çoklu girdi + kısıt (LCS, LIS, para oyunu) → DP3 · Ders 26 subproblem expansion (parantezleme, Floyd-W.) → DP4 · Ders 27 tamsayı + pseudopolinom (rod cutting, subset sum)

dört DP dersi teknikleri KADEMELİ tanıttı — her ders bir öncekine yeni bir araç ekledi (Demaine 1:03:02)

Şekil 34.7: DP taksonomisi: dört dersin diagonal toparlanması — her örnek alt problem × constraint × branching × combination ekseninde (Demaine L18 §8-9 İMZA, 1:03:02). 11 DP örneği satır; 4 sınıflama sütunu. Alt problem tipi renk kodlu (prefix/suffix · substring · tamsayı amber · vertex yeşil); constraint (non-exp vs expansive ×k amber); branching (sabit · derece · doğrusal); combination (tek-en-iyi · çoklu amber). Pseudopoli satırlar AMBER vurgu: rod cutting (GERÇEK poli $O(L^2)$, girdi L+1 kelime) ve subset sum (PSEUDO poli $O(n{\cdot}T)$, $T=2^n$ olabilir) — ikisi de tamsayı. Ders rozetleri DP1=Ders 23, DP2=Ders 24, DP3=Ders 26, DP4=Ders 27 (araya Ders 25 = PS8). Alt şerit: dört ders KADEMELİ tanıttı — DP1 temel → DP2 çoklu+kısıt → DP3 expansion → DP4 tamsayı+pseudo (Demaine 1:03:02). Veri L18 §8-9 BİREBİR (assert): 11 örnek; pseudopoli = {rod cutting, subset sum}; expansive = {para oyunu, parantezleme, Floyd-Warshall, Bellman-Ford}; çoklu-birleştirme = {Fibonacci, Floyd-Warshall, parantezleme}; derece = {DAG SP, Bellman-Ford}.

34.11 10. Dört Dersin Özeti

Dört DP dersi, teknikleri **kademeli** tanıttı: basit alt problemler (DP1) $\rightarrow$ çoklu girdi + kısıt (DP2) $\rightarrow$ subproblem expansion (DP3) $\rightarrow$ tamsayı alt problem + pseudopolinom (DP4). Her ders bir önceki üzerine yeni bir araç ekledi — basit branching'ten karmaşık birleştirmeye.

> *"these four dp lectures were all about showing you these main techniques of dynamic programming."* — Demaine, 1:03:02

Özü: alt problemleri (sequence/integer/vertex) tanımla, kısıt/expansion ile state hatırla, "ne bilsem işim biterdi?" sorusunu yerel kaba kuvvetle dene, memoize et — DP çok güçlü bir tasarım çatısıdır. (Şekil 34.7 bu kademeli yolculuğu DP1 $\rightarrow$ DP4 alt şeridinde özetler.)

34.12 Bu Dersin Özeti

1. **Tamsayı alt problem**: integer girdiyi 0 ... n'ye böl (Fibonacci, rod cutting, subset sum).
2. **Rod cutting**: $x(\ell) = \max\{v(p) + x(\ell - p)\}$; $O(L^2)$, gerçek polinom (girdi $L + 1$ kelime).
3. **Subset sum**: karar problemi; $x(i, t) = \text{OR}(\text{koyma}, \text{koy})$; $O(n \cdot T)$.
4. **Karar problemi**: min/max yerine OR; "evet" kolay kanıt, "hayır" zor.
5. **Pseudopolinom**: girdi boyutu $\times$ sayılara polinom; $T = 2^n$ olabilir $\rightarrow$ poli değil.
6. **Hiyerarşi**: strongly $>$ weakly (log sayı, radix sort) $>$ pseudopoly (sayı, subset sum).
7. **DP karakterizasyonu**: alt problem (prefix/substring/integer/vertex) $\times$ constraint $\times$ branching $\times$ combination.

> **!** Tek Bir Cümle
>
> DP'nin son aracı tamsayı alt problemdir: integer girdiyi küçük sürümlerine böleriz; rod cutting $O(L^2)$ gerçek polinom ama subset sum $O(n \cdot T)$ yalnız pseudopolinomdur ($T = 2^n$ olabilir) — yine de sayılar küçükse mükemmel, tıpkı radix sort gibi.

34.13 Kontrol Soruları

> **i** Soru 1: Subset sum bir «karar problemi» olduğu için recurrence'ı diğer DP'lerden nasıl farklı? Neden OR?
>
> **Cevap:** Çoğu DP **optimizasyon** problemidir (min/max) — recurrence'ın dışına min veya max koyarız. Subset sum ise **karar problemi**: çıktı tek bit (evet/hayır), bir değer değil. a_i'yi koyma ($x(i + 1, t)$) ya da koy ($x(i + 1, t - a_i)$) seçeneklerinden **herhangi biri** "evet" verirse cevap "evet"tir $\rightarrow$ birleştirici **OR** (Python'da `any`). Min/max sayıları, OR ise booleanları birleştirir. (Yine de yapı aynı: alt problemleri tanımla, yerel ikili seçimi kaba kuvvetle dene, memoize et.)

ℹ Soru 2: O(L^2) rod cutting «polinom» ama O(n·T) subset sum «polinom değil» — fark nedir?

Cevap: Polinom zaman = sürenin **girdi boyutuna** (kelime sayısı) polinom olması. Rod cutting girdisi $L+1$ kelime; $O(L^2)$, L'ye polinom → **polinom**. Subset sum girdisi $n+1$ kelime; süre $O(n \cdot T)$ hem n'e hem T'ye bağlı. T bir kelimeye sığar (w bit) → $T \leq 2^w$; w'nin üst sınırı yoktur ($w = n$ bile olabilir) → $T = 2^n \to n \cdot T = n \cdot 2^n$ **üstel**. Yani T girdi boyutuyla sınırlı değil; süre girdi boyutuna polinom **değil**. (Fark: L hem girdi boyutu hem süre parametresi; T yalnız bir girdi *sayısı*, boyut değil.)

ℹ Soru 3: «Pseudopolinom» ne demek, ve ne zaman gerçek polinoma dönüşür?

Cevap: Pseudopolinom = sürenin girdi boyutuna **ve girdideki sayıların kendisine** polinom olması (sabit derece). $O(n \cdot T)$ böyledir: n (boyut) $\times T$ (sayı). Gerçek polinom değildir çünkü T üstel büyüyebilir. **Özel durum:** girdi sayıları boyutun polinomuysa ($T \leq \text{poly}(n)$), pseudopolinom **polinoma** iner. Bu, radix sort'un doğrusal çalıştığı koşulla **aynıdır** (sayılar poli-sınırlı). Hiyerarşi: strongly poly (sayıdan bağımsız) > weakly poly (sayının log'u, radix sort) > pseudopoly (sayının kendisi, subset sum). "Sayılar küçükse hızlı" — pratikte çoğu zaman yeterince iyi.

ℹ Soru 4: Dört DP dersi hangi teknikleri kademeli olarak tanıttı?

Cevap: DP1 (SRTBOT + memoization): temel çerçeve, basit alt problemler (Fibonacci, bowling), prefix/suffix/substring. **DP2:** çoklu girdi (alt problem çarpımı, LCS), alt problem kısıtı (LIS), oyunlarda min/max (para oyunu). **DP3:** subproblem expansion (Floyd-Warshall vertex-prefix, parantezleme min+max, parmaklama state). **DP4:** tamsayı alt problem + pseudopolinom (rod cutting, subset sum). Her ders bir öncekine yeni bir araç ekledi: basit branching'ten derece/doğrusal branching'e, tek-en-iyi birleştirmeden çoklu birleştirmeye. Sıra, DP'nin tüm ana tekniklerini metodik biçimde göstermek için seçildi.

34.14 Egzersizler

Egzersiz 1. Rod cutting'i $L = 7$, $[1, 10, 13, 18, 20, 31, 32]$ üzerinde $x(\ell)$ tablosuyla elle çöz; optimal bölünmeyi (33) bul.

Egzersiz 2. Subset sum'ı Python'da yaz ($x(i,t)$, OR recurrence); $A = \{3, 4, 3, 1\}$, $T = 6$ için tabloyu doldur ve ebeveyn izinden alt kümeyi çıkar.

Egzersiz 3. Subset sum'ın neden $O(n \cdot T)$ olduğunu ve $T = 2^n$ durumunda neden üstel olduğunu input-size argümanıyla açıkla.

Egzersiz 4. Counting sort ve direct-access-array'in de neden pseudopolinom olduğunu (u'ya bağımlılık) yaz; radix sort'un neden weakly polynomial olduğunu açıkla.

Egzersiz 5. Gördüğün her DP dersi örneğini (Fibonacci, bowling, LCS, LIS, para oyunu, parantezleme, Floyd-Warshall, rod cutting, subset sum) alt problem tipine (prefix/suffix/substring/integer/vertex) göre sınıfla.

34.15 Sonraki Ders İçin Hazırlık

⚠ Sonraki: Ders 28 (L19) — Hesaplama Karmaşıklığı (P, NP, indirgemeler)

Ders 28 (L19): Hesaplama Karmaşıklığı (P, NP, indirgemeler). Erik Demaine ile, "hangi problemler verimli çözülebilir?" sorusuna geçiyoruz: **P** (polinom-zaman çözülebilir), **NP** (çözümü polinom-zamanda doğrulanabilir), **NP-tamlık** ve **indirgeme**. Subset sum'ın "evet kolay, hayır zor" gözlemi tam buraya bağlanır. (L19, Ders 30 = Quiz 3 Gözden Geçirme kapsamında değil; tanımlarını final sınavında kullanırsın.)
Ders 28 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 2 (subset sum) ve 5 (DP sınıflama) çöz.
- Pseudopolinom hiyerarşisini (strongly/weakly/pseudo) ezberden anlat.
- Ana cümleyi tekrar oku: *"Sayılar küçükse pseudopolinom mükemmeldir; $T = 2^n$ olabileceğinden poli değildir."*

34.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
Tamsayı alt problem	Integer girdiyi $0 \ldots n$'ye böl (Fibonacci, rod, subset)	Böl. 1
Rod cutting	$x(\ell) = \max\{v(p) + x(\ell - p)\}$; $O(L^2)$	Böl. 2
Polinom zaman	Süre girdi boyutuna (kelime) polinom	Böl. 3
Karar problemi	Evet/hayır çıktı; recurrence'da OR	Böl. 4-5
Subset sum	$x(i, t) = \text{OR}(\text{koyma}, \text{koy})$; $O(n \cdot T)$	Böl. 5
Pseudopolinom	Boyut $\times$ sayılara polinom; $T = 2^n$ olabilir	Böl. 6-7
Hiyerarşi	strongly $>$ weakly (log) $>$ pseudo (sayı)	Böl. 7
DP karakterizasyonu	alt problem $\times$ constraint $\times$ branching $\times$ combination	Böl. 8-9

34.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu dersin iki örneği — tamsayı alt problemli rod cutting ve subset sum — ve pseudopolinom kavramı, sistem mühendisliği, kriptografi ve graduate algoritmalarındaki çok sayıda araca bağlanır; köprülerin özeti:

1. **Subset sum** $\to$ **knapsack**, bütçe-dağıtım, kaynak tahsisi; NP-tamlık (kriptografi — subset-sum tabanlı şifreleme tarihçesi).
2. **Karar problemi + sertifika** $\to$ **NP** tanımı; "evet kolay, hayır zor" $\to$ sonraki ders (P/NP). Certificate verifier (`verify_subset_certificate`) NP Tanım-2'nin ta kendisidir.
3. **Pseudopolinom** $\to$ "sayılar küçükse hızlı"; gerçek sistemde T sınırlıysa pratik — kapasite/bütçe küçük tutulduğunda DP tablosu patlamaz.
4. **Pseudopoli** $=$ **radix sort koşulu** $\to$ sayı-sınırı bilinci; pseudopolinom'un poli-olma koşulu ($T \leq \text{poly}(n)$), radix sort'un doğrusal koşuluyla **aynı** kavramdır — iki yerde **çift-kayıt** aynı fikrin.
5. **DP diagonal review** $\to$ tasarım deseni kütüphanesi; **OMSCS CS 6515**'in DP omurgası — alt problem $\times$ constraint $\times$ branching $\times$ combination ekseni graduate algoritmalarında tekrar tekrar karşına çıkar.
6. **Tamsayı alt problem** $\to$ çok boyutlu DP tabloları; bütçe/kapasite parametreleri bir koordinat olarak eklenir (state $=$ kapasite). **OMSCS pseudo-bilinç:** bir DP'nin pseudopolinom mu gerçek polinom mu olduğunu ayırt etmek — graduate seviyede "bu algoritma büyük girdide patlar mı?" refleksinin temelidir.

Tek bir şey alıp gideceksen

DP'nin son aracı **tamsayı alt problemdir** — integer girdiyi küçük sürümlerine böl. Ama dikkat: rod cutting $O(L^2)$ gerçek polinomken, subset sum $O(n \cdot T)$ yalnız **pseudopolinom**dur, çünkü T bir kelimeye sığsa da 2^n kadar büyük olabilir. Pseudopolinom "polinom değil ama yeterince iyi": sayılar küçükse (radix sort koşulu) gerçek polinoma iner. Karar problemlerinde min/max yerine **OR** kullanırsın. Dört DP dersi, basit alt problemlerden pseudopolinoma uzanan tüm tasarım tekniklerini kademeli öğretti — DP, "ne bilsem işim biterdi?" sorusunu yerel kaba kuvvetle çözen güçlü bir çatıdır.

35 Hesaplama Karmaşıklığı: P, NP, NP-Tamlık

Bütün bir alan tek derste — sınıf hiyerarşisi, halting, şanslı algoritma ve certificate, reduction ile zorluk kanıtı

ℹ Oturum bilgisi

- **Demaine'in videosu:** YouTube — Lecture 19: Complexity (≈59 dk)
- **OCW sayfası:** MIT 6.006 Lecture 19: Complexity
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 28 (L19)
- **Hoca:** Erik Demaine (hesaplama karmaşıklığı; bütün bir alan tek derste)
- **Okuma süresi:** ≈28 dk

  Bu ders **ödevlerde işlenmedi**; final'de yalnız **TANIMLAR** test edilir (Ders 31'in retrospektif notu). Önceki dört DP dersinden (Ders 23-27) sonra perspektif tersine döner: artık "nasıl iyi çözeriz?" değil, "**neden iyi çözemeyiz?**" (alt sınır) sorusu merkezdedir. Subset sum'ın (Ders 27) "evet kolay, hayır zor" gözlemi tam buraya, **NP**'ye bağlanır.

35.1 Bu Derste Ne Var?

Bütün bir alanı tek derste: **hesaplama karmaşıklığı (computational complexity)** (Erik Demaine). Algoritmalar "nasıl iyi çözeriz?" derken, karmaşıklık "**neden iyi çözemeyiz?**" (alt sınır) der. Buradaki ölçek **polinom vs üstel** (n vs $n \log n$ değil).

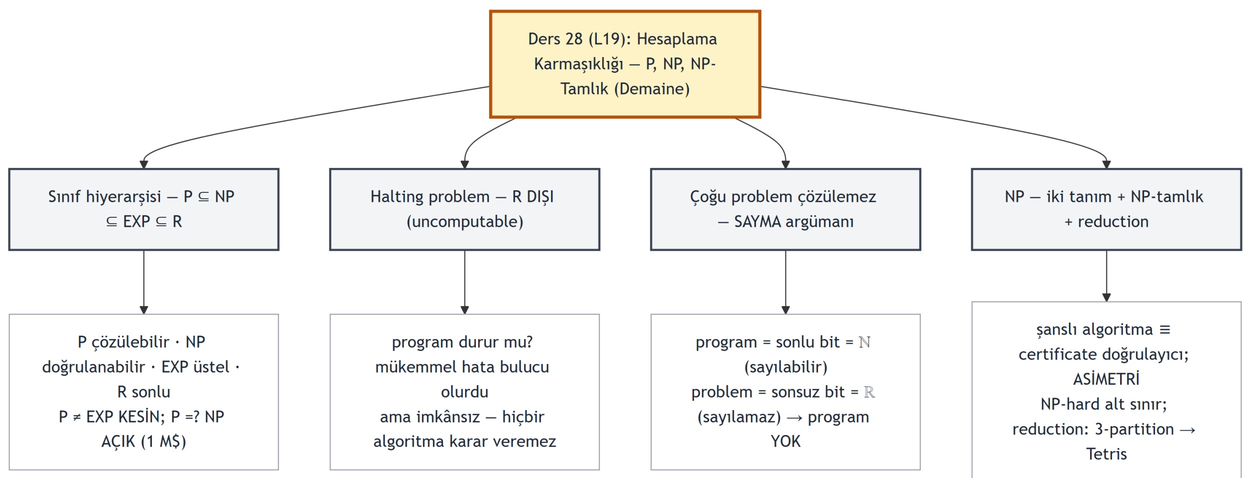
> *"cover an entire field, which is computational complexity."* — Demaine, 0:19

Şaşırtıcı sonuç: **çoğu problemin hiçbir algoritması yoktur.**

> *"most problems actually have no algorithm."* — Demaine, 1:32

Üç ana fikir:

1. **Sınıf hiyerarşisi** — $P \subseteq NP \subseteq EXP \subseteq R$; halting problem R dışında (uncomputable); çoğu problem çözülemez.
2. **NP** — "şanslı algoritma" (hep doğru tahmin) veya "certificate ile polinom-zamanda doğrulanabilir".
3. **NP-tamlık + reduction** — bir problemi bilinen-zor bir probleme indirgeyerek zorluğunu kanıtla.



Şekil 35.1: Ders 28'in (L19) kavram haritası: kök = Hesaplama Karmaşıklığı (Demaine) — bütün bir alan tek derste, perspektif tersine döner ALT SINIR tarafı yani neden iyi çözemeyiz. Dört dal — (1) Sınıf hiyerarşisi P alt küme NP alt küme EXP alt küme R: P polinom zamanda CÖZÜLEBİLİR verimli örnek negatif çevrim tespiti Bellman-Ford; NP polinom zamanda DOĞRULANABİLİR örnek Tetris ve subset sum; EXP üstel zamanda çözülebilir örnek n çarpı n satranç; R herhangi sonlu zamanda decidable; P ile EXP farkı KESİN ama P ile NP eşit mi AÇIK bir milyon dolarlık problem. (2) Halting problem R DIŞI uncomputable: verilen bir program durur mu sorusu, mükemmel bir hata bulucu olurdu ama imkânsızdır hiçbir algoritma karar veremez. (3) Çoğu problem çözülemez SAYMA argümanı: program sonlu bit dizisi yani doğal sayı SAYILABİLİR countable, karar problemi her girdi için evet hayır yani sonsuz bit dizisi yani sıfır bir aralığında reel sayı SAYILAMAZ uncountable, reel kat kat fazla yani problemlere yetecek program YOK yani çoğu problem çözülemez. (4) NP sınıfı iki eşdeğer tanım artı NP-tamlık: Tanım bir ŞANSLI algoritma tahmin yapabilir ve bir evet yolu varsa hep doğru tahmin eder; Tanım iki DOĞRULAYICI girdi artı certificate alıp polinom zamanda kontrol eder, ASİMETRİ evet ispatlanabilir hayır değil; NP-hard NP kadar zor alt sınır, NP-complete NP kesişim NP-hard NP'nin en zoru Tetris TSP üç-SAT; REDUCTION bilinen-zor problemi hedefe indirge örnek üç-partition oku Tetris zorluk kanıtı. Birleştirici tema: problemler zorluk ekseninde sınıflanır P verimli çözülebilir alt küme NP verimli doğrulanabilir alt küme EXP alt küme R, çoğu problem R'nin bile dışındadır halting gibi uncomputable, NP-tam problemler NP'nin en zorudur ve bir problemi NP-tam bir probleme indirgeyerek P eşit değil NP ise verimli çözülemez kanıtlanır, şans engineer edilemez modern güvenliğin dayanağıdır.

35.2 1. Karmaşıklık: Alt Sınır Tarafı

Algoritmalar "iyi çözebiliriz" gösterir; karmaşıklık "**iyi çözemeyiz**" (alt sınır) kanıtlar. Eski alt sınırlar (sıralama $n \log n$) "n vs $n \log n$" düzeyindeydi; bu ders **polinom vs üstel** düzeyinde. Polinom = iyi süre (her zaman hedefimiz); üstel = genelde kolay elde edilir (kaba kuvvet).

> *"cover an entire field, which is computational complexity."* — Demaine, 0:19

Perspektif tersine döner: dört DP dersi boyunca "ne kadar hızlı çözebiliriz?" diye sorduk; şimdi soru "**bu problem verimli çözülebilir mi, yoksa imkânsız mı?**"

35.3 2. P, EXP, R Hiyerarşisi

- **P** = polinom-zamanda (girdi boyutu n'e polinom) çözülebilir problemler — "verimli".
- **EXP** = üstel-zamanda (2^{n^c}) çözülebilir. $P \subseteq EXP$ (ve kesin $P \neq EXP$).
- **R** = **sonlu** (recursive) zamanda çözülebilir. $P \subseteq EXP \subseteq R$.

> *"P… is the set of all problems solvable in polynomial time."* — Demaine, 1:50

Örnekler: $n \times n$ **satranç** EXP'de ama P'de değil (kanıtlı); **negatif çevrim tespiti** P'de (Bellman-Ford); **Tetris** (mükemmel-bilgi) EXP'de, P'de mi bilinmiyor. Çoğu problem **karar problemi**dir (yes/no çıktı).

Şekil 35.2 bu iç içe yapıyı tek bir zorluk ekseni üzerinde gösterir: $P \subseteq NP \subseteq EXP \subseteq R$ dört yarım-kapsül (ortak sol kenar, kademeli sağ kenar) hâlinde nest edilir; sınıf üyeliği bir **üst sınır** ("şu kadar zamanda çözülebilir"), NP-hard ise bir **alt sınır** ("bu noktanın sağında") olarak ayrışır. Motor tanığı NP'nin somut anlamını çalıştırır: subset sum'ın EVET girdisi ($A = \{2, 5, 7, 8, 9\}$, $T = 21$) için kısa-doğrulanabilir bir certificate vardır ($\{5, 7, 9\}$), HAYIR girdisi ($T = 25$) için yoktur — "NP = polinom-doğrulanabilir" rozetinin çalışan kanıtı.

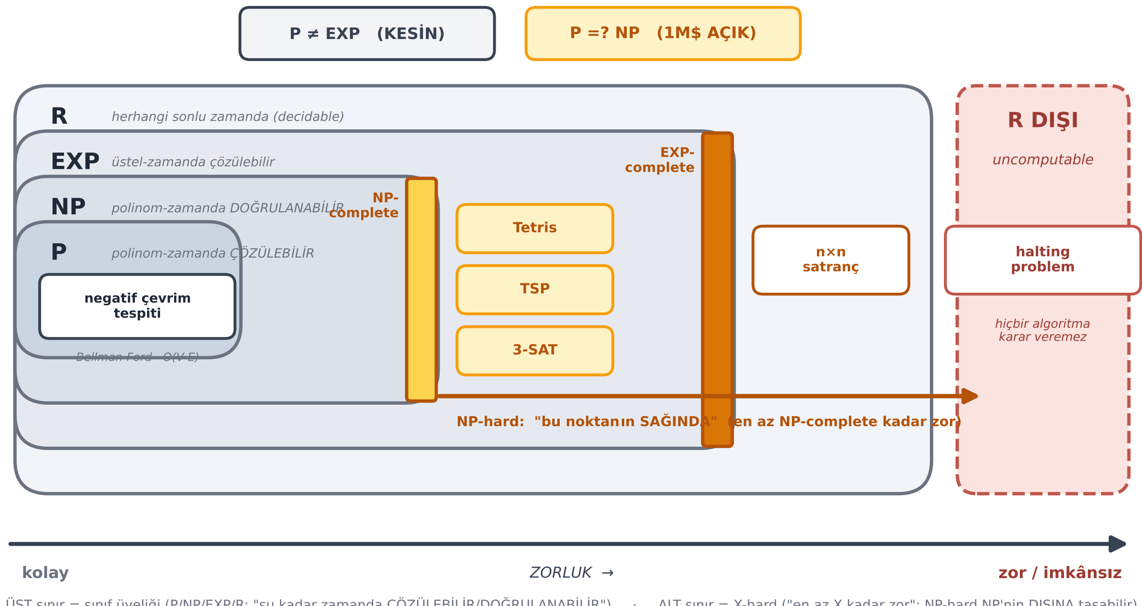
35.4 3. Halting Problem: Uncomputable

R'nin de dışında problemler var. En ünlüsü **halting problem**: "verilen bir program durur mu (sonsuz döngü var mı)?"

> *"Given a computer program, does it ever halt?"* — Demaine, 10:49

Bu, mükemmel bir hata bulucu olurdu — ama **imkânsızdır**: tüm girdileri çözen bir algoritma yoktur. Böyle problemlere **uncomputable** (R dışı) denir.

> *"We call such problems uncomputable."* — Demaine, 11:36



Şekil 35.2: Hesaplama sınıfları hiyerarşisi (Demaine L19 §2 İMZA + Kontrol-4, 1:50): P ⊆ NP ⊆ EXP ⊆ R — soldan sağa zorluk artar, R dışı uncomputable. İç içe DÖRT yarım-kapsül (ortak sol kenar, kademeli sağ): P 'polinom-zamanda ÇÖZÜLEBİLİR' (örnek negatif çevrim tespiti Bellman-Ford O(V·E)), NP 'polinom-zamanda DOĞRULANABİLİR', EXP 'üstel-zamanda', R 'herhangi sonlu zamanda decidable'. NP sağ ucu amber şerit = NP-complete (Tetris/TSP/3-SAT rozetleri); EXP sağ ucu = EXP-complete (n×n satranç). NP-hard = NP-complete'ten SAĞA uzanan amber ok (alt sınır, kapsül dışına taşar). Eksenin SAĞ ÖTESİ kırmızı-soluk 'R DIŞI uncomputable' (halting problem). İki rozet: P ≠ EXP KESİN + P =? NP 1M$ AÇIK. Alt not: ÜST sınır = sınıf üyeliği · ALT sınır = X-hard. Motor tanığı (assert): build_subset_example == ([2,5,7,8,9],21,25); subset_sum_certificate(A,21)==[5,7,9] toplam 21; verify_subset_certificate(A,21,[5,7,9]) is True; subset_sum_certificate(A,25) is None (HAYIR için kısa kanıt yok); verify_subset_certificate(A,21,[2,8]) is False.

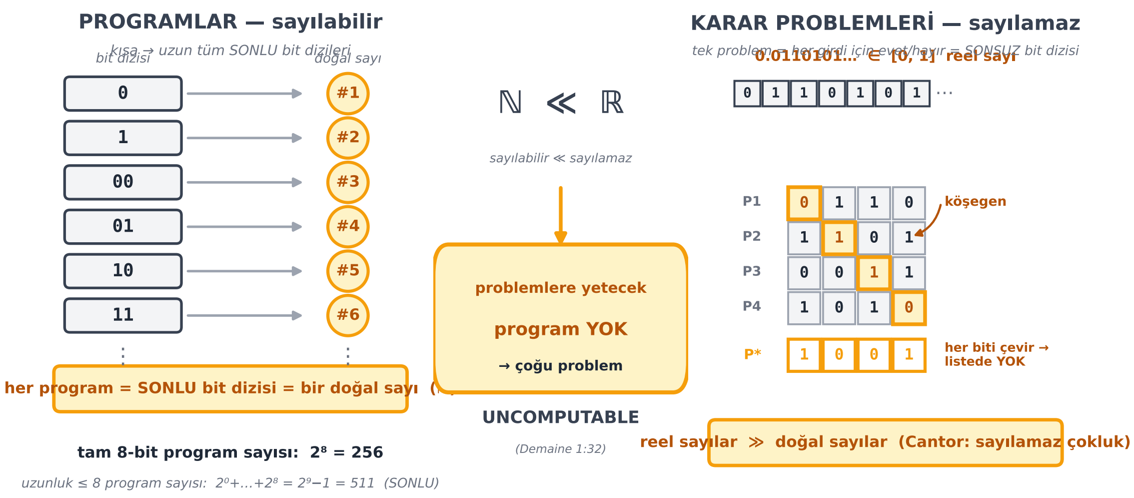
35.5 4. Çoğu Problem Çözülemez

Şaşırtıcı gerçek: çoğu karar problemi uncomputable. **Sayma argümanı**:

- **Program** = sonlu bit dizisi → bir doğal sayı. Doğal sayılar **sayılabilir (countable)**.
- **Karar problemi** = her girdi (sonsuz girdi) için yes/no → sonsuz bit dizisi → $[0, 1]$ aralığında bir **reel sayı**. Reel sayılar **sayılamaz (uncountable)**.

Reel sayılar, doğal sayılardan "çok daha fazladır" → problemlere yetecek kadar program **yoktur** → çoğu problem çözülemez. Şans eseri, **önemsediğimiz çoğu problem R'dedir** ($n \times n$ satranç bile — yalnız yavaş).

Şekil 35.3 bu argümanı iki panelde somutlaştırır: solda programlar kısa→uzun sonlu bit dizileri olarak doğal sayılara eşlenir (tam 8-bit program sayısı $= 2^8 = 256$, sonlu); sağda tek bir karar problemi sonsuz bit dizisi → $[0, 1]$ reel sayı olur ve mini Cantor köşegen şeması (köşegeni ters çevir → listede olmayan problem) reel'in "kat kat fazla" olduğunu gösterir. Ortadaki köprü $\mathbb{N} \ll \mathbb{R}$ → "problemlere yetecek program yok" çıkarımını taşır.



Şekil 35.3: Sayma argümanı (Demaine L19 §4 İMZA, 1:32): programlar SAYILABİLİR, problemler SAYILAMAZ → çoğu problem ÇÖZÜLEMEZ (uncomputable). SOL panel 'PROGRAMLAR — sayılabilir': kısa→uzun sonlu bit dizileri (0,1,00,01,10,11) doğal sayılara (#1..#6) eşlenmiş; rozet 'her program = SONLU bit dizisi = bir doğal sayı (ℕ)'; tam 8-bit program sayısı $2^8 = 256$, uzunluk ≤ 8 program sayısı $2^9 - 1 = 511$ (SONLU). ORTA köprü: ℕ ℝ (sayılabilir sayılamaz) → amber kutu 'problemlere yetecek program YOK → çoğu problem UNCOMPUTABLE (Demaine 1:32)'. SAĞ panel 'KARAR PROBLEMLERİ — sayılamaz': tek problem = her girdi için evet/hayır = SONSUZ bit dizisi → 0.0110101… ∈ [0,1] reel; mini Cantor 4×4 köşegen tablosu + çevrilmiş köşegen P* 'her biti çevir → listede YOK'; rozet 'reel sayılar doğal sayılar (Cantor)'. Alt not: neyse ki önemsediğimiz çoğu problem R'dedir. Veri CANLI hesap (assert): prog_count_exact(8)==256; prog_count_upto(8)==511; diag_flip(Cantor) listedeki hiçbir satıra eşit değil + köşegen biti ters çevrilmiş.

35.6 5. NP: Tanım 1 — Şanslı Algoritma

NP ($P \subseteq NP \subseteq EXP$), yalnız **karar problemleri** için. P gibi "polinom-zamanda çözülebilir" ama **şanslı algoritma** ile.

> *"a lucky algorithm, which can make guesses and always makes the right guess."* — Demaine, 22:46

Şanslı algoritma: tahmin yapabilir ve **bir "evet"e götüren yol varsa** hep doğru tahmin eder (non-deterministic polynomial time). DP'deki "tahmin et" sezgisinin gerçek hâli — ama DP tüm seçenekleri denerken, şanslı algoritma yalnız doğru tahminin maliyetini öder. **Asimetri:** "evet"i bulur, "hayır"ı garanti etmez.

35.7 6. NP: Tanım 2 — Doğrulayıcı

Eşdeğer tanım: NP = **polinom-zamanda doğrulanabilen (checkable)** karar problemleri.

> *"NP is a set of decision problems that can be checked in polynomial time."* — Demaine, 31:31

Bir **doğrulayıcı (verifier)** girdi + **certificate** (kanıt) alır, polinom-zamanda yes/no der. İki koşul: (1) her **evet** girdisi için, doğrulayıcıyı "evet" dedirten bir certificate **vardır**; (2) her **hayır** girdisi için, *hiçbir* certificate doğrulayıcıyı "evet" dedirtmez. Yani "evet"ler ispatlanabilir, "hayır"lar değil. Örnek: subset sum (alt kümeyi göster → topla, doğrula); Tetris (hamle dizisi = certificate, kuralları uygula).

Şekil 35.4 NP'nin iki tanımını tek figürde, motor üzerinde birleştirir: üst panelde şanslı makine $A = \{2, 5, 7, 8, 9\}, T = 21$ için karar ağacında hep doğru tahmin eder ve $\{5, 7, 9\} \to 21$ EVET yaprağına ulaşır (amber yol); "tahminleri yazarsan certificate olur" köprüsü Tanım-2'ye bağlanır. Alt panelde doğrulayıcı, certificate $\{5, 7, 9\}$'u $O(|\text{cert}| + |A|)$'da kontrol eder ($5 + 7 + 9 = 21$✓), kötü certificate $\{5, 7, 8\}$'i ($= 20 \neq 21$) reddeder; asimetri rozeti $T = 25 \to$ certificate `None` ile "hayır kısa-kanıtlanamaz"ı gösterir.

35.8 7. P ≠ NP Konjektürü

$P \subseteq NP$ biliniyor; $P = NP$ **mi?** bilinmiyor (1 milyon dolarlık açık problem). Çoğunluk $P \neq NP$ sanır.

> *"you cannot engineer luck."* — Demaine, 39:17

Sezgi: NP = şansla çözülebilir, P = şansız çözülebilir; $P = NP$ olsaydı "şans hiçbir şey kazandırmazdı" — ki tuhaf. Eşdeğer ifade:

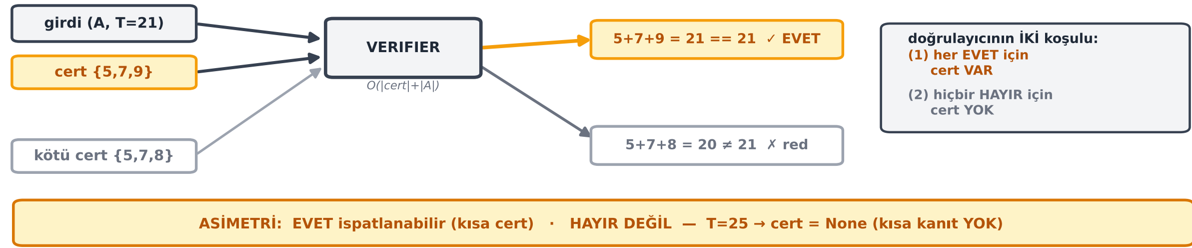
> *"it's harder to come up with proofs than it is to check them."* — Demaine, 39:56

(İspat bulmak, doğrulamaktan zor.)

NP'nin iki eşdeğer tanımı: ŞANSLI tahmin ≡ hızlı DOĞRULAMA

Tanım 1 — ŞANSLI algoritma: doğru tahmin dizisi = certificate

Tanım 2 — DOĞRULAYICI: certificate'i polinom zamanda kontrol et



Şekil 35.4: NP'nin iki eşdeğer tanımı (Demaine L19 §5-6, 22:46 + 31:31): ŞANSLI tahmin ≡ hızlı DOĞRULAMA. ÜST panel 'Tanım 1 — ŞANSLI algoritma': karar ağacı kökte A={2,5,7,8,9}, T=21; her öğede koy/koyma tahmini; şanslı makine bir EVET-yolu VARSA hep doğru tahmin eder ve {5,7,9}→21 ✓ EVET yaprağına ulaşır (amber yol, diğer dallar soluk); köprü kutusu 'tahminleri YAZARSAN → certificate olur = Tanım 2'ye köprü (Demaine 22:46)'. ALT panel 'Tanım 2 — DOĞRULAYICI': akış (A,T)+cert {5,7,9} → VERIFIER O(|cert|+|A|) → '5+7+9 = 21 == 21 ✓ EVET'; kötü cert {5,7,8} → '5+7+8 = 20 ≠ 21 × red'; iki koşul kutusu (1) her EVET için cert VAR (2) hiçbir HAYIR için cert YOK; ASİMETRİ rozeti 'EVET ispatlanabilir · HAYIR DEĞİL — T=25 → cert = None'. Veri MOTORDAN (assert): build_subset_example == ([2,5,7,8,9],21,25); subset_sum_certificate(A,21)==[5,7,9] toplam 21; verify_subset_certificate(A,21,[5,7,9]) is True; verify_subset_certificate(A,21,[5,7,8]) is False (toplam 20); subset_sum(A,25)[0] is False; subset_sum_certificate(A,25) is None.

35.9 8. NP-hard ve NP-complete

- **NP-hard:** NP'deki *tüm* problemler kadar zor (alt sınır). NP'deki her problem buna **indirgenir**.
- **NP-complete:** NP **ve** NP-hard (NP'nin en zoru).

> *"NP-complete… the problems that are in NP and they are NP-hard."* — Demaine, 43:11

$P \neq NP$ ise, NP-tam problemler **P'de değildir** (polinom-zamanda çözülemez). Tetris NP-tamdır → $P \neq NP$ varsayımıyla, bir Tetris dizisini hayatta kalıp kalamayacağını polinom-zamanda hesaplayamazsın.

35.10 9. Reduction: İndirgeme ile Zorluk Kanıtı

Reduction (indirgeme): A'yı B'ye çevir — A girdisini B girdisine, B çözümünü A çözümüne. Algoritma için: çözmek istediğini bildiğine indirge. **Zorluk için: bilinen-zor problemi hedefe indirge.**

> *"Reductions are the easy way to use algorithms."* — Demaine, 44:38

Anlamı: $A \leq B$ reduction'ı varsa, "A en az B kadar kolay" = "**B en az A kadar zor**". Gördüğümüz indirgemeler: ağırlıksız SP → ağırlıklı SP (ağırlık$=1$); tamsayı-ağırlık → ağırlıksız (kenar böl); en uzun yol → en kısa yol (negatifle). NP-hardness kanıtı: bilinen NP-tam bir problemi (örn. **3-partition**) hedefe indirge. 3-partition → jigsaw puzzle (NP-hard); 3-partition → Tetris (NP-hard).

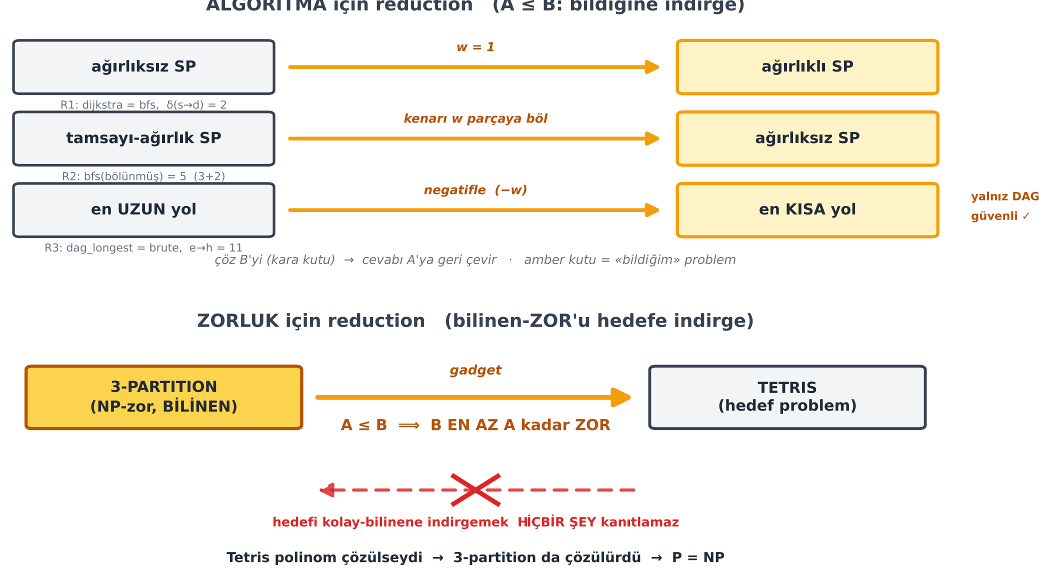
Şekil 35.5 reduction'ın **yönünü** — sınavların klasiği — iki panelde ayırır. Üst panel algoritma için reduction'ı ($A \leq B$: bildiğine indirge) üç çalışır motor-tanığıyla gösterir: **R1** ağırlıksız → ağırlıklı SP ($w = 1$; `dijkstra` $=$ `bfs`, $\delta(s \to d) = 2$, 60 rastgele çizgede aynı), **R2** tamsayı-ağırlık → ağırlıksız (kenar böl; $\delta(t) = 5$), **R3** en uzun → en kısa yol (negatifle; yalnız DAG güvenli, $e \to h = 11$, `brute` ile aynı). Alt panel zorluk için reduction'ı verir: bilinen-zor **3-partition** → **Tetris** (gadget), büyük amber yön oku "$A \leq B \Rightarrow B$ en az A kadar zor"; ters yön kırmızı çarpıyla yasaklanır ("hedefi kolay-bilinene indirgemek hiçbir şey kanıtlamaz").

Bu indirgemelerden ikincisi (**R2:** tamsayı-ağırlık → ağırlıksız) tek başına da öğreticidir: çünkü "ücretsiz" değildir. Şekil 35.6 bu kenar-bölme reduction'ını motor üzerinde açar: solda orijinal ağırlıklı çizge (a, b, c, d; Dijkstra δ: a:0, b:3, c:1, d:5), sağda her w-ağırlıklı kenar w parçaya bölünmüş ağırlıksız çizge — BFS seviyeleri orijinal düğümlerde **birebir aynı** değerleri verir. Ama bedel: yeni düğüm sayısı $= \Sigma w = 11$; ağırlıklar büyürse çizge **patlar** — bu, subset sum'ın T büyüdükçe patlaması (Ders 27, pseudopolinom) ile aynı köprüdür: girdi *bit sayısında* üstel.

35.11 10. NP-complete Örnekleri

NP-tam problemler her yerde:

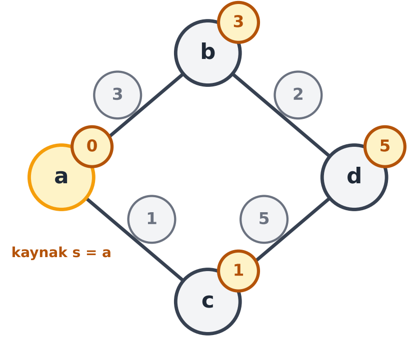
- **Optimizasyon/sayı:** subset sum (pseudopoli en iyisi), 3-partition, knapsack, **TSP** (gezgin satıcı: tüm düğümleri gezen en kısa yol).
- **Çizge:** en uzun **basit** yol, **3-renklendirme** (2 polinom!), maksimum klik.
- **Mantık: SAT / 3-SAT** (Boole formülünü doğru yapan atama var mı).
- **Oyun/bulmaca:** Minesweeper, Sudoku, Super Mario, Zelda, Pokemon (bazıları P-space, NP'den de zor).



Şekil 35.5: Reduction YÖNÜ — sınavların klasiği (Demaine L19 §9 İMZA, 44:38). ÜST panel 'ALGORİTMA için reduction (A ≤ B: bildiğine indirge)': üç çalışır motor-tanığı — R1 ağırlıksız SP → ağırlıklı SP (w=1; dijkstra=bfs, δ(s→d)=2), R2 tamsayı-ağırlık SP → ağırlıksız SP (kenarı w parçaya böl; bfs(bölünmüş)=5, 3+2), R3 en UZUN yol → en KISA yol (negatifle −w; dag_longest=brute, e→h=11, yalnız DAG güvenli ✓); çöz B'yi kara kutu → cevabı A'ya geri çevir, amber kutu = «bildiğim» problem. ALT panel 'ZORLUK için reduction (bilinen-ZOR'u hedefe indirge)': 3-PARTITION (NP-zor BİLİNEN) → gadget → TETRIS (hedef); büyük amber yön oku 'A ≤ B ⟹ B EN AZ A kadar ZOR'; ters-yön kırmızı çarpılı ok 'hedefi kolay-bilinene indirgemek HİÇBİR ŞEY kanıtlamaz'; zincir 'Tetris polinom çözülseydi → 3-partition da çözülürdü → P = NP (Demaine 44:38)'. Veri MOTORDAN (assert): R1 dijkstra(w=1)==bfs, δ(s→d)=2; R2 reduce_integer_to_unweighted + bfs == dijkstra, δ(t)=5; R3 dag_longest_path==brute_dag_longest, e→h=11; subset cert [5,7,9] toplam 21 (NP-tam çıpa).

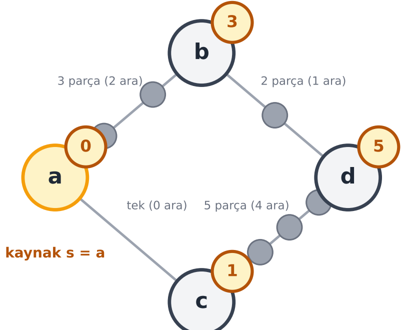
R2 redüksiyonu: tamsayı-ağırlık → ağırlıksız · bfs(bölünmüş) == dijkstra(orijinal)

Orijinal: AĞIRLIKLI çizge · Dijkstra δ rozetleri



kenar ağırlığı = adım maliyeti · δ(a,d)=5 (a→b→d: 3+2=5 < a→c→d: 1+5=6)

Kenar-bölünmüş: AĞIRLIKSIZ çizge · BFS seviye rozetleri (AYNI)



bedel: Σw = 11 yeni düğüm — sayılar büyükse çizge PATLAR (pseudopolinom bedel · Ders 27 köprüsü)

Şekil 35.6: R2 redüksiyonu (Demaine L19 §9, R2): tamsayı-ağırlık → ağırlıksız · bfs(bölünmüş) == dijkstra(orijinal). SOL panel 'Orijinal: AĞIRLIKLI çizge · Dijkstra δ rozetleri': 4 düğüm a,b,c,d kenar ağırlıkları (a→b:3, a→c:1, b→d:2, c→d:5); δ rozetleri motordan a:0, b:3, c:1, d:5; δ(a,d)=5 (a→b→d: 3+2=5 < a→c→d: 1+5=6); kaynak s=a. SAĞ panel 'Kenar-bölünmüş: AĞIRLIKSIZ çizge · BFS seviye rozetleri (AYNI)': her w-kenar w parçaya bölünür (a→b 3 parça/2 ara, a→c tek/0 ara, b→d 2 parça/1 ara, c→d 5 parça/4 ara, gri ara düğümler); BFS seviye rozetleri orijinal düğümlerde AYNI (amber) çünkü ağırlıksız kenar = 1 adım, w parça = w adım. Alt not: bedel Σw = 11 yeni düğüm — sayılar büyükse çizge PATLAR (pseudopolinom bedel · Ders 27 köprüsü). Veri MOTORDAN (assert): reduce_integer_to_unweighted + bfs orijinal düğümlerde == dijkstra (a:0,b:3,c:1,d:5); ara düğüm sayısı = Σw−|E| = 11−4 = 7; her kenarın parça sayısı = ağırlığı.

EXP-complete: $n \times n$ satranç (iki-oyunculu doğası gereği). (LCS de **n dizi** ile NP-tam olur; sabit sayı dizi polinom.)

Şekil 35.7 bu kataloğu bir kart-galerisi olarak dizer: dört kategori sütunu (sayı, çizge, mantık, oyun/bulmaca) altında 12 NP-tam problem, her biri "NP-tam" rozetli. Üç kontrast rozeti NP-tam'ın **ince çizgisini** vurgular: "3-renklendirme NP-tam ama 2-renklendirme polinom", "en uzun basit yol NP-tam ama en kısa yol polinom", "subset sum pseudopolinom var ama gerçek polinom yok ($P \neq NP$ ise)". Sağ altta EXP-complete kutusu ($n \times n$ satranç, iki-oyunculu) ve LCS'in n-dizi notu; alt şerit "$P \neq NP$ ise hiçbiri polinom-zamanda çözülemez (yaklaşık/sezgisel/SAT-solver'a git)".

35.12 Bu Dersin Özeti

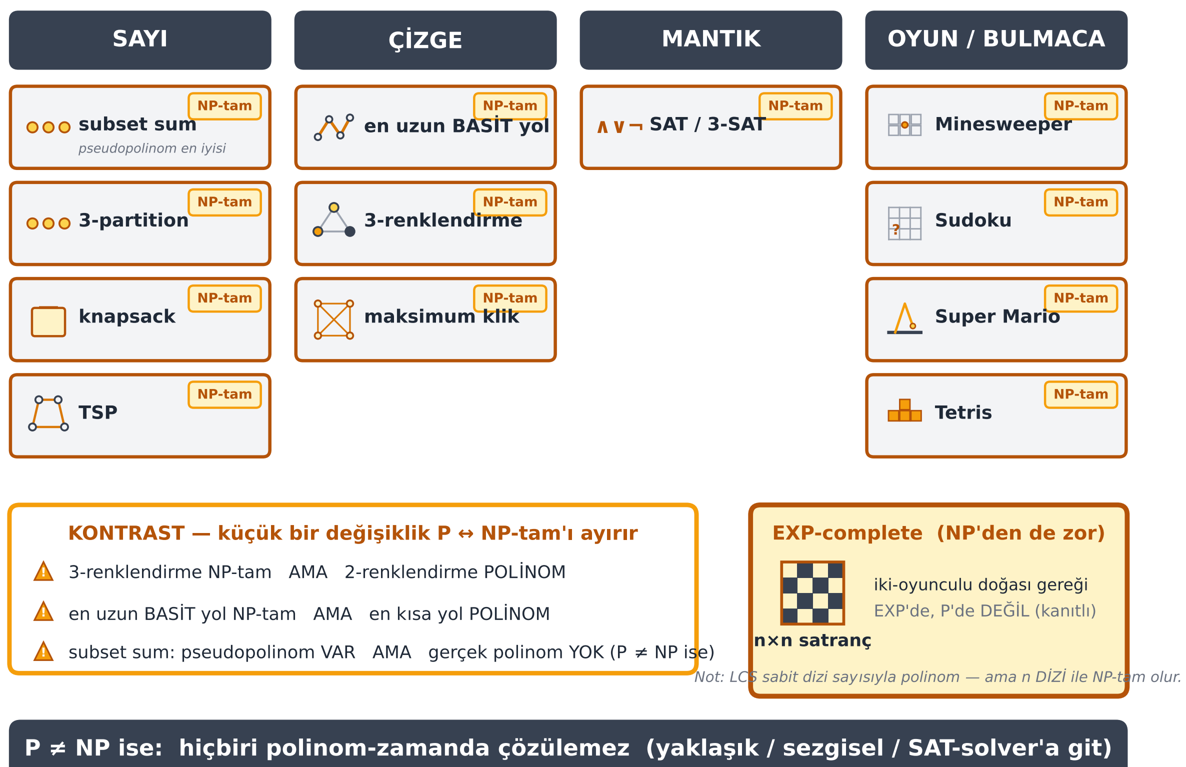
1. $P \subseteq NP \subseteq EXP \subseteq R$; $P \neq EXP$ kesin; $P = NP$ açık (1 M$).
2. **Halting problem** R dışında (uncomputable); çoğu problem çözülemez (countable program vs uncountable problem).
3. **NP** = şanslı algoritma (hep doğru tahmin) = certificate'le polinom-doğrulanabilir.
4. **Asimetri**: "evet" ispatlanabilir, "hayır" değil.
5. $P \neq NP$ (varsayım): "şans engineer edilemez"; ispat bulmak doğrulamaktan zor.
6. **NP-hard** = NP kadar zor; **NP-complete** = $NP \cap NP$-hard (NP'nin en zoru).
7. **Reduction**: bilinen-zor problemi hedefe indirge → zorluk kanıtı (3-partition → Tetris).

> **!** Tek Bir Cümle
>
> P (verimli çözülebilir) $\subseteq$ NP (verimli doğrulanabilir) $\subseteq$ EXP $\subseteq$ R; çoğu problem R'nin bile dışındadır; NP-tam problemler (Tetris, TSP, 3-SAT) NP'nin en zorudur ve bir problemi NP-tam bir probleme indirgeyerek "$P \neq NP$ ise verimli çözülemez" kanıtlanır.

35.13 Kontrol Soruları

> **i** Soru 1: «Çoğu problem çözülemez» iddiası hangi sayma argümanına dayanır?
>
> **Cevap:** Bir **program** sonlu bir bit dizisidir → bir doğal sayıya karşılık gelir; doğal sayılar **sayılabilir (countable)**. Bir **karar problemi** ise, her olası girdi (sonsuz girdi) için bir yes/no belirtir → sonsuz bir bit dizisi → $[0, 1]$ aralığında bir **reel sayı**; reel sayılar **sayılamaz (uncountable)**. Sayılamaz çokluk, sayılabilirden "kat kat fazladır" — yani problem sayısı, program sayısından çok daha fazla. Her program en fazla bir problemi çözdüğünden, problemlere yetecek program **yoktur** → çoğu problem çözülemez (uncomputable). (Neyse ki önemsediğimiz çoğu problem R'dedir.)

> **i** Soru 2: NP'nin iki tanımı (şanslı algoritma / doğrulayıcı) nasıl eşdeğer, ve «asimetri» nedir?
>
> **Cevap: Şanslı algoritma:** tahmin yapan, bir "evet"e götüren yol varsa hep doğru tahmin eden polinom-zaman algoritması. **Doğrulayıcı:** girdi + certificate alıp polinom-zamanda kontrol eden algoritma. Eşdeğerlik: şanslı algoritmanın "doğru tahminleri" tam olarak certificate'tır — tahminleri yazarsan certificate olur; certificate verilirse doğrulayıcı onu "tahmin gibi" kontrol eder. **Asimetri:** her ikisi de



Şekil 35.7: NP-tam Hayvanat Bahçesi (Demaine L19 §10 İMZA): NP-tam problemler her yerde. 4 kategori sütunu kart-galerisi (her kart 'NP-tam' rozetli, mini kavramsal ikon, SAYI YOK): SAYI (subset sum 'pseudopolinom en iyisi', 3-partition, knapsack, TSP), ÇİZGE (en uzun BASİT yol, 3-renklendirme, maksimum klik), MANTIK (SAT / 3-SAT), OYUN/BULMACA (Minesweeper, Sudoku, Super Mario, Tetris). KONTRAST rozetleri — küçük bir değişiklik P ↔ NP-tam'ı ayırır: 3-renklendirme NP-tam AMA 2-renklendirme POLİNOM · en uzun BASİT yol NP-tam AMA en kısa yol POLİNOM · subset sum pseudopolinom VAR AMA gerçek polinom YOK (P ≠ NP ise). EXP-complete kutusu (sağ alt): n×n satranç (iki-oyunculu doğası gereği, EXP'de P'de DEĞİL) + 'LCS sabit dizi polinom ama n DİZİ ile NP-tam'. Alt şerit: P ≠ NP ise hiçbiri polinom-zamanda çözülemez (yaklaşık/sezgisel/SAT-solver'a git). KAVRAMSAL figür (assert): 4 kategori [4,3,1,4] = 12 NP-tam problem L19 §10 birebir; 3 kontrast çifti; EXP-complete satranç ayrı.

yalnız “**evet**” cevaplarını garanti eder — evet için certificate vardır/doğrulayıcı evet der; ama “hayır” için kısa bir kanıt yoktur (subset sum’da “bu sayı temsil edilemez”i ispatlamanın bilinen kısa yolu yok). Bu yüzden Tetris’i “hayatta kalınabilir mi?” diye tanımlamak NP’dedir, “kalınamaz mı?” değil.

Soru 3: Reduction ile zorluk nasıl kanıtlanır? A ≤ B neyi ima eder?

Cevap: A’dan B’ye bir reduction (A girdisini B girdisine, B çözümünü A çözümüne çeviren), “**A en az B kadar kolaydır**” der (B’yi çözebilirsem A’yı da çözerim). Mantıksal karşıt-tersi: “**B en az A kadar zordur**”. Zorluk kanıtı için: **bilinen NP-tam bir problemi** (A = 3-partition) **hedef probleme** (B = Tetris) indirgeriz; bu, “Tetris en az 3-partition kadar zor” = Tetris NP-hard demektir. Yani Tetris için polinom algoritma olsaydı, 3-partition (ve dolayısıyla tüm NP) polinom-zamanda çözülürdü → $P = NP$. $P \neq NP$ varsayımıyla, Tetris polinom-zamanda çözülemez.

Soru 4: P, NP, EXP, R, NP-hard, NP-complete sınıfları zorluk ekseninde nasıl konumlanır?

Cevap: Tek bir “zorluk” ekseni düşün (solda kolay, sağda zor). **P** (en solda küçük segment), **NP** (P’yi içerir, biraz daha geniş), **EXP** (NP’yi içerir), **R** (EXP’yi içerir, sonlu-zaman) — hepsi **üst sınır** (“bu çizginin solundasın”). **NP-hard** bir **alt sınır**dır (“şu noktanın sağındasın” — NP’nin en zorundan itibaren sağa, sonsuza). **NP-complete** = $NP \cap NP$-hard, yani NP’nin tam sağ ucu (hem NP’de hem NP kadar zor; Tetris, TSP). **EXP-complete** = EXP’nin sağ ucu ($n \times n$ satranç). $P = NP$ olup olmadığını bilmediğimizden, P ile NP arası “boşluk” var mı kesin değil — ama $P \neq EXP$ kesin.

35.14 Egzersizler

Egzersiz 1. P, NP, EXP, R sınıflarını birer örnek problemle eşle; her birinin “üst sınır” mı “alt sınır” mı belirttiğini yaz.

Egzersiz 2. Halting problem’in neden uncomputable olduğunu, “çoğu problem çözülemez” sayma argümanıyla ilişkilendir.

Egzersiz 3. NP’nin iki tanımının (şanslı algoritma / doğrulayıcı) Tetris üzerinde nasıl çalıştığını adım adım yaz.

Egzersiz 4. 3-partition → Tetris indirgemesinin neden Tetris’i NP-hard yaptığını, $A \leq B$ mantığıyla açıkla.

Egzersiz 5. Verilen bir optimizasyon problemini (örn. TSP) karar problemine çevir ($\leq x$ var mı) ve ikili aramayla optimumun nasıl bulunacağını göster.

35.15 Sonraki Ders İçin Hazırlık

⚠ Sonraki: Ders 29 (PS9) — Problem Oturumu 9 (Justin Solomon)

Ders 29 (PS9): Problem Oturumu 9 (karmaşıklık + genel tekrar). Hoca değişir: bu oturumu **Justin Solomon** yürütür (Demaine'in ders dizisinden farklı). Bir problem oturumuyla, gördüğümüz tüm konuları (özellikle karmaşıklık ve indirgeme) gerçek problemlerde pekiştiriyoruz. P/NP ayrımı, NP-hardness kanıtı ve reduction sezgisi merkezde.
Ders 29 Öncesi Yapılacak:

- Bu dersin egzersizlerini, özellikle Egzersiz 3 (NP iki tanım) ve 4 (reduction) çöz.
- P/NP/EXP/R hiyerarşisini ve NP-hard/NP-complete ayrımını ezberden çiz.
- Ana cümleyi tekrar oku: *"Bilinen-zor problemi hedefe indirge → zorluk kanıtı; şans engineer edilemez."*

35.16 Anahtar Kavramlar (Cheat Sheet)

Kavram	Tanım	Sayfada
P	Polinom-zaman çözülebilir; "verimli"	Böl. 2
NP	Şanslı algoritma / certificate'le doğrulanabilir	Böl. 5-6
EXP / R	Üstel-zaman / sonlu-zaman; $P \subseteq NP \subseteq EXP \subseteq R$	Böl. 2
Uncomputable	R dışı (halting); çoğu problem böyle	Böl. 3-4
$P \neq NP$	Varsayım; "şans engineer edilemez"	Böl. 7
NP-hard	NP kadar zor (alt sınır); her NP buna indirger	Böl. 8
NP-complete	$NP \cap NP$-hard; Tetris, TSP, 3-SAT	Böl. 8, 10
Reduction	$A \leq B \to B$ en az A kadar zor; zorluk kanıtı	Böl. 9

35.17 Builder ve OMSCS Bağlantıları

6 köprü

Bu dersin sınıf hiyerarşisi, NP'nin iki tanımı ve reduction aracı, sistem mühendisliği, kriptografi ve graduate algoritmalarındaki çok sayıda araca bağlanır; köprülerin özeti:

1. $P \neq NP \rightarrow$ **kriptografi/güvenlik** temeli (RSA, hash); "tek yön kolay, ters zor" — modern şifreleme bu konjektürün doğru olduğuna bahse girer.
2. **NP-complete farkındalığı** $\rightarrow$ TSP, scheduling, knapsack; "verimli çözme yok → yaklaşık/sezgisel/**SAT-solver**'a git" (pratikte 3-SAT örnekleri devasa olsa da modern SAT-solver'lar çoğu gerçek örneği çözer).
3. **Reduction** $\rightarrow$ problem indirgeme refleksi; bir problemi bildiğin-zor bir probleme oturtmak — **OMSCS CS 6515 NP-tamlık çekirdeği**'nin temel becerisi.
4. **Halting** $=$ **uncomputable** $\rightarrow$ **statik analiz / bug tespiti sınırı**; "tüm sonsuz döngüleri (veya tüm bug'ları) bulan program yoktur" — derleyici ve linter'ların neden eksiksiz olamadığının teorik nedeni.
5. **Certificate / doğrulayıcı** $\rightarrow$ blockchain ispatı, **zero-knowledge** kanıt, hızlı doğrulama; "çözmek zor, doğrulamak kolay" asimetrisi (`verify_subset_certificate` NP Tanım-2'nin çalışan örneğidir).
6. **Decision problem** $\rightarrow$ optimizasyon $\leftrightarrow$ karar dönüşümü (ikili aramayla); "$\leq x$ var mı?" sorusu binary search'le optimumu verir — graduate algoritmalarında sürekli kullanılan dönüşüm.

! Tek bir şey alıp gideceksen

Problemler zorluk ekseninde sınıflanır: **P** (verimli çözülebilir) $\subseteq$ **NP** (verimli doğrulanabilir) $\subseteq$ **EXP** $\subseteq$ **R** (sonlu-zaman). Çoğu problem R'nin bile dışındadır — halting problem gibi **uncomputable** (çünkü reel-sayı-çoklukta problem, doğal-sayı-çoklukta program var). **NP-tam** problemler (Tetris, TSP, 3-SAT) NP'nin en zorudur; bir problemi NP-tam bir probleme **indirgeyerek** "$P \neq NP$ ise verimli çözülemez" kanıtlanır. NP'de "evet" ispatlanabilir ama "hayır" değildir — ve "şans engineer edilemez" ($P \neq NP$), modern güvenliğin dayanağıdır.

36 Problem Oturumu 9

Pseudopolinom DP oturumu — 0/1 ve sınırsız knapsack, precomputation sinyali ve adversarial minimax

Oturum bilgisi

- **Solomon'un videosu:** YouTube — Problem Session 9 (≈85 dk)
- **OCW sayfası:** MIT 6.006 Problem Session 9
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 29 (PS9)
- **Hoca:** Justin Solomon
- **Okuma süresi:** ≈24 dk
- DP problem oturumlarının **İKİNCİSİ**; pseudopolinom dili "liberating" (Solomon 0:26).

36.1 Bu Problem Oturumu Ne Hakkında?

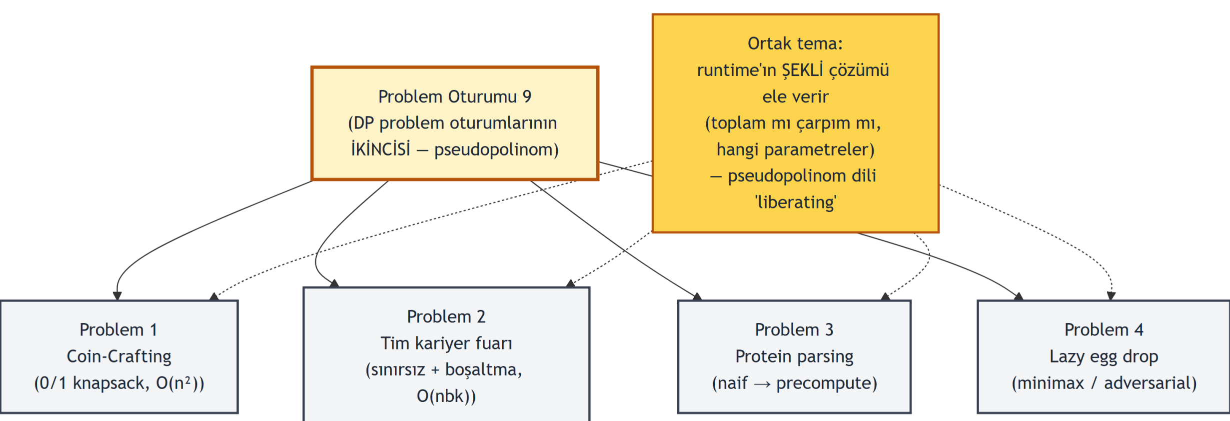
Dokuzuncu problem oturumu (Justin Solomon), **dinamik programlama** üzerine iki oturumun ikincisi; bu kez **pseudopolinom** DP odakta dört problemi **SRTBOT** çerçevesiyle çözer (Şekil 39.1). Solomon "bu set öncekinden kolaydı" der — çünkü pseudopolinom, runtime'da "kullanmaman gereken" sayısal parametreleri (k, b) kullanmaya izin verir, bu da recurrence kurmayı kolaylaştırır.

> *"we learned in class about pseudo polynomial time style dynamic programs. Somehow, that language is a little bit liberating."* — Solomon, 0:26

Beş problem planlanmıştı; Solomon **dördünü** işledi (9-5 "duvar örme" zaman yetmediği için atlandı — bkz. Bölüm 36.7 öncesi son not). Her problem "İfade → Yaklaşım → Çözüm → Karmaşıklık" akışıyla işlenir. Yeni temalar: **0/1 vs sınırsız knapsack**, **zaman ekseninin topolojik sıra vermesi**, **toplam-terimli runtime → precomputation sinyali**, ve **minimax (adversarial) DP**.

Yaklaşım — ortak strateji: istenen runtime'ın şeklini hocanın ipucu gibi oku

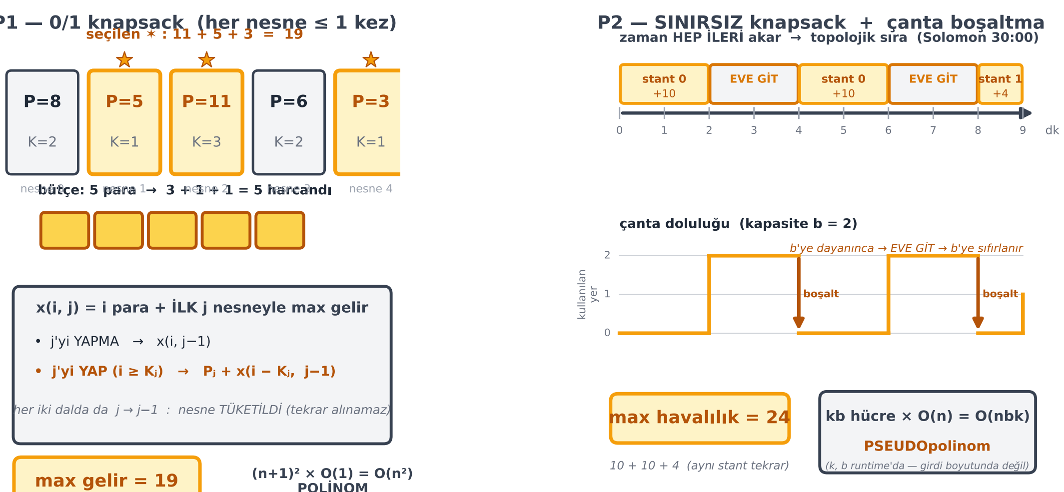
Dört problemin tamamı aynı refleksle başlar: önce alt problemi (Subproblem) tanımla, sonra "bu alt problemde hangi yerel kararı versem geri kalanı bir küçük alt probleme iner?" diye **yerel kaba kuvvet** uygula; bu recurrence'ı (Relation) verir. Pseudopolinom dili bu oturumda işi kolaylaştırır: runtime'da k ve b gibi sayısal parametreleri kullanmaya izin verince recurrence doğal akar. İncelik istenen runtime'ı bir teşhis aracı gibi okumakta: $O(nbk)$ gibi bir **çarpım** doğrudan tablo boyutu × hücre işidir; $O(k|P| + k^2|S|)$ gibi bir **toplam** ise "DP dışında ön-işleme var" der; bir **min içinde max** ise adversarial bir minimax kurar. Bu oturum, DP kasını bu dört SRTBOT'la pseudopolinom yönünde çalıştırır.



Şekil 36.1: Problem Oturumu 9'un kavram haritası: kök (PS9) dört probleme dallanır ve ortadaki ortak tema düğümü dördünü birden yönlendirir. Problem 1 Coin-Crafting'i 0/1 knapsack ile çözer — her nesneden en fazla bir tane, j−1'e geçerek nesneyi tüketir, $(n+1)^2$ hücre × O(1) = $O(n^2)$ polinom. Problem 2 Tim the Beaver kariyer fuarını SINIRSIZ knapsack + çanta boşaltma ile çözer — aynı stant tekrar tekrar alınır, çanta dolunca eve gidip boşaltılır, zaman hep ileri aktığı için topolojik sırayı zaman verir, kb hücre × O(n) = O(nbk) pseudopolinom. Problem 3 Protein Parsing'i naif v1'den (string-karşılaştırma her adımda, O(|S|·|P|·k) iki dev çarpım) precompute v2'ye (hash + m üyelik tablosu) indirger; runtime'da çarpım değil TOPLAM görmek precomputation sinyalidir, $O(k·|P| + k^2·|S|)$. Problem 4 Lazy Egg Drop'u minimax DP ile çözer — yumurta adversarial, min içinde max; topolojik sıra harcanan kaynaktan değil belirsizliğin DARALMASINDAN gelir, $O(n^3k)$. Ortak tema — istenen runtime'ın şekli (toplam mı çarpım mı, hangi parametreler) çözümün yapısını ele verir — Solomon'un dört probleme de aynı kapıdan girmesini sağlar.

Şekil 36.2 iki knapsack varyantını yan yana motordan **gerçek** verilerle gösterir: solda Problem 1 (0/1, $O(n^2)$ polinom), sağda Problem 2 (sınırsız + çanta boşaltma, $O(nbk)$ pseudopolinom). Tek bir "kapasite içinde max değer" çekirdeği iki farklı kısıtla karşı karşıya gelir; figür her ikisini birden kapsar ve aşağıdaki iki problemin köprüsüdür.

İki knapsack: 0/1 (her nesne ≤ 1 kez, O(n²)) vs SINIRSIZ + boşaltma (zaman = topolojik sıra, O(nbk) pseudopolinom)



Şekil 36.2: İki knapsack yan yana — Problem 1 ve Problem 2 İMZA. Veri MOTORDAN (P1: P=[8,5,11,6,3] K=[2,1,3,2,1] bütçe 5 → 19 = bitmask; P2: C=[10,4] W=[2,1] t=[1,0] b=2 h=1 k=9 → 24 = eylem-DFS). SOL P1 — 0/1 knapsack (her nesne ≤ 1 kez): 5 nesne kartı (P fiyat üst, K eritme alt), seçilenler (11+5+3) amber yıldızlı; bütçe çubuğu 5 para → 3+1+1 dolu; x(i,j) recurrence kutusu (yap → j−1'e GEÇ / yapma → j−1, her iki dalda nesne tüketilir); sonuç 19 rozeti; '$(n+1)^2$ × O(1) = $O(n^2)$ POLİNOM'. SAĞ P2 — SINIRSIZ + çanta boşaltma: zaman çizgisi 9 dk + motor-optimal eylem dizisi (stant0→eve→stant0→eve→stant1, aynı stant tekrar); çanta doluluk basamak grafiği (b=2'ye dayanınca EVE GİT → b'ye sıfırlanır amber ok); 'zaman HEP İLERİ → topolojik sıra (Solomon 30:00)'; sonuç 24 rozeti; 'kb × O(n) = O(nbk) PSEUDOpolinom (k, b runtime'da)'.

36.2 Problem 1: Coin-Crafting — 0/1 Knapsack

İfade. Bir hırsızın n özdeş altın parası var; bunları eritip n nesneden bazılarını üretip alıcıya satacak. Nesne i: fiyat P_i, eritme sayısı K_i (üretmek için gereken para). Her nesneden **en fazla bir** tane (0/1). Toplam para bütçesiyle geliri maksimize et.

> **Yaklaşım — iki doğal parametre: hangi nesneyi yaptım + kaç para harcadım**
>
> İki doğal parametre var: hangi nesneyi yaptım (nesne indeksi) + kaç para harcadım (kalan bütçe). Bu klasik **0/1 knapsack**. Nesne sırası önemsiz olduğu için topolojik sırayı nesne indeksi verir; her nesne için yerel kaba kuvvet ikiye iner — yap ya da yapma. "Her nesneden en fazla bir tane" kısıtı recurrence'ın yapısına gömülür: her iki seçenekte de $j \to j-1$ ilerler, yani bir kez bakılan nesne tüketilir.

> *"the two variables here, when I make a new object, is what object did I make? And how many coins did I spend?"* — Solomon, 14:30

Şekil 36.2 (sol panel) bu 0/1 knapsack'i motordan **gerçek** verilerle gösterir: örnek fiyatlar $P = [8, 5, 11, 6, 3]$ ve eritmeler $K = [2, 1, 3, 2, 1]$, bütçe 5 para için optimal gelir 19 — nesneler $11 + 5 + 3$ seçilir (K toplamı $3 + 1 + 1 = 5$, tam bütçe). Bağımsız bitmask tanığı da aynı 19'u verir.

Çözüm. S: $x(i, j) = i$ para ve ilk j nesneyle elde edilebilen maksimum gelir. **R:** iki seçenek — j'yi yapma $\to x(i, j-1)$; j'yi yap (yalnız $i \geq K_j$ ise) $\to P_j + x(i - K_j, j-1)$; ikisinin max'ı. **T:** ikinci indeks (j) azalır. **B:** $x(0, j) = 0$ (para yok), $x(i, 0) = 0$ (nesne yok). **O:** $x(n, n)$.

```
def coin_craft(P, K, budget=None):                  # (n+1)² × O(1) = O(n²)
    n = len(P)
    budget = n if budget is None else budget        # bütçe = n özdeş para
    x = {(i, 0): 0 for i in range(budget + 1)}      # taban: nesne yok → 0
    for j in range(1, n + 1):
        for i in range(budget + 1):
            best = x[(i, j - 1)]                     # j'yi YAPMA → j-1
            if K[j - 1] <= i:                        # j'yi YAP (para yeterli)
                best = max(best, P[j - 1] + x[(i - K[j - 1], j - 1)])
            x[(i, j)] = best                         # her iki dalda da j → j-1
    return x[(budget, n)], x
```

Karmaşıklık. $(n+1)^2$ alt problem $\times$ $O(1)$ iş = $O(n^2)$.

> **Bonus — SRTBOT'u koda çevirmenin iki yolu**
>
> Solomon SRTBOT'u koda çevirmenin iki yolunu gösterdi: **memoization** (yukarıdan-aşağıya; "zaten hesapladıysam tabloya bak, döndür" — DP'nin sihri tek satırda) ve **bottom-up** (aşağıdan-yukarıya; j sütunlarını sırayla doldur, recursion yok). İkisi sabit-çarpan farkıyla eşdeğer; pratikte bottom-up (yığın yükü yok + netlik) tercih edilir. Yukarıdaki kod bottom-up sürümdür.

36.3 Problem 2: Tim the Beaver Kariyer Fuarı — Sınırsız Knapsack + Çanta Boşaltma

İfade. n stant; nesne i: havalılık C_i, ağırlık W_i, sıra-bekleme t_i; her sıraya girmek $+1$ dk. Çanta kapasitesi b; eve gidip çanta boşaltma h dk (sonra $+1$). Toplam süre k. Aynı nesneden **birden çok** alınabilir (sınırsız). k dakikada maksimum toplam havalılığı bul; hedef $O(nbk)$.

> **Yaklaşım — iki kısıt: süre ve ağırlık; topolojik sırayı zaman verir**
>
> İki kısıt var: süre ve çanta ağırlığı. **Kilit:** çanta boşaltılabildiği için ağırlık monotonik azalmaz — ama boşaltmak da süre harcar. Yine de **zaman hep ileri akar** → bir adım atan her seçenek ya işi bitirir ya kalan süreyi kesin azaltır; bu da topolojik sırayı zamandan doğurur (boşaltmak ağırlığı geri yükselttiğinde bile). Aynı stant tekrar tekrar alınabildiği için bu **sınırsız** knapsack'tir: nesne indeksinde ileri geçmek yerine her seçenekte tüm stantlar yeniden taranır.

> *"time always moves forward for Tim the Beaver… that's what's going to give us our topological order."* — Solomon, 30:00

Şekil 36.2 (sağ panel) bu sınırsız knapsack'i motordan **gerçek** verilerle gösterir: iki stant $(C, W, t) = (10, 2, 1)$ ve $(4, 1, 0)$, çanta $b = 2$, ev $h = 1$, süre $k = 9$ için optimal havalılık 24 — motor-optimal senaryo stant 0 → eve → stant 0 → eve → stant 1 ($10 + 10 + 4$, aynı stant tekrar), tam 9 dakikayı kullanır. Bağımsız eylem-dizisi DFS tanığı da 24 verir.

Çözüm. S: $x(i, j) = i$ dakika ve çantada j ağırlık-yeri kalmışken maksimum havalılık. **R:** üç tip seçeneğin max'ı — (1) pes et $\to$ 0; (2) her stant $\bar{m}$ için: $C_{\bar{m}} + x(i - t_{\bar{m}} - 1,\ j - W_{\bar{m}})$ (uygunsa: $i - t_{\bar{m}} - 1 \geq 0 \wedge j - W_{\bar{m}} \geq 0$); (3) eve git: $x(i - h - 1,\ b)$ (havalılık yok, ağırlık-yeri b'ye sıfırlanır; $i - h - 1 \geq 0$ ise). **T:** her seçenek ya biter ya süreyi azaltır. **B:** $x(0, j) = 0$. **O:** $x(k, b)$.

```python
def career_fair(C, W, t, b, h, k):              # kb × O(n) = O(nbk)
    n = len(C)
    x = {(0, j): 0 for j in range(b + 1)}        # taban: süre yok → 0
    for i in range(1, k + 1):                    # zaman artan (topolojik)
        for j in range(b + 1):
            best = 0                             # (1) pes et
            for m in range(n):                   # (2) stant m (SINIRSIZ tekrar)
                ii, jj = i - t[m] - 1, j - W[m]
                if ii >= 0 and jj >= 0:
                    best = max(best, C[m] + x[(ii, jj)])
            if i - h - 1 >= 0:                    # (3) eve git → yer b'ye SIFIRLANIR
                best = max(best, x[(i - h - 1, b)])
            x[(i, j)] = best
    return x[(k, b)], x
```

Karmaşıklık. kb alt problem $\times$ $O(n)$ iş (stantlar üzerinde döngü) = $O(nbk)$. Pseudopolinom: k ve b runtime'da görünür — girdi boyutunda değil, sayısal değerlerinde.

36.4 Problem 3: Protein Parsing — Precomputation ile Hızlandırma

İfade. ACTG'den oluşan DNA dizisi S; uzunluğu $\leq k$ olan marker listesi P. S'yi alt-dizilere böl (division); değer = P'de olan parça sayısı. Maksimum değeri bul; hedef $O(k|P| + k^2|S|)$.

> **Yaklaşım — toplam-terimli runtime precomputation sinyalidir**
>
> Önce naif DP kurulur, sonra runtime düzeltilir. Asıl ipucu istenen runtime'ın **şeklinde**: $O(k|P|+k^2|S|)$ bir **toplam**, klasik "alt problem × iş" formülünün verdiği **çarpım** değil. Çoğu DP bir tablo doldurur ve runtime bir çarpımdır; bir toplam görünce "DP dışında ön-işleme (precomputation) var" diye okumak gerekir. Burada ön-işleme, P marker'larını hash'lemek ve S'nin her aralığının markerlığını bir m tablosuna doldurmaktır; DP kısmı ondan sonra tabloyu $O(1)$ okur.

> *"I see two terms. Most of our dynamic programming things involve filling in a table, where you would expect there to be a product. So… maybe I have to do some pre-computation."* — Solomon, 56:30

Şekil 36.3 bu iki aşamayı motordan **gerçek** verilerle gösterir: $S = $ ACTGACTGA ve $P = \{$ACT, GA, TGA, CT$\}$ için maksimum değer 4 — optimal bölünme ACT | GA | CT | GA, dört markerla dokuz harfi tam döşer. Üç bağımsız tanık (naif v1, precompute v2, tüm-bölünmeler brute) birebir 4 verir.

Çözüm — v1 (naif, yavaş). S: $x(i) = S[i{:}]$ sonekinin maksimum değeri. **R:** $x(i) = \max(\ x(i+1)$ [karakteri atla, değer yok], her marker için: marker S'ye i'de uyuyorsa $1 + x(i + |\text{marker}|)\ )$. **T:** i artar. **B:** $x(|S|) = 0$. **O:** $x(0)$. Süre: $|S|$ alt problem $\times |P|$ marker $\times\ O(k)$ string-karşılaştırma = $O(|S| \cdot |P| \cdot k)$ — iki büyük sayının çarpımı, çok yavaş.

Çözüm — v2 (precomputation). $m[i, j] = 1$ eğer $S[i{:}j] \in P$, yoksa 0. (1) P'deki tüm string'leri hash'le $\to O(k|P|)$ (birinci terim). (2) m'i doldur: her i için $j = i + 1 \ldots i + k$ (uzunluk $\leq k$), $S[i{:}j]$'yi hash'le $O(k)$ + P-hash'te ara $\to O(k^2|S|)$ (ikinci terim). Revize **R :** $x(i) = \max_{j \in 1 \ldots k}(\ m[i, i + j] + x(i + j)\ )$. DP kısmı $O(|S| \cdot k)$ — diğer terimlerden küçük.

(İndeks köprüsü: prose, kaynaktaki gibi m'in ikinci indeksini bitiş konumu olarak kullanır — $m[i, j]$, parça $S[i{:}j]$. Aşağıdaki kod ve figür ise ikinci indeksi parça uzunluğu alır: kodun $m[i][j]$'si, prose'un $m[i, i+j]$'sine denktir.)

```
def protein_v2(S, P, kmax):                         # O(k·|P| + k²·|S|)
    n = len(S)
    pset = set(P)                                   # (1) P'yi hash'le — O(k·|P|)
    m = [[0] * (kmax + 1) for _ in range(n + 1)]  # (2) m[i][j] üyelik tablosu
    for i in range(n):                              #     — O(k²·|S|)
        for j in range(1, min(kmax, n - i) + 1):
            if S[i:i + j] in pset:
                m[i][j] = 1
    x = [0] * (n + 1)                               # (3) küçük DP — O(|S|·k)
    for i in range(n - 1, -1, -1):
        best = 0
        for j in range(1, min(kmax, n - i) + 1):
            best = max(best, m[i][j] + x[i + j])  # yalnız TABLOYU okur
        x[i] = best
    return x[0], m
```

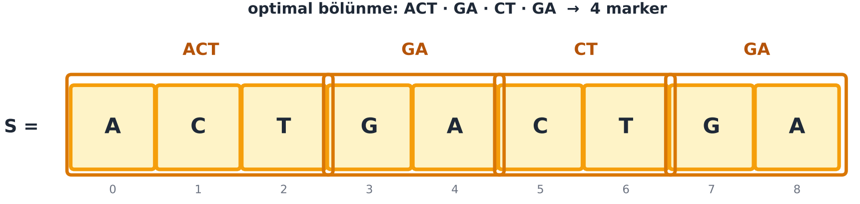
Karmaşıklık. $O(k|P| + k^2|S|)$. İlginç: tüm precomputation'dan sonra DP kısmı runtime'da **önemsiz** kalır.

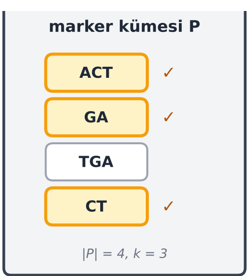
36.5 Problem 4: Lazy Egg Drop — Minimax DP

İfade. n katlı bina (kat yükseklikleri $h(i)$, sıralı), k yumurta. Yumurtanın kırılmadan atılabildiği **en yüksek katı** bul. Her atış maliyeti = inilen kat yüksekliği (merdiven inip kontrol). Yumurta kırılırsa üst sınır + bir yumurta gider; kırılmazsa geri alınır (yeniden kullanılır), alt sınır. **Toplam yüksekliği** (en kötü durumda) minimize et; hedef $O(n^3k)$.

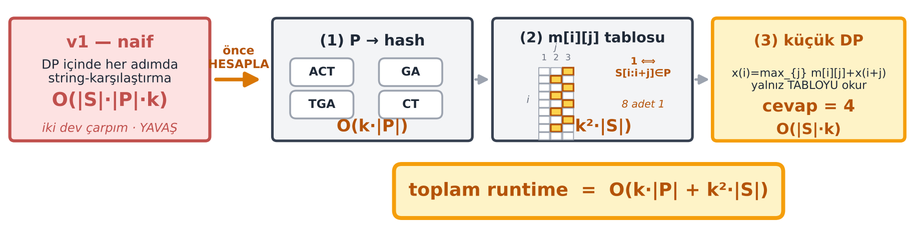
> Yaklaşım — adversarial yumurta: belirsizlik aralığını izle, min içinde max kur
>
> Bu bir deney tasarımı problemi. Belirsiz kat aralığı hep **bağlantılıdır** (bir kat kırılırsa üstündeki tüm katlar da kırılır), bu yüzden durumu bir aralık $[i, j]$ olarak izlemek yeterli. Yumurta **adversarial** kabul

Protein ayrıştırma: en çok markeri kapsayan bölünme





Naif tekrar-hesap vs PRECOMPUTE: runtime'da TOPLAM = sinyal



v1 = v2 = brute = 4

runtime'da TOPLAM görüyorsan precompute sinyali (Solomon 56:30)

Şekil 36.3: Protein parsing: naif tekrar-hesap vs PRECOMPUTE — Problem 3. Veri MOTORDAN (S=ACTGACTGA, P={ACT,GA,TGA,CT}, k=3, üç tanık v1==v2==brute==4, m tablosunda 8 adet 1). ÜST: DNA şeridi 9 harf kutu + optimal bölünme ACT|GA|CT|GA (4 amber marker bloğu) + sağda P marker listesi (kullanılanlar ✓). ALT: iki-aşamalı boru hattı — v1 naif (soluk kırmızı, O(|S|·|P|·k) iki dev çarpım YAVAŞ) → 'önce HESAPLA' oku → v2 üç precompute kutusu: (1) P → hash O(k·|P|), (2) m[i][j] üyelik tablosu mini görsel (1'ler amber) O(k^2·|S|), (3) küçük DP O(|S|·k); toplam runtime O(k·|P| + k^2·|S|) rozeti; 'runtime'da TOPLAM görüyorsan precompute sinyali (Solomon 56:30)'.

edilir: kırılır/kırılmaz seçimini en çok işi yaptıracak şekilde yapar, çünkü en kötü durumu sınırlamaya çalışıyoruz. Bu da recurrence'ı bir **minimax** yapar — biz hangi kattan atacağımızı seçerek maliyeti minimize ederiz, yumurta sonucu seçerek maliyeti maksimize eder (min içinde max).

> *"the egg might be an adversarial egg… we're trying to upper bound the amount of work… and I'm trying to design a procedure that minimizes my work."* — Solomon, 1:15:15

Şekil 36.4 bu minimax'ı motordan **gerçek** verilerle gösterir: $n = 6$ kat, $h[f] = f$, $k = 2$ yumurta için minimum en-kötü maliyet 14 — DP'nin ilk atışı kat $f = 2$, kırılırsa $[1, 1]$'e 1 yumurta, kırılmazsa $[3, 6]$'ya 2 yumurta düşer. Bağımsız politika-simülasyonu tanığı yedi gerçek eşiğin hepsinde doğru eşiği bulur ve en kötü maliyetini tam 14 (adversarial = en kötü gerçek eşik) verir. $k = 1$ tek-yumurta hâli zorunlu aşağıdan-yukarı tarama yapar ($\Sigma h = 21$); yumurta arttıkça maliyet $[21, 14, 12, 12]$ doymaya gider.

Çözüm. S: \$x(i, j, e) = \$ belirsiz katlar $[i, j]$ ve e yumurta kalmışken minimum toplam yükseklik. **R (minimax):** $x(i, j, e) = \min_{f \in [i,j]} [\, h(f) + \max(\, x(i, f-1, e-1)$ [kırıldı], $x(f+1, j, e)$ [kırılmadı])]\$. **T:** belirsizlik genişliği $j - i$ **kesin azalır** (yumurta harcaması değil!) → topolojik. **B:** $x(i, j, 0) = \infty$ (yumurta bitti ama kat belirsiz — min'de asla seçilmez); $x(i, i-1, e) = 0$ (aralık tek kata indi). **O:** $x(1, n, k)$.

```
def egg_drop(h, k):                                  # n²k × O(n) = O(n³k)
    n = len(h) - 1
    memo, choice = {}, {}

    def x(i, j, e):
        if j < i:    return 0                         # aralık çözüldü
        if e == 0:   return INF                       # yumurtasız belirsizlik
        if (i, j, e) in memo: return memo[(i, j, e)]
        best, bf = INF, None
        for f in range(i, j + 1):                     # hangi kattan atayım? (MIN)
            cand = h[f] + max(x(i, f - 1, e - 1),  # kırıldı  (yumurta -1)  ┐ adversarial
                              x(f + 1, j, e))      # kırılmadı (aralık üstü) ┘ (MAX)
            if cand < best:
                best, bf = cand, f
        memo[(i, j, e)] = best; choice[(i, j, e)] = bf
        return best

    return x(1, n, k), memo, choice
```

Karmaşıklık. n^2 (iki kat indeksi) $\times$ k (yumurta) alt problem $\times$ $O(n)$ (f döngüsü) = $O(n^3 k)$.

> **Atlanan 5. problem (duvar örme):** Karo dizerek duvar örme; runtime **üstel** ama problem buna izin verir — "küçük parametrelerde üstel" kabul, yasak olan **iki dev üstelin çarpımı**. Solomon zaman yetmediği için işlemedi (transkript 1:24:34).

Şekil 36.4: Lazy egg drop: minimax DP + politika simülasyonu — Problem 4 İMZA. Veri MOTORDAN (h=[0,1,2,3,4,5,6], k=2 → x(1,6,2)=14; politika-sim 7 eşik HEPSİ doğru + max maliyet 14; ilk atış f=choice[(1,6,2)]=2; k=1 → Σh=21; doyma [21,14,12,12]). SOL: 6 katlı bina, DP'nin ilk atışı f=2 amber vurgu; kırıldı dalı (amber, [1,1] e=1) ve kırılmadı dalı (slate, [3,6] e=2); minimax kutusu x(i,j,e)=min_f[h(f)+max(kırıldı,kırılmadı)] + 'yumurta ADVERSARIAL (Solomon 1:15:15)'; topolojik-sıra rozeti 'belirsizlik j−i DARALIR (yumurtadan değil)'. SAĞ ÜST: politika-simülasyon tablosu — 7 gerçek eşik t=0..6 için DP maliyeti [3,3,10,14,14,13,13], max=14 amber; 'DP politikası HER eşikte doğru eşiği bulur, en kötü == x(1,n,k)=14 adversarial tanık'. SAĞ ALT: doyma eğrisi k=1..4 → [21,14,12,12], 4. yumurta işe yaramaz (belirsizlik-daralması baskın).

36.6 Ne Öğrendik?

! Altı Taşınabilir Araç

Bu oturum, DP problem oturumlarının ikincisiydi ve SRTBOT çerçevesini dört pseudopolinom problemde uyguladı:

1. **0/1 vs sınırsız knapsack (P1 vs P2):** “her nesneden bir tane” ($j-1$’e geç) $\leftrightarrow$ “sınırsız” (aynı indekste kal / her seçeneği tara). İkisi de iki-parametreli (bütçe + nesne/zaman).
2. **Zaman = topolojik sıra (P2):** ağırlık geri artsa bile (çanta boşaltma), zaman hep ilerlediği için sıra garanti. “t hem zaman hem topological order”.
3. **SRTBOT → kod (P1):** memoization (top-down, “zaten hesapladıysam döndür”) vs bottom-up (tabloyu sırayla doldur); sabit-çarpan eşdeğer, genelde bottom-up tercih.
4. **Toplam-terimli runtime → precomputation (P3):** runtime’da çarpım değil **toplam** görürsen, DP dışında ön-işleme (hash + tablo) yap; DP kısmı küçülür.
5. **Minimax DP (P4):** adversarial sonuç (yumurta kırılır/kırılmaz) → min içinde max; topolojik sıra “harcanan kaynak”tan değil **belirsizlik daralmasından** gelebilir.
6. **“Hocayı teşhis et”:** istenen runtime’ın şekli (toplam mı çarpım mı, hangi parametreler) çözümün yapısını ele verir.

36.7 Sonraki

Ders 30 (PS10): Problem Oturumu 10 (Quiz 3 Tekrarı) — Jason Ku

Sırada **Ders 30 (PS10): Problem Oturumu 10 (Quiz 3 Tekrarı)** var — bu kez hoca değişir, Jason Ku ile DP ve karmaşıklık bloğunun kapanış tekrarını yaparız. Bu oturumdaki pseudopolinom + minimax sezgileri ve “runtime’ı teşhis et” refleksi, Quiz 3 hazırlığının çekirdeğidir.

37 Quiz 3 Gözden Geçirme

Dönemin son quiz tekrarı — SRTBOT derin tekrar ve üç gerçek sınav problemi (Lotto, DNA, Tapas)

ℹ Oturum bilgisi

- **Ku'nun videosu:** YouTube — Quiz 3 Review (≈84 dk)
- **OCW sayfası:** MIT 6.006 Quiz 3 Review
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 30 (PS10 / Quiz 3 Review)
- **Hoca:** Jason Ku
- **Okuma süresi:** ≈26 dk

  Bu **normal bir ders değil** — Quiz 3 öncesi **toplu tekrar** oturumudur. Quiz 3 = **DP bloğu**; Quiz 1-2 konuları (veri yapıları, çizge) kümülatif "fair game"dir ama doğrudan test edilmez.

37.1 Bu Quiz Review Ne Hakkında?

Bu, Jason Ku ile dönemin **son quiz tekrarıdır**: Quiz 3, tamamen **dinamik programlama (DP)** üzerine. Ku önce **SRTBOT recursive framework**'ünü adım adım tekrar eder, sonra **Spring '18'den üç gerçek sınav/ödev problemini** (Lotto, DNA babalık testi, Tapas) baştan sona çözer. Dördüncü problem (Gokemon Po) zaman yetmediği için bırakıldı.

> *"This is our last quiz review for the term, quiz 3… It's on dynamic programming… problem sets 7 and 8, and lectures 15 through 18."* — Ku, 0:19

⚠ Kapsam — kitap-numara remap

Ku, kapsamı orijinal MIT numaralandırmasıyla anar: "lectures 15 through 18 + problem sets 7 and 8". Bu kitaptaki karşılığı:

- **Ders 23-27 (L15-L18; araya Ders 25 = PS8 girer)** — DP temelleri (SRTBOT, LCS/LIS, Floyd-Warshall, pseudopolinom/subset sum).
- **DP problem oturumları Ders 25 (PS8) ve Ders 29 (PS9)** — Ku'nun andığı "PS 7-8", **orijinal MIT numaralandırmasıdır**.

Alıntılar birebir korunur (Ku "PS7-8" der); kapsam ifadeleri kitap-numaraya remap edilir.

Quiz 3, kursun **dinamik programlama sınavıdır**: alt problem tanımı, ilişki (recurrence), topolojik sıra, taban durum, çözüm kurma ve süre analizi. Veri yapıları Quiz 1'de, çizge algoritmaları Quiz 2'de kaldı; burada tek konu DP'dir.

- **Bu = üçüncü ve son sınav.** OMSCS CS 6515 (Graduate Algorithms) ekseninde DP, dersin **en ağır blokudur** — lisansüstü algoritma sınavlarının yarısı DP üzerinedir. Burada kurulan **SRTBOT disiplini** (önce alt problemi sözle tam tanımla, sonra recurrence'ı matematikle yaz) doğrudan oraya taşınır.
- **Durum genişletme refleksi.** Naif suffix DP kısıtı uygulayamadığında reflekste **state ekle** (Lotto: offset; Tapas: tatlı sayacı) — bu, "alt probleme bir soru sor" disiplininin en sık sınav uygulamasıdır.
- **Pseudopolinom teşhisi.** Hedefte **çıplak sayı** görünce "gerçek problem"den ayırt et: **pseudopolinom** teşhistir (subset sum / knapsack imzası).

Ders 30: Quiz 3 Review
SRTBOT çerçevesi
S·R·T·B·O·T derin tekrar
Durum genişletme (Lotto)
suffix DP · geçmişi gelecek kısıtı olarak hatırla · O(n)
Çoklu-dizi Boolean DP (DNA)
üç dizi · bağımlı indeks ∨ 4 seçim · $O(n^4)$
Pseudopolinom knapsack (Tapas)
0/1 knapsack + tatlı sayacı · −∞ taban · O(nks)
Sınav stratejisi
söz + matematik · çarp → topla · polinom teşhisi · taban dengesi

Şekil 37.1: Ders 30'un (Quiz 3 Review) kavram haritası; kök = Quiz 3 Review. Beş dal = (1) SRTBOT çerçevesi: Subproblem, Relation, Topological order, Base case, Original problem, Time; her özyinelemeli algoritmanın iskeleti, alt problemler örtüştüğünde memoize ile DAG'a çöker. (2) durum genişletme (Lotto): suffix DP, geçmişi gelecek kısıtı olarak hatırla, offset state, lineer zaman. (3) çoklu-dizi Boolean DP (DNA): üç dizi, bağımlı indeks elenir, dört seçimli OR, kuartik zaman. (4) pseudopolinom knapsack (Tapas): sıfır-bir knapsack artı tam-s tatlı sayacı, eksi sonsuz taban dengesi, k çıplak sayı olduğu için pseudopolinom. (5) sınav stratejisi: söz artı matematik, çarp toplama, polinom teşhisi, taban dengesi, parent pointer. Sonuç: Quiz 3 = SRTBOT disiplini sınavı.

37.2 SRTBOT Çerçevesi — Derin Tekrar

DP, **özyinelemeli (recursive) çerçevenin** özel hâlidir: alt problemler **birden çok kez** kullanılınca, çakışan alt problemleri **bir kez** hesaplayıp memoize ederiz. Özyineleme ağacı, aynı-değerli düğümler birleşince bir **DAG**'a çöker; DP tam da alt problemlerin örtüştüğü durumdur.

S — Subproblems (Alt problemler). x değişkeniyle alt problemleri tanımla: memonun **ne döndürdüğü** ve **ne kadar büyük** olduğu. **Kritik kural:** tanımında görünmeyen parametre kullanırsan yanlış yaparsın.

> *"if you have parameters in your subproblem that don't appear in your subproblem definition, you're doing it wrong."* — Ku (Subproblems)

Sıra (sequence) problemlerinde yaygın seçim: **prefix / suffix** (tek uçtan karar) veya **bitişik alt-dizi** (iki uçtan/ortadan karar). Bu sınıfta **suffix** kullanıyoruz (soldan-sağa okumak kolay); prefix ile birebir aynı — "aynı madalyonun iki yüzü". Çoklu girdide indeksleri **çarpar** (her diziden bir prefix/suffix) ve ek **durum** tutarız (max mı min mi? sıra kimde? çantada ne kadar yer kaldı?).

R — Relate (Bağıntı). Alt problemleri **matematiksel ifadeyle** ilişkilendir (kesinlik için söz değil, formül). Genelde $x(\dots) = \max / \min / \sum / \vee / \wedge$ (bir dizi seçim). Strateji: alt problem hakkında bir **soru** sor ("ilk karakterle ne yapayım?"); cevabı bilseydim daha küçük alt probleme inerdim → **polinom sayıda alt problem** olduğu için o sorunun tüm cevaplarını lokal **kaba kuvvetle** dene.

> *"it's really important that you write this in math, because it needs to be precise."* — Ku (Recursion)

T — Topological order (Topolojik sıra). "Daha küçük"ün ne demek olduğunu tanımla: bir indeks/parametre hep **artar veya azalır** (bazen indekslerin **toplamı**). Bağıntı acyclic ise alt problem grafı DAG'dır → bottom-up hesaplanır, sonsuz döngü olmaz.

B — Base cases (Taban durumlar). Özyinelemenin memo sınırının **dışına** taştığı her yer için sabit-zamanlı cevap. **Taban durum yoksa algoritma sonlu zamanlı bile değildir.**

> *"if you write code without a base case, it's going to be wrong."* — Ku (Base Cases)

O — Original problem (Özgün problem). Özgün cevabı alt problem(ler)den üret — çoğu zaman tek bir alt problem, bazen hepsinin max'ı (LIS, max subarray). Tam çözüm dizisini geri getirmek için **parent pointer** sakla (en kısa yoldaki gibi geri yürü).

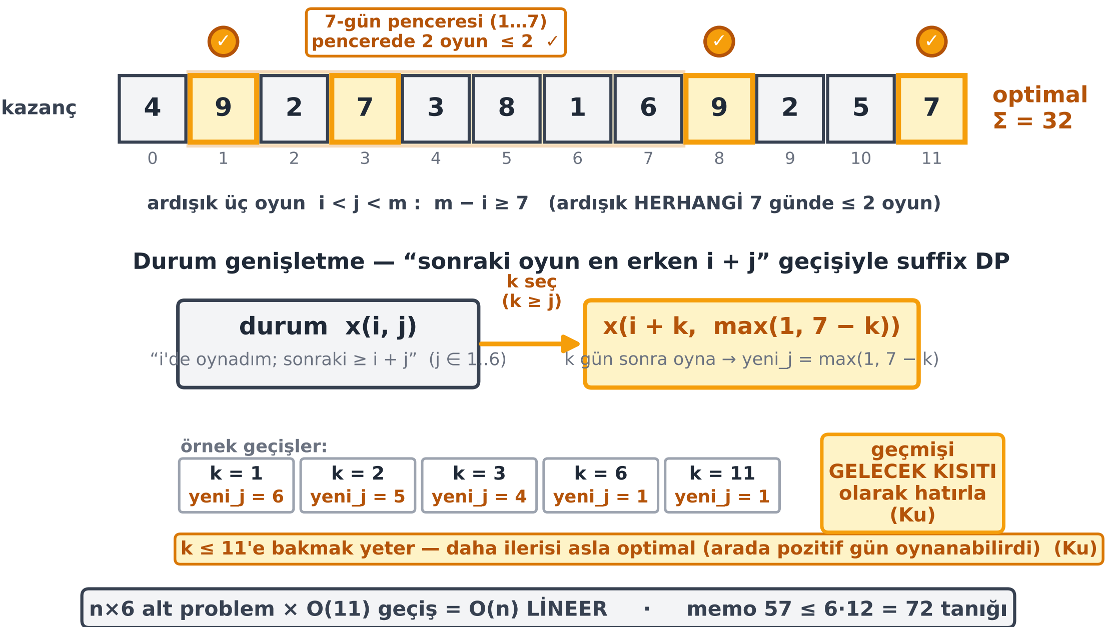
T — Time (Süre). Genelde alt problem başına işin toplamı; iş her alt problemde aynı sınırlıysa **çarp**. Alt problem sayısı = her parametrenin değer sayısının **çarpımı** (toplamı DEĞİL — her biri bağımsız seçilir). İş/alt problem = bağıntıda üzerinde optimize edilen seçim sayısı (branching).

> *"I multiply those numbers together. A lot of people will maybe say, oh, I add them together. No."*
> — Ku (Running Time)

37.3 Quiz-tarzı Problemler (Spring ’18, Tam Çözüm)

Aşağıda üç gerçek Spring ’18 problemi var; her birinin çözümünü açmadan önce kendin dene. Üçü, SRT-BOT’un üç farklı tekniğini sergiler: **durum genişletme** (Lotto), **çoklu-dizi Boolean DP** (DNA), **0/1 knapsack + sayaç** (Tapas). Tüm sayılar QR3 motoruyla doğrulanmıştır.

Şekil 37.2 Problem 1’in imza fikrini gösterir: suffix DP’de naif kısıt uygulanamayınca **durum genişletme** ile “geçmişi gelecek kısıtı olarak hatırla” — sonraki izinli oyun offset’i j state’e eklenir, k döngüsü sabit kalır ve algoritma **lineer** olur.



Şekil 37.2: Lotto — durum genişletme, suffix DP, O(n) lineer (QR3 Problem 1, İMZA figür, Spring-18). 12-gün kazanç şeridi L = [4,9,2,7,3,8,1,6,9,2,5,7]; ardışık HERHANGİ 7 günde en fazla 2 oyun (eşdeğer kısıt: ardışık üç oyun i<j<m için m − i ≥ 7). ÜST panel: optimal oyun günleri amber halka + ✓ (motordan brute-bitmask: günler 1,3,8,11 — kazançlar 9,7,9,7); örnek 7-gün penceresi amber bant, pencerede ≤ 2 oyun. ALT panel: durum x(i, j) = ‘i’de oynadım; sonraki ≥ i+j’ (j ∈ 1..6); geçiş x(i+k, max(1, 7−k)); k ≤ 11’e bakmak yeter (daha ilerisi asla optimal — arada pozitif gün oynanabilirdi). Motordan: lotto(L) = 32 == bitmask brute; memo 57 ≤ 6·12 = 72 (durum genişletme ×6 tanığı). n×6 alt problem × O(11) = O(n) LİNEER.

Problem 1 — Lotto (Tiffany Bannen): suffix DP + durum genişletme, O(n) lineer

İfade. n gün, gün i kazancı $L(i)$ (pozitif tam sayı). **Ardışık herhangi 7 günde en fazla 2 kez** loto oyna. Toplam kazancı maksimize et. **Lineer zaman** istenir.
Naif deneme. $x(i) = i \ldots n$ günlerinde max kazanç; R: oyna $L(i) + x(i+1)$ ya da oynama $x(i+1)$. Sorun: kazançlar pozitif → hep oyna; **7-gün kısıtı uygulanmıyor.**
Düzeltme — durum genişletme. Gün i'de **oynadığımı varsay** (LIS gibi) ve **bir sonraki izinli oyun** offset'i j'yi state'e ekle: $x(i,j)$ = "i'de oynadım, sonraki izinli oyun $i+j$" ($j \in 1..6$, çünkü asla $i + 6$'dan ileri kısıtlanamam — daha eski oyunlar daha sağa itemez).

> *"I'm remembering what happened in the past by describing it as a restriction of something in the future."* — Ku

Çözüm. S: yukarıdaki $x(i,j)$. **R:** $x(i) = L(i) + \max_k\ x(i+k, \text{yeni_j})$, yeni_j $= \max(1, 7-k)$; $k =$ bir sonraki oynama günü offset'i. **T:** i kesin artar (suffix). **B:** $x(n) = L(n)$ (son gün $L(n)$ kullanılır); $L(i)$'yi max dışına alıp $\cup\ \{0\}$ yazılırsa bağıntı tabana indirgenir. **O:** $\max_i\ x(i,1)$ (ilk ~sabit).
Karmaşıklık — anahtar. k döngüsü neden sabit? İki oyun "ortada" sıkışınca: 10 ara eleman → en fazla **11**'e kadar bak. Bundan ileri oynamak asla optimal değil (arada pozitif kazançlı bir gün oynanabilirdi). Yani n alt problem $×, O(11) = $ $O(n)$ **lineer**.
Kategori (Demaine L18): suffix alt problem + lokal durum genişletme; sabit ama > 2 branching; özgün cevapta alt problemleri birleştirme. "Kavramsal olarak en kolay, implement etmesi en zor" tür.

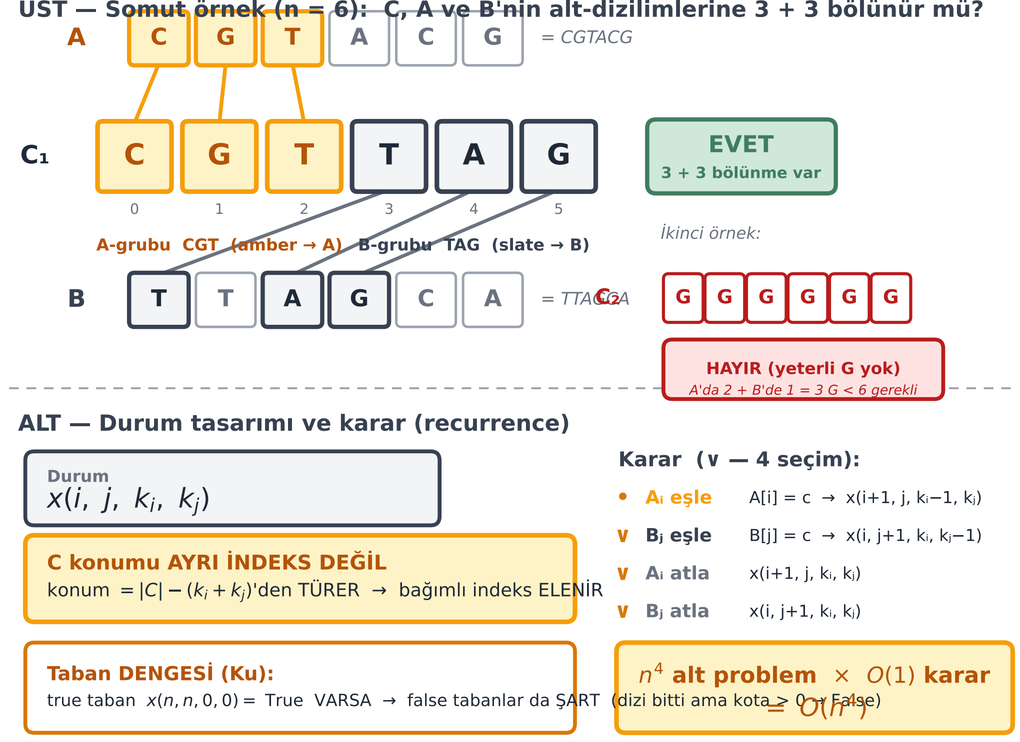
İndeks köprüsü — Lotto prose vs kod

Prosede taban "son gün $L(n)$" diye 1-tabanlı anlatılır; motorda (`lotto`) diziler 0-tabanlıdır, dolayısıyla son gün indeksi $n-1$ ve k döngüsü `range(j, min(11, n-1-i)+1)` ile sınırlanır. Aynı matematik (D25/D29 emsali): 1-tabanlı sınav anlatımı ile 0-tabanlı kod, -1 kaymasıyla birebir örtüşür; motor `lotto(L) = 32` değeri her iki okumayı da doğrular.

Şekil 37.3 Problem 2'nin imza fikrini gösterir: üç dizi + C'nin tam bölünmesi; C'deki konum **ayrı bir indeks değildir** ($k_i + k_j$'den türer, bağımlı indeks elenir) ve karar dört seçimli bir **OR**'dur.

Problem 2 — DNA Babalık Testi (Alice/Bob/Charlie): 3-string Boolean LCS, O(n^4) kuartik

İfade. Üç n-uzunluklu DNA dizisi A, B, C (CGTA). Charlie (C), **eşit uzunlukta iki (bitişik olmayan) alt-diziye** bölünebiliyorsa eşleşir: biri A'nın alt-dizisi, diğeri B'nin alt-dizisi. C'nin **tüm** karakterleri kullanılmalı (A, B'ninkiler değil). Charlie sahte mi? $O(n^4)$ istenir.
Yaklaşım. LCS'e benzer ama **üç** dizi + C'nin tam bölünmesi. Naif (i, j, k prefix indeksleri) yetmez — C'nin kaçını A'ya, kaçını B'ye verdiğimi bilmem gerek. **Durum:** $k_i = C$'nin A'ya eşleşecek kalan karakter sayısı, $k_j = B$'ye eşleşecek kalan. C'deki konum **bağımsız değil** — $k_i + k_j$'den hesaplanır (C suffix'inde $k_i + k_j$ karakter kalmalı).
Çözüm. S: $x(i, j, k_i, k_j) = $ Boolean: A suffix(i)'den k_i-uzunlukta + B suffix(j)'den k_j-uzunlukta alt-diziyle, C suffix'inin (uzunluk $k_i + k_j$) **tüm** karakterleri eşleşebilir mi? **R** ($\vee$ **over 4 seçim):** (1) $A_i = $ ilgili C karakteri ise & $k_i > 0$: $x(i+1, j, k_i-1, k_j)$; (2) B_j eşleşir & $k_j > 0$: $x(i, j+1, k_i, k_j-1)$; (3) A_i'yi atla: $x(i+1, j, k_i, k_j)$ ($i < n$); (4) B_j'yi atla: $x(i, j+1, k_i, k_j)$ ($j < n$). **T:** $i + j$ kesin artar (en az biri artar). **B:** $x(n, n, 0, 0) = $ **true** (her şey eşleşti); A bitti ama $k_i > 0$ → **false** (ve B için simetrik). **O:** $x(0, 0, n/2, n/2)$; n çift değilse direkt false.

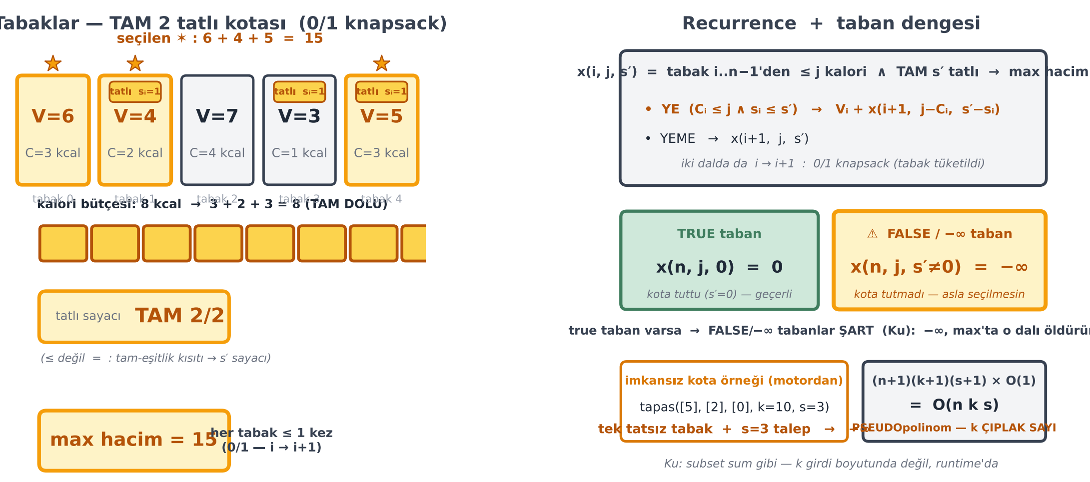


Şekil 37.3: DNA babalık testi — 3-string Boolean DP, $O(n^4)$ kuartik (QR3 Problem 2, Spring-18). Üç n-uzunluklu DNA dizisi; C, eşit uzunlukta iki ayrık alt-dizilime bölünüp biri A'nın diğeri B'nin alt-dizilimi olabilir mi? ÜST panel (somut örnek n=6): C_1=CGTTAG → EVET, 3+3 bölünme (A-grubu CGT amber → A=CGTACG; B-grubu TAG slate → B=TTAGCA); ikinci örnek C_2=GGGGGG → HAYIR (A'da 2 + B'de 1 = 3 G < 6 gerekli). ALT panel (durum/recurrence): $x(i, j, k_i, k_j)$; C konumu AYRI İNDEKS DEĞİL, $|C| - (k_i+k_j)$'den TÜRER → bağımlı indeks elenir; karar ∨ 4 seçim (A_i eşle / B_j eşle / A_i atla / B_j atla); taban dengesi: true taban x(n,n,0,0)=True VARSA false tabanlar da ŞART (dizi bitti ama kota>0 → False). Motordan: dna_match(A,B,C_1)=True == brute (2^n boyama + is_subsequence D24); C_1 boyaması A→[0,1,2], B→[3,4,5]; tek-uzunluk → False. n^4 alt problem × $O(1) = O(n^4)$.

“If you give us a true base case, you better be giving us some false base cases.” — Ku (Base Cases)

Karmaşıklık. $i, j \in 0..n$; $k_i, k_j \in 0..n/2 \to n^4$ alt problem $\times$ $O(1)$ (4 seçim) = $O(n^4)$ **kuartik**. (Gerçek babalık değil — kurgu.)

Şekil 37.4 Problem 3’ün imza fikrini gösterir: çekirdek 0/1 knapsack’ine **TAM-s tatlı** sayacı eklenir; bu tam-eşitlik kısıtı tabanı ikiye böler ve “kota tutmadı → $-\infty$” dengesi $-\infty$ tabanlarını zorunlu kılar. k çıplak sayı olduğu için runtime **pseudopolinom**.



Şekil 37.4: Tapas — 0/1 knapsack + tatlı sayısı, O(nks) pseudopolinom (QR3 Problem 3, İMZA figür, Spring-18). n tabak; tabak i: hacim V_i, kalori C_i, tatlılık $s_i \in \{0,1\}$. En fazla k kalori ye; TAM s tatlı tabak ye; 0/1; doyma için hacmi maksimize et. SOL panel: 5 tabak kartı (V hacim üst, C kalori alt, tatlı rozeti); seçilenler (0,1,4) amber ; kalori çubuğu 8’e karşı 3+2+3=8 TAM dolu; tatlı sayacı TAM 2/2; max hacim 15. SAĞ panel: x(i, j, s) iki seçim (ye / yeme); taban dengesi — true taban x(n,j,0)=0 (yeşilimsi) AMA x(n,j,s ≠0)=−∞ (amber uyarı: kota tutmadı → asla seçilme); imkansiz mini-örnek tapas([5],[2],[0],10,3)=−∞ (tek tatsız tabak + s=3); polinom-teşhis (n+1)(k+1)(s+1) × O(1) = O(nks) PSEUDOpolinom (k çıplak sayı — Ku: subset sum gibi). Motordan: tapas(…) = 15 = brute (brute; optima) {0,1,4}, V=[6,4,5] → 15, ΣC=8≤8, Σs=2

ℹ Problem 3 — Tapas (Obert Ratkins): 0/1 knapsack + tatlı sayısı, O(nks) pseudopolinom

İfade. n tabak; tabak i: hacim V_i, kalori C_i, tatlılık $s_i \in \{0, 1\}$. En fazla k **kalori** ye; **tam** s **tatlı tabak** ye; aynı tabağı **iki kez alma** (0/1). Doyma için **hacmi maksimize et**. $O(nks)$ istenir.
Önce: polinom mu? Girdi = her tabak için 3 sayı, n tabak → n, s polinom; ama k **sadece bir sayı** (bir kelimede, üstel büyüklükte olabilir). Yani **pseudopolinom** (subset sum / knapsack gibi). Sınavda sık sorulur: “runtime polinom mu?” — çözmeden, runtime’a bakıp girdi boyutuyla sınırlanıp sınırlanmadığına bak.

> *"this is a pseudopolynomial running time. Because k is just a number in my problem, similar to subset sum."* — Ku

Çözüm. S: $x(i, j, s') = P_i \dots P_n$ tabaklarından, en fazla j kalori ve **tam** s' tatlı ile elde edilebilen max hacim. **R (max, 2 seçim):** ye → $V_i + x(i+1, j-C_i, s'-s_i)$ (yalnız $C_i \leq j \wedge s_i \leq s'$); yeme → $x(i+1, j, s')$ (hep). **T:** i kesin artar. **B:** $x(n+1, j, 0) = 0$ (menü bitti, tam s yendi — iyi); $x(n+1, j, s') = $ $-\infty$ $s' \neq 0$ ise (tatlı kotası tutmadı — asla seçilmesin). **O:** $x(1, k, s)$.
Karmaşıklık. $(n+1)(k+1)(s+1)$ alt problem $×, O(1) = $ $O(nks)$ **pseudopolinom**.

> **İndeks köprüsü — Tapas prose vs kod**
>
> Prosede taban "$x(n+1, j, 0)$" ve özgün cevap "$x(1, k, s)$" diye 1-tabanlı yazılır; motorda (`tapas`) tabaklar 0-tabanlıdır, dolayısıyla taban `i == n` (tüm tabaklar tüketildi) ve özgün cevap `x(0, k, s)`'dir. Aynı recurrence (D25/D29 emsali): 1-tabanlı sınav anlatımı, kodda bir indeks kaymasıyla birebir aynı tabloyu üretir; motor `tapas(...) = 15` ve imkansız kotada -∞ her iki okumayı da doğrular.

Atlanan 4. problem — Gokemon Po

Canavar yakala; bir konuma git (ride-share ücreti) + bedava yakala, VEYA uygulama-içi satın al (farklı ücret, konum değişmez). Anahtar: **en son nerede olduğumu hatırla** → sonraki konuma mesafe. Pseudopolinom değil; konum durumuyla genişletme. Ku zaman yetmediği için bıraktı (transkript 1:09).

37.4 Quiz Hazırlığı Egzersizleri

Egzersiz 1. Lotto'yu **prefix** alt problemiyle yeniden kur (gün i'de oynadığımı varsay, önceki izinli oyun). Sonucun suffix versiyonuyla aynı $O(n)$ olduğunu göster.

Egzersiz 2. DNA probleminde C'deki konumu neden ayrı bir indeks olarak tutmaya gerek olmadığını ($k_i + k_j$'den türediğini) bir örnekle açıkla.

Egzersiz 3. Tapas'ta "tam s tatlı" kısıtını "en fazla s tatlı"ya çevirirsen taban durumlar nasıl değişir? Hâlâ $-\infty$ gerekir mi?

Egzersiz 4. Verilen bir runtime'ın (örn. $O(nW)$, $O(n^2)$, $O(2^n \cdot n)$) polinom mu pseudopolinom mu üstel mi olduğunu, girdi boyutuna göre sınıflandır.

Egzersiz 5. Üç problemin her birinde "alt problem hakkındaki soru"yu (R adımı) tek cümleyle yaz (Lotto: sonraki ne zaman oynayayım? DNA: ilk karakteri kime eşleyeyim? Tapas: bu tabağı yiyeyim mi?).

37.5 Sınav Stratejisi (Kapsam Notları)

- **Söz + matematik:** S'yi sözle (memo ne döndürür), R'yi matematik formülle yaz — iletişim puanın yarısı.
- **Prefix** ↔ **suffix** değiştirilebilir; diziyi ters çevir, aynı DP. Bu sınıf suffix kullanır.
- **Alt problem sayısı = parametrelerin çarpımı** (toplamı değil). İş/alt problem = branching.

- **Polinom teşhisi:** runtime'daki her terimi girdi boyutuyla sınırlayabiliyor musun? k/W gibi "çıplak sayı" varsa **pseudopolinom**.
- **Parent pointer:** sadece optimum değeri değil, **seçimi** (hangi gün/tabak/eşleşme) geri getirmek için max'ta hangi alt probleme gittiğini sakla.
- **Taban durum dengesi:** $\vee$-temelli Boolean DP'de bir **true** taban varsa, mutlaka **false** tabanlar da olmalı (yoksa hep true döner).

Builder Notu — Pseudopolinom Teşhisi Gerçek-Dünya Refleksi

"Runtime'da çıplak sayı var mı?" sorusu sınıf dışında her yerde işe yarar: bir DP çözümünün $O(nW)$ olması, W büyüdükçe (yüksek bütçe, büyük kapasite) pratikte yavaşlayacağı anlamına gelir — subset sum, knapsack, coin change hepsi bu sınıftandır. Mühendislikte refleks: bir runtime gördüğünde "girdi boyutu kaç bit?" diye sor; k bir kelimede sığan bir sayıysa ama runtime k ile çarpılıyorsa, girdi log k bit iken çalışma $2^{\log k}$ ile büyür — **pseudopolinom**. Quiz 3'ün Tapas problemi tam olarak bu refleksi test eder.

37.6 Toplu Cheat Sheet — SRTBOT

Adım	Ne yapılır	Quiz'de dikkat
S — Subproblems	$x(\dots)$ tanımı: memo ne döndürür, ne kadar büyük	Tanımda olmayan parametre = sıfır puan
R — Relate	Matematik bağıntı: max / min / $\sum$ /$\vee$ over seçimler	Söz değil formül; "soru sor" stratejisi
T — Topological	Bir indeks (veya toplam) hep artar/azalır $\to$ DAG	Acyclic kanıtı yoksa sonsuz döngü
B — Base cases	Memo sınırı dışı için sabit-zaman cevap	Taban yoksa sonlu zaman bile değil
O — Original	Özgün cevabı alt problemden üret (+ parent pointer)	Tek alt problem mi, max mı?
T — Time	#alt problem (parametre çarpımı) $\times$ iş/alt problem	Çarp, toplama; polinom/pseudopolinom ayrımı

DP alt problem kategorileri (Demaine L18): suffix/prefix (tek-uç karar) · bitişik alt-dizi (iki-uç karar) · çoklu dizi (indeks çarpımı) · durum genişletme (sıra/yön/bütçe/tatlı/konum) · sayıya göre genişletme (pseudopolinom).

37.7 Bu Quiz Review'in Özeti

1. **Quiz 3 = DP bloğu** (Ders 23-27; L15-L18); tek konu dinamik programlama, omurga **SRTBOT**.
2. **SRTBOT çerçevesi**: alt problemi sözle tam tanımla, recurrence'ı matematikle yaz, topolojik sırayı kanıtla, taban durumları (true + false dengesi) kur, özgün cevabı üret, süreyi analiz et.
3. **Lotto (durum genişletme)**: naif suffix kısıt uygulayamayınca offset state ekle; "geçmişi gelecek kısıtı olarak hatırla"; $O(n)$ lineer.

4. **DNA (çoklu-dizi Boolean DP)**: üç dizi; bağımlı indeks (C konumu = $k_i + k_j$) elenir; ∨ 4 seçim; true/false taban dengesi; $O(n^4)$.
5. **Tapas (pseudopolinom knapsack)**: 0/1 knapsack + tam-s tatlı sayacı; kota tutmadı → $-\infty$ taban; k çıplak sayı → $O(nks)$ pseudopolinom.
6. **Strateji**: söz + matematik, çarp (toplama değil), polinom teşhisi, parent pointer, taban dengesi.

! Tek Bir Cümle

Quiz 3'te kritik olan: alt problemi **sözle** tam tanımla (tanımsız parametre = puan yok), bağıntıyı **matematikle** yaz, topolojik sırayı kanıtla, taban durumları (true + false dengesi) unutma, runtime'ı **girdi boyutuyla** sınıflandır (polinom mu pseudopolinom mu). Üç gerçek problem üç tekniği gösterdi: **durum genişletme** (Lotto — "geçmişi gelecek kısıtı olarak hatırla"), **çoklu-dizi Boolean DP** (DNA — bağımlı indeks elenir), **0/1 knapsack + sayaç** (Tapas — pseudopolinom).

37.8 Builder ve OMSCS Bağlantıları

5 köprü

Bu son tekrar oturumu, "alt problemi sözle tanımla, recurrence'ı matematikle yaz, runtime'ı girdi boyutuyla sınıflandır" disiplinini kurar — köprülerin özeti:

1. **Quiz 3 = DP midterm** → OMSCS CS 6515: dinamik programlama, graduate algoritma dersinin **en ağır blokudur** (sınavların yarısı DP).
2. **Durum genişletme** → gerçek sistemde "state machine + bütçe" tasarımı: kalan kapasite, kalan adım, son konum — hepsi DP state'i olur.
3. **Çoklu-dizi DP** → biyoinformatik (dizi hizalama), diff/merge araçları, versiyon kontrol — LCS ailesinin pratik karşılıkları.
4. **Pseudopolinom teşhisi** → "runtime'da çıplak sayı var mı?" refleksi: knapsack, subset sum, coin change'in pratik yavaşlık sınırı.
5. **SRTBOT iletişim disiplini** → mühendislikte "önce problemi net tanımla, sonra çöz"; kod yazmadan önce alt problemi sözle ifade etmek, gerçek refleks.

! Tek bir şey alıp gideceksen

SRTBOT, **her** özyinelemeli algoritmanın çerçevesidir; DP, alt problemler **örtüştüğünde** (memoize → DAG) devreye girer. Quiz 3'te kritik olan: alt problemi **sözle** tam tanımla (tanımsız parametre = puan yok), bağıntıyı **matematikle** yaz, topolojik sırayı kanıtla, taban durumları (true + false dengesi) unutma, runtime'ı **girdi boyutuyla** sınıflandır (polinom mu pseudopolinom mu). Üç gerçek problem üç tekniği gösterdi: durum genişletme (Lotto), çoklu-dizi Boolean DP (DNA), 0/1 knapsack + sayaç (Tapas).

38 Toparlanma ve Sonraki Dersler

Dönem retrospektifi — dört hedef, üç ünitenin tek omurgası ve 6.046 köprüsü

ℹ Oturum bilgisi

- **Ku'nun videosu:** YouTube — Lecture 20: Course Review (≈38 dk — normal dersten kısa)
- **OCW sayfası:** MIT 6.006 Lecture 20
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 31 (L20)
- **Hoca:** Jason Ku (dönem retrospektifi; sondan bir önceki ders)
- **Okuma süresi:** ≈24 dk

  SONDAN BİR ÖNCEKİ ders — test edilebilir materyal bitti; bu ders **yüksek-seviye sentez**. Önceki ders (Ders 30 / Quiz 3 Review, Ku) DP'yi gözden geçirdi; bu retrospektif tüm dönemi tek bir omurgada toplar ve 6.046'ya köprü kurar. Son ders (Ders 32 / L21) dönemi kapatır.

38.1 Bu Derste Ne Var?

Sondan bir önceki ders (**Jason Ku**): **dönem retrospektifi** + "buradan nereye?". Test edilebilir tüm materyal bitti; bu ders öğrendiklerimizi bağlama oturtur ve sonraki teori derslerini tanıtır — **yüksek-seviye sentez**.

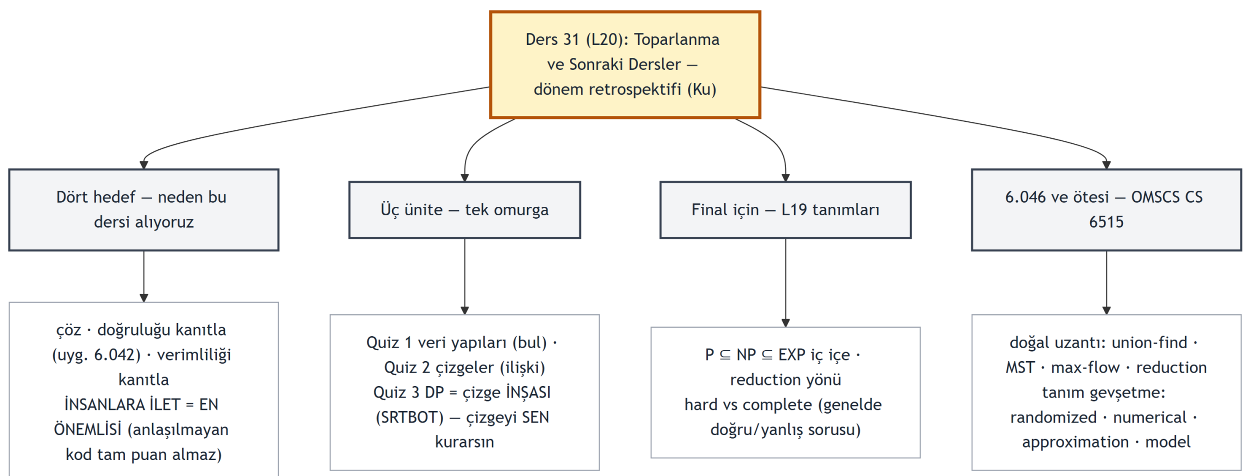
> *"Welcome, everybody, to the second-to-last lecture of 6.006."* — Ku, 0:18

Üç ana parça: (1) **6.006'nın 4 hedefi** (neden bu dersi alıyoruz), (2) **3 ünitenin tek bir hikâye olarak** özeti (veri yapıları → çizge → DP), (3) **6.046 ve ötesi** (karmaşıklığı/modeli gevşeten teori dalları).

38.2 1. 6.006'nın 4 Hedefi

Ders dört (aslında 3+1) hedef üzerine kuruluydu:

1. **Zor hesaplama problemlerini çöz** — algoritma üret (6.0001/6.009'daki gibi "bilgisayara çözdüğünü göster").
2. **Doğruluğu kanıtla** — her geçerli girdi için doğru çıktı; sonsuz girdi uzayı → özyineleme + tümevarım. "Bu ders aslında **uygulamalı 6.042**'dir."
3. **Verimliliği kanıtla** — "iyi" ne demek? **Hesaplama modeli** (word-RAM) + asimptotik + **ölçeklenebilirlik** (performans, girdi büyüme oranına göre).



Şekil 38.1: Ders 31'in (L20) kavram haritası: kök = Toparlanma ve Sonraki Dersler (Ku) — dönem retrospektifi, yüksek-seviye sentez, test edilebilir materyal bitti. Dört dal — (1) Dört hedef: zor problemi çöz algoritma üret, doğruluğu kanıtla özyineleme artı tümevarım uygulamalı 6.042, verimliliği kanıtla word-RAM artı asimptotik artı ölçeklenebilirlik, insanlara ilet EN ÖNEMLİSİ doğru ama anlaşılmayan kod tam puan almaz. (2) Üç ünite tek omurga: Quiz 1 veri yapıları bir şey bul dizi vs küme artı sıralama heap counting radix; Quiz 2 çizgeler düğüm durum kenar geçiş SSSP genellik-runtime takası DAG BFS Dijkstra Bellman-Ford; Quiz 3 DP eşittir çizge inşası SRTBOT alt problem düğüm bağıntı kenar topolojik sıra DAG, çizgeyi SEN kurarsın yaratıcı kısım. (3) Final için L19 tanımlar: P alt küme NP alt küme EXP iç içe, reduction yönü bilinen-zoru hedefe indirge, hard vs complete. (4) 6.046 OMSCS CS 6515: doğal uzantı union-find MST max-flow gerçek reduction artı analiz çok daha zor; tanım gevşetme randomized Las Vegas Monte Carlo, numerical kesinliği zamanla öde, approximation sabit-çarpan yakın, model değişimi cache quantum parallel. Birleştirici tema: 6.006'nın amacı sadece algoritma yazmak değil doğruluğu ve verimliliği insanlara iletmek; üç ünite tek özyinelemeli hikâyedir bul ilişkilendir inşa et; 6.046 bu hikâyeyi derinleştirir ve doğru verimli tanımını gevşetir.

4. **İnsanlara iletişim** — en önemlisi. Kod doğru olsa bile ne yaptığı anlaşılmazsa tam puan yok.

> *"your algorithm might be correct or efficient, but you need to be able to communicate that to humans."* — Ku

Bilgisayar bilimi büyük ölçüde **başkalarıyla** çalışmaktır; ne yaptığını ve neden doğru/verimli olduğunu iletemezsen sınırlısın. Şekil 38.2 dört hedefi soldan sağa kademeli bir disiplin olarak dizer ve dördüncü kademeyi (iletişim) amber/büyük olarak öne çıkarır.

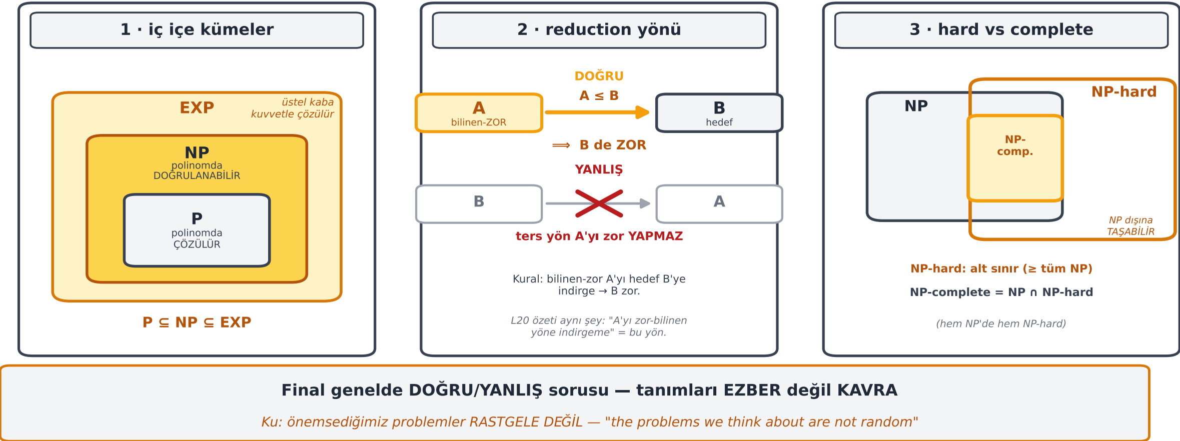
Şekil 38.2: 6.006'nın dört hedefi (Ku L20 §1): bir algoritmayı yalnız çözmek değil, doğruluğunu ve verimliliğini KANITLAYIP İNSANLARA İLETMEK. Soldan sağa DÖRT kademe (akış oklarıyla bağlı): (1) ZOR PROBLEMİ ÇÖZ — algoritma tasarla; (2) DOĞRULUĞU KANITLA — özyineleme + tümevarım 'uygulamalı 6.042'; (3) VERİMLİLİĞİ KANITLA — word-RAM + asimptotik + ölçeklenebilirlik; (4) İNSANLARA İLET — EN ÖNEMLİSİ (amber, büyük), Ku alıntısı 'your algorithm might be correct or efficient, but you need to be able to communicate that to humans'. Alt şerit: doğru ama anlaşılmayan kod TAM PUAN ALMAZ — bilgisayar bilimi büyük ölçüde BAŞKALARIYLA çalışmaktır. KAVRAMSAL figür (sayı yok).

38.3 2. Karmaşıklık — Final İçin (L19 Özeti)

L19 (Ders 28) hiçbir ödevde işlenmedi; final'de **tanımlar** test edilir. Mesaj: çoğu problem "iyi" (polinom) çözülemez — **ama** önemsediğimiz problemler **rastgele değildir**: ya **certificate ile polinom-zamanda doğrulanır** (NP) ya da **üstel kaba kuvvetle** çözülür.

> *"the problems we think about are not random."* — Ku

Final'de beklenenler: P/NP/EXP **iç içe geçişi** ($EXP \supseteq NP \supseteq P$), **reduction yönü** (A'yı zor-bilinen B'ye indirgersen A da zordur), **hard vs complete** ayrımı. Genelde doğru/yanlış sorusu. Şekil 38.3 bu üç tanımı yan yana üç mini-kartla toplar.



Şekil 38.3: Final hazırlığı — L19 tanımları (L20 §2): sınıflar · reduction yönü · hard vs complete. KART 1 'iç içe kümeler': EXP ⊇ NP ⊇ P iç içe bant (önemsediğimiz problemler rastgele değil — ya NP'de polinom-DOĞRULANABİLİR ya da üstel kaba kuvvetle ÇÖZÜLEBİLİR). KART 2 'reduction yönü': DOĞRU yön amber ok — bilinen-ZOR A ≤ hedef B ⟹ B ZOR; TERS yön kırmızı çarpı (A'yı zor YAPMAZ). KART 3 'hard vs complete': NP-hard = alt sınır (NP DIŞINA taşabilir); NP-complete = NP ∩ NP-hard. Alt şerit: tanımları EZBER değil KAVRA + Ku tanığı 'the problems we think about are not random'. KAVRAMSAL figür (sayı yok).

38.4 3. Üç Ünite — Quiz 1: Veri Yapıları (Kara Kutular)

Sabit-olmayan boyutta girdide **bir şey bulmak**. İki sorgu tipi: **dizi (extrinsic sıra)** vs **küme (intrinsic anahtar)**.

- **Dizi:** dinamik dizi (Python list — en yaygın; sona ekle/çıkar süper) ve sıra AVL (ortaya ekleme için).
- **Küme:** hash table (sözlük işlemleri $O(1)$ beklenen), küme AVL (dinamik **sıralı** işlemler: önceki/sonraki), sıralı dizi (statikse yeterli).

Sonra bu veri yapılarını **sıralama** için kullandık: öncelik kuyruğu sıralaması (dizi/sıralı dizi → n^2), **heap sort** ($n \log n$), AVL sort, ve **counting/radix** (direct access array → lineer, polinom-sınırlı sayılar).

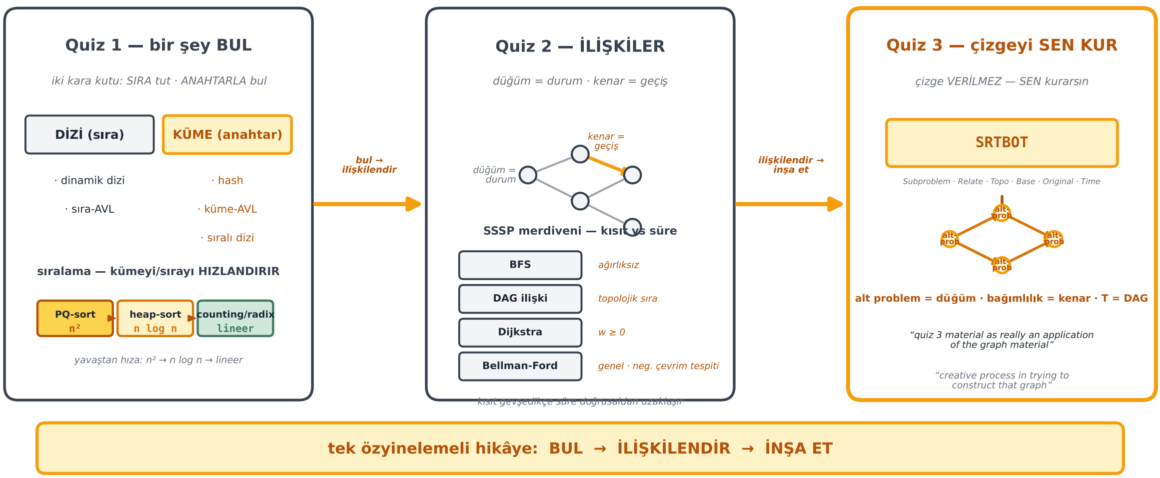
Bu ünite, sonraki ikisiyle birlikte **tek bir omurganın** ilk parçasıdır. Şekil 38.4 bu üç üniteyi tek özyinelemeli hikâye olarak (bul → ilişkilendir → inşa et) yan yana dizer; bu **İMZA figür** §3, §4 ve §5'i birden kapsar.

38.5 4. Üç Ünite — Quiz 2: Çizgeler

Çizge = elemanlar **arası ilişki**: düğüm = sistemin **durumu**, kenar = **geçiş**. Ayrık sistemleri (yol ağı, sıra-tabanlı oyun) modellemenin güçlü çerçevesi.

Odak: **tek-kaynak en kısa yollar** — genellik/runtime takasıyla birden çok algoritma:

- **DAG** (çevrim yok) → lineer (DAG relaxation).



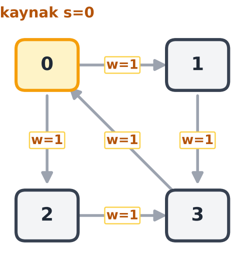
Şekil 38.4: Üç ünite, tek omurga (Ku L20 §3-5 İMZA): Quiz 1 (veri yapıları + sıralama) → Quiz 2 (çizgeler) → Quiz 3 (DP = çizge inşası). PANO 1 'Quiz 1 — bir şey BUL': iki kara kutu DİZİ (sıra: dinamik dizi / sıra-AVL) vs KÜME (anahtar: hash / küme-AVL / sıralı dizi); sıralama şeridi n^2 → n log n → lineer (kümeyi/sırayı HIZLANDIRIR). PANO 2 'Quiz 2 — İLİŞKİLER': düğüm=durum kenar=geçiş mini-çizge; SSSP merdiveni BFS → DAG ilişki → Dijkstra(w≥0) → Bellman-Ford (kısıt gevşedikçe süre doğrusaldan uzaklaşır). PANO 3 'Quiz 3 — çizgeyi SEN KUR': SRTBOT → alt-problem DAG'ı (alt problem=düğüm, bağımlılık=kenar, T=DAG); Ku alıntıları 'quiz 3 material as really an application of the graph material' + 'creative process in trying to construct that graph'. Üç pano amber oklarla bağlı; alt rozet 'tek özyinelemeli hikâye: BUL → İLİŞKİLENDİR → İNŞA ET'. KAVRAMSAL sentez (asimptotik etiketler sınıf adıdır, ölçülmüş rakam değil).

- **BFS** → ağırlıksız.
- **Dijkstra** → negatif olmayan ağırlık.
- **Bellman-Ford** → genel (negatif çevrim tespiti).

Bu merdivenin önemli bir özelliği: **uygulanabildiği yerde hepsi aynı cevabı verir**. Şekil 38.5 ağırlıksız ($w = 1$) küçük bir çizgede üç algoritmanın birebir aynı δ ürettiğini motor tanığıyla gösterir — `bfs == dijkstra(w=1) == bellman_ford_classic` her düğümde aynı; `_verify` D31 koşusunda `random.seed(31)` ile 60 rastgele çizgede `sssp_backbone_check` 60/60 True. Kural: **kısıtın en dar uygulanabilir basamağını seç** (en hızlısı odur).

Aynı çizge, üç algoritma: genellik ARTAR, cevap DEĞİŞMEZ

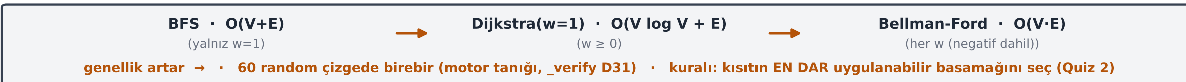
SSSP omurgası — ağırlıksız (w=1) çizgede BFS = Dijkstra(w=1) = Bellman-Ford aynı δ (uygulanabilir oldukça)



ağırlıksız çizge — her kenar w=1

düğüm	BFS	Dijkstra(w=1)	Bellman-Ford
0	0	0	0
1	1	1	1
2	1	1	1
3	2	2	2

üç sütun BİREBİR aynı → tek δ



Şekil 38.5: SSSP omurga tanığı (Ku L20 §4, motor-tanıklı): ağırlıksız (w=1) çizgede BFS = Dijkstra(w=1) = Bellman-Ford aynı δ — genellik ARTAR, cevap DEĞİŞMEZ. SOL küçük çizge (4 düğüm, kaynak s=0 amber, her kenar w=1); SAĞ tablo (satır=düğüm, sütun=BFS / Dijkstra(w=1) / Bellman-Ford) üç sütun BİREBİR aynı → tek δ (amber eşitlik kuşağı). Beklenen δ (kaynak 0): δ(0)=0, δ(1)=1, δ(2)=1, δ(3)=2 — _engine'den CANLI alınır + assert; sssp_backbone_check True. Alt rozet: SSSP merdiveni BFS O(V+E) → Dijkstra(w=1) O(V log V + E) → Bellman-Ford O(V·E) (genellik artar) · 60 random çizgede birebir (_verify D31) · kuralı kısıtın EN DAR uygulanabilir basamağını seç (Quiz 2).

38.6 5. Üç Ünite — Quiz 3: DP = Uygulamalı Çizge

DP, **özyinelemeli çerçevenin** çizgeye uygulanışıdır. SRTBOT aslında bir **çizge inşa eder**: alt problemler = düğümler, bağıntı = kenarlar, topolojik sıra + taban durumlar.

> *"You can actually think of the quiz 3 material as really an application of the graph material."* — Ku

Kilit fark: Quiz 2'de çizge sana **verilir**; Quiz 3'te çizgeyi **sen kurarsın** — bu yüzden DP yaratıcıdır ve daha zordur.

"There's a creative process in trying to construct that graph." — Ku

Bu nokta Şekil 38.4'in üçüncü panosunda görünür: SRTBOT, alt-problemleri düğüm, bağıntıyı kenar yapan bir DAG kurar; yaratıcı kısım o çizgeyi var etmektir. Üç ünite böylece tek omurgada kapanır — **bul → ilişkilendir → inşa et**.

38.7 6. 6.046 — Doğal Uzantı

6.046 (Design and Analysis of Algorithms) — lisans devam dersi (OMSCS karşılığı **CS 6515 Graduate Algorithms**). İki bakış. **Birincisi: 006'nın uzantısı** — algoritmayı *söylemek* kolay, ama **doğruluk + verimlilik analizi** çok daha zor. İçerik:

- **union-find** + **amortization via potential analysis** (dinamik dizideki "pahalı işi seyrek yap" sezgisinin formel hâli; dinamik bağlı bileşenler, near-constant sorgu).
- **Minimum Spanning Tree** (greedy) — ağırlıklı çizgede tüm düğümleri bağlayan min-ağırlık ağaç.
- **Network flow / max-flow & cuts** — kapasiteli çizgede source→sink max "su"; Dijkstra/Bellman-Ford gibi **artımlı** polinom algoritmalar.
- **Tasarım paradigmaları** (divide-and-conquer, DP, **greedy**) derinlemesine; **reductions** ile gerçekten NP-hard kanıtı.

Greedy zordur: tüm seçimleri taramaz, **lokal en iyiyi** seçip ilerler ve umut eder — DP'nin "hepsini dene"sinden farklı, daha çetin paradigma. Şekil 38.6 bu uzantıyı (sol sütun) ve bir sonraki bölümdeki tanım gevşetmeyi (sağ sütun) tek bir köprü şemasında toplar; her kart 6.006 köküne (← "…") bağlıdır.

38.8 7. 6.046 — "Doğru/Verimli" Tanımını Gevşetmek

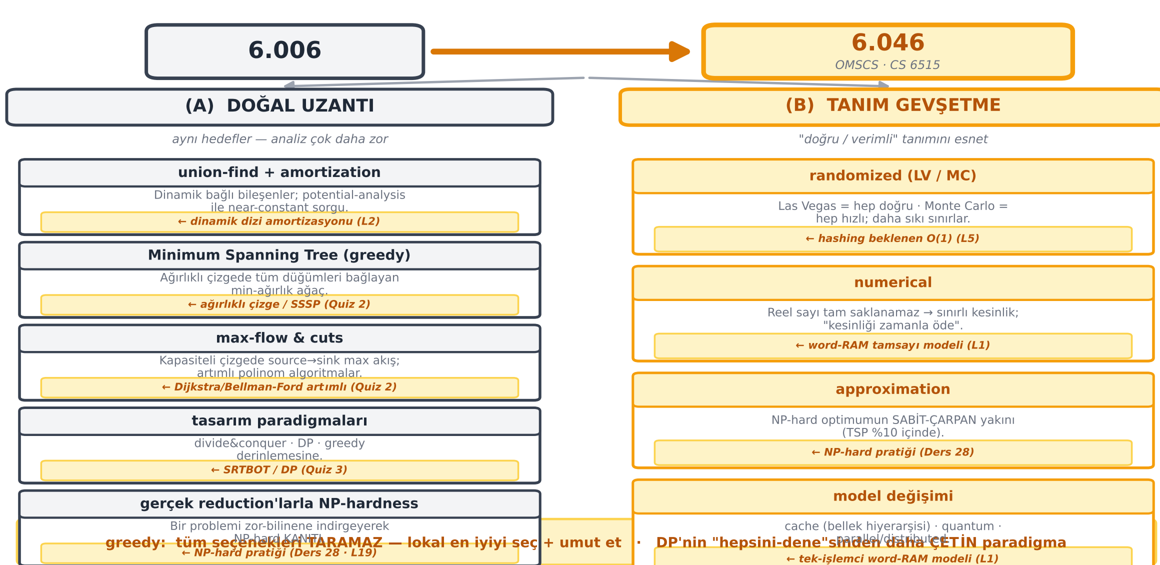
İkinci bakış: **doğruluk veya hesaplama modelini** gevşet.

"6.046 [is about] change my definition of what it means to be correct or efficient." — Ku

- **Randomized algoritmalar: Las Vegas** = *hep doğru, muhtemelen verimli* (hashing böyle — bazen zincir uzun). **Monte Carlo** = *hep verimli, muhtemelen doğru* (sabit zaman ama bazen yanlış). Randomization genelde daha hızlı sınırlar verir.
- **Numerical algoritmalar:** reel sayılar bilgisayarda tam saklanamaz → **sınırlı kesinlikle** hesapla; kesinliği zamanla öde (uzun bölme / karekök analojisi). Sürekli (continuous) optimizasyon.

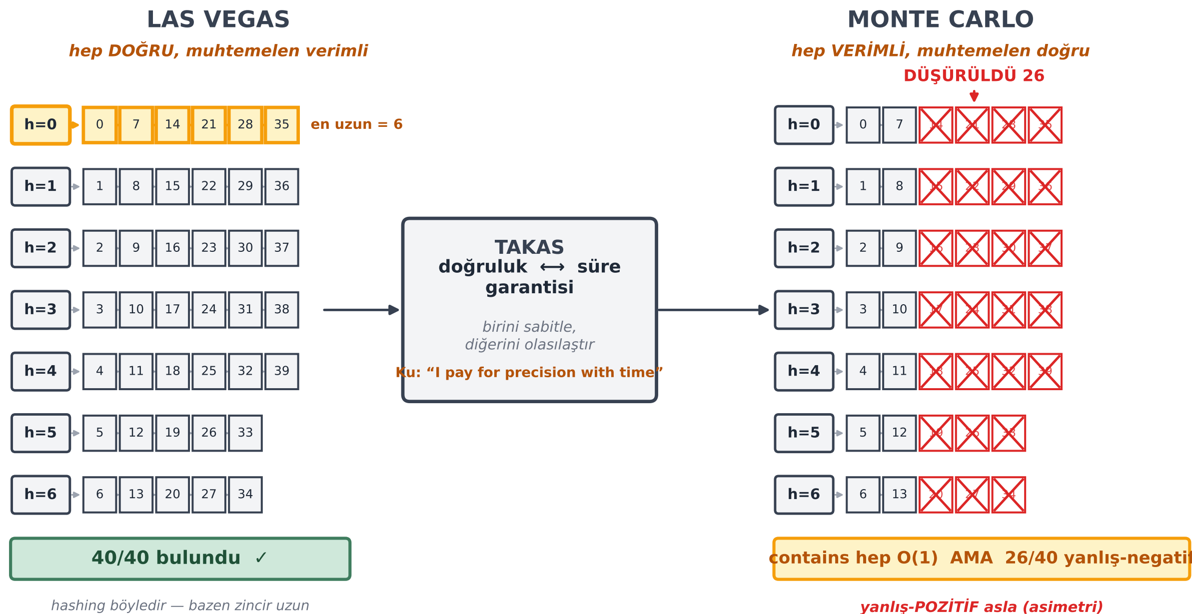
"I pay for precision with time." — Ku

- **Approximation algoritmalar:** NP-hard optimizasyonda optimumu değil, **sabit çarpan** yakınını polinom-zamanda hedefle (TSP'de %10 içinde).
- **Hesaplama modelini değiştir: cache model** (bellek hiyerarşisi: register/L1/…/disk farklı maliyet), **quantum** (entanglement/superposition — şifre kırma), **parallel** (k CPU → k-kat hızlanma; her zaman mümkün değil; multicore paylaşımlı bellek vs distributed).



Şekil 38.6: 6.006 → 6.046 köprüsü (Ku L20 §6-7) — iki bakış: doğal uzantı (006 köklerinden büyür) vs 'doğru/verimli' tanımını gevşetme. ÜST köprü bandı '6.006 → 6.046 (OMSCS · CS 6515)' amber ok. SOL sütun (A) DOĞAL UZANTI 5 kart: union-find + amortization (← dinamik dizi amortizasyonu L2); MST greedy (← ağırlıklı çizge / SSSP Quiz 2); max-flow & cuts (← Dijkstra/Bellman-Ford artımlı Quiz 2); tasarım paradigmaları d&c·DP·greedy (← SRTBOT/DP Quiz 3); gerçek reduction'larla NP-hardness (← NP-hard pratiği Ders 28·L19). SAĞ sütun (B) TANIM GEVŞETME 4 kart: randomized LV/MC (← hashing beklenen O(1) L5); numerical 'kesinliği zamanla öde' (← word-RAM tamsayı modeli L1); approximation sabit-çarpan yakın TSP %10 (← NP-hard pratiği Ders 28); model değişimi cache/quantum/parallel (← tek-işlemci word-RAM L1). Alt not: greedy tüm seçenekleri TARAMAZ — lokal en iyiyi seç + umut et; DP'nin 'hepsini-dene'sinden çetin. KAVRAMSAL figür (sayı yok).

Randomization'ın somut yüzü iki turdur — aynı 40 anahtarı aynı 7 kovaya iki farklı sözleşmeyle yerleştir. Şekil 38.7 bu **İMZA** tanığı motor-tanıklı gösterir: Las Vegas chaining (`HashTableChaining`) 40/40 doğru (en uzun zincir 6); Monte Carlo (`MonteCarloHashSet`, kova başına ≤2) `contains` hep $O(1)$ ama **26 öğe düşer** → tam 26 yanlış-negatif — ve **yanlış-pozitif asla** (asimetri). `monte_carlo_demo(range(40), 7)` birebir `(26, 26, False)` döndürür: yanlış-negatif = dropped, yanlış-pozitif yok.



Şekil 38.7: Las Vegas vs Monte Carlo (Ku L20 §7 İMZA, motor-tanıklı): aynı 40 anahtar, aynı 7 kova, iki randomizasyon sözleşmesi. SOL LAS VEGAS chaining (HashTableChaining) — hep DOĞRU, muhtemelen verimli; tüm zincirler saklanır (en uzun zincir 6 amber vurgulu); 40/40 bulundu ✓. SAĞ MONTE CARLO (MonteCarloHashSet, kova başına ≤2 öğe) — hep VERİMLİ, muhtemelen doğru; taşan öğeler kırmızı çarpıyla DÜŞÜRÜLDÜ (26 öğe) → contains hep O(1) AMA 26/40 yanlış-negatif; yanlış-POZİTİF asla (asimetri). ORTA TAKAS kutusu: doğruluk süre garantisi — birini sabitle, diğerini olasılaştır; Ku 'I pay for precision with time'. Tüm sayılar _engine'den CANLI: monte_carlo_demo(range(40), 7) == (26, 26, False); Las Vegas 40/40 doğru, en uzun zincir = max(chain_lengths) (assert).

38.9 Bu Dersin Özeti

1. **4 hedef:** algoritma üret · doğruluğu kanıtla (uygulamalı 6.042) · verimliliği kanıtla (model+asimptotik) · **insanlara ilet** (en önemli).
2. **L19 final'de:** tanımlar — P/NP/EXP iç içe, reduction yönü, hard vs complete.
3. **Quiz 1:** veri yapıları (dizi vs küme) + onları sömüren sıralama (heap/counting/radix).
4. **Quiz 2:** çizgeler — durum/geçiş; SSSP genellik-runtime takası (DAG/BFS/Dijkstra/Bellman-Ford).
5. **Quiz 3:** DP = çizge **inşası** (SRTBOT: düğüm/kenar/topolojik sıra); yaratıcı kısım çizgeyi kurmaktır.
6. **6.046:** ya 006'nın uzantısı (union-find/MST/flow/reductions) ya da tanım gevşetme (randomized/numerical/approximation/model).

> **!** Tek Bir Cümle
>
> 6.006 sadece algoritma yazmayı değil, doğruluk ve verimliliği **insanlara iletmeyi** öğretir; üç ünite tek bir omurgadır (veri yapısı → çizge → DP=çizge inşası) ve 6.046 bunu derinleştirip “doğru/verimli” tanımını gevşetir (randomized, approximation, farklı hesaplama modelleri).

38.10 Kontrol Soruları

> **i** Soru 1: Dersin 4 hedefinden hangisi «en önemli» sayılır ve neden quiz’de doğru kod tam puan almaz?
>
> **Cevap: İletişim (communication)** en önemli hedef. Diğer üçü — algoritma üretmek, doğruluğu kanıtlamak, verimliliği kanıtlamak — teknik olarak yeterli görünse de, 6.006 bunların **insanlara aktarılmasını** ister. Quiz’de doğru ama “ne yaptığı anlaşılmayan” bir Python script’i tam puan almaz; çünkü ders, bir algoritmanın **ne yaptığını ve neden doğru/verimli olduğunu** başkalarına iletme becerisini ölçer. Bilgisayar bilimi büyük ölçüde başkalarıyla çalışmaktır; iletemezsen, ne kadar yetkin olursan ol sınırlı kalırsın.

> **i** Soru 2: «Quiz 3 (DP) aslında çizge materyalinin uygulamasıdır» ifadesini SRTBOT üzerinden açıkla. Quiz 2’den farkı ne?
>
> **Cevap:** SRTBOT bir **çizge inşa eder**: **alt problemler** = düğümler, **bağıntı (R)** = kenarlar (bir alt problemin hangi küçük alt problemlere bağlı olduğu), **topolojik sıra (T)** = çizgenin DAG olması, **taban durumlar (B)** = çıkış-kenarı olmayan düğümler. Yani DP, üzerinde en kısa yol / erişilebilirlik tarzı bir hesaplama yapılan bir DAG’dır. **Quiz 2’den fark:** orada çizge sana **verilir** (düğüm ve kenarlar belli, algoritma uygula). Quiz 3’te çizgeyi **sen kurarsın** — düğümleri (alt problemleri) ve kenarları (bağıntıyı) yaratman gerekir. Bu yaratıcı inşa, DP’yi çizge algoritmalarını uygulamaktan daha zor yapar.

> **i** Soru 3: Las Vegas ve Monte Carlo randomized algoritmaları arasındaki fark nedir? Hashing hangisidir?
>
> **Cevap: Las Vegas:** *hep doğru, muhtemelen (beklenen) verimli*. **Monte Carlo:** *hep verimli, muhtemelen doğru*. Hashing bir **Las Vegas** örneğidir: bir öğenin kümede olup olmadığını **her zaman doğru** söyler, ama bazen zincir uzunsa beklenenden yavaştır (verimlilik olasılıksal). Aynı hash table’ı **Monte Carlo** yapabilirsin: her kovada çarpışanların yalnız ilk ikisini sakla → **her zaman sabit zaman** (hep verimli) ama bazen yanlış (saklanmayan öğe “yok” görünür). Şekil 38.7 bunu motor tanığıyla gösterir: aynı 40 anahtarda Monte Carlo 26 yanlış-negatif üretir ($O(1)$ garantisi karşılığında), Las Vegas 40/40 doğrudur. Pratikte bazen “bazen yanlış ama hep hızlı” iyi bir takastır.

> **i** Soru 4: 6.046’nın 006’yı iki farklı şekilde nasıl genişlettiğini özetle.
>
> **Cevap: (A) Doğal uzantı:** Aynı hedefler, ama **daha zor analiz**. Algoritmayı söylemek kolay; asıl zorluk doğruluk + verimlilik kanıtında. Yeni konular: union-find (potential analysis ile amortization), MST (greedy), network flow / max-flow & cuts (artımlı algoritmalar), tasarım paradigmaları (divide-conquer/DP/greedy derin), gerçek reduction’larla NP-hardness kanıtı. **(B) Tanım gevşetme:** “Doğru” veya “verimli/model” tanımını esnetir. **Randomized** (Las Vegas/Monte Carlo), **numerical** (reel sayılar

> sınırlı kesinlikle; kesinliği zamanla öde), **approximation** (NP-hard optimumun sabit-çarpan yakını), **model değişimi** (cache hiyerarşisi, quantum, parallel/distributed). İkisi birlikte 006'nın doğal devamı + yeni teori dallarıdır.

38.11 Egzersizler

Egzersiz 1. Quiz 1-2-3'ü tek cümlelik bir "omurga" hikâyesi olarak yaz (veri yapısı → çizge → DP=çizge inşası).

Egzersiz 2. Bir set veri yapısı seçimi için karar ağacı çiz: dinamik sıralı işlem gerekiyor mu? sözlük yeter mi? statik mi? (hash table / set AVL / sıralı dizi).

Egzersiz 3. Verilen bir problemi (örn. TSP) önce tam-optimal (NP-hard) sonra approximation açısından ele al; "%10 içinde yeter" yaklaşımının pratik değerini tartış.

Egzersiz 4. word-RAM ile cache modelini karşılaştır: hangi algoritma analizleri ikisinde farklı sonuç verir? (örn. ardışık vs rastgele bellek erişimi).

Egzersiz 5. Hashing'i Monte Carlo'ya çeviren "ilk iki çarpışmayı sakla" fikrini kodla ve hata olasılığını tartış.

38.12 Sonraki Ders İçin Hazırlık

Sonraki: Ders 32 (L21) — Son Ders: Algoritmalar Her Yerde (Ku + Demaine + Solomon)

Ders 32 (L21): Son Ders — dönemin kapanışı ve **kursun FİNALİ**. Üç hoca: **Jason Ku** açar, **Erik Demaine** ile **Justin Solomon** devam eder. Bu retrospektifteki "omurga" görüşü (veri yapısı → çizge → DP) ve 6.046 köprüsü, son derste nihai bağlamına oturur — algoritmaların gerçek dünyadaki uygulamaları.
Ders 32 Öncesi Yapılacak:

- L19 tanımlarını (P/NP/EXP, reduction yönü, hard/complete) final için gözden geçir.
- Üç ünitenin cheat sheet'lerini (Quiz 1-2-3 review dersleri) birlikte oku.
- 6.046 (CS 6515) konularına göz at: union-find, MST, max-flow, randomized.

38.13 Anahtar Kavramlar (Cheat Sheet)

Tema	Özet	Sayfada
4 Hedef	Algoritma · doğruluk · verimlilik · iletişim (en önemli)	Böl. 1
L19 (final)	P/NP/EXP iç içe · reduction yönü · hard vs complete	Böl. 2

Tema	Özet	Sayfada
Quiz 1	Veri yapıları (dizi vs küme) + sıralama (heap/counting/radix)	Böl. 3
Quiz 2	Çizge (durum/geçiş); SSSP genellik-runtime takası	Böl. 4
Quiz 3	DP = SRTBOT ile çizge inşası (yaratıcı)	Böl. 5
6.046 (A)	union-find, MST, max-flow, reductions (zor analiz)	Böl. 6
6.046 (B)	randomized, numerical, approximation, model değişimi	Böl. 7
Randomized	Las Vegas (hep doğru) vs Monte Carlo (hep hızlı)	Böl. 7

38.14 Builder ve OMSCS Bağlantıları

7 köprü

Bu retrospektifin omurga görüşü, dört hedefi ve 6.046 köprüsü, graduate algoritmalarına ve gerçek sistem mühendisliğine doğrudan bağlanır; köprülerin özeti:

1. **6.046 = CS 6515** (Graduate Algorithms) — OMSCS çekirdek dersi; bu retrospektif **birebir hazırlık** (omurga görüşü ve 6.046 konu listesi CS 6515 ile örtüşür).
2. **İletişim hedefi → kod review, tasarım dokümanı, teknik yazım**: "ne + neden" anlatımı; doğru kod yetmez, anlaşılır olmalı (en önemli builder becerisi).
3. **Max-flow / network flow** → matching, scheduling, segmentation; gerçek **sistem optimizasyonu** (kapasiteli çizgede source→sink akış pek çok kaynak-tahsis problemine oturur).
4. **Randomized (Las Vegas / Monte Carlo)** → hashing, **sketch / probabilistic veri yapıları (Bloom filter)**, ML sampling; "hep hızlı, bazen yanlış" takası `monte_carlo_demo`'nun çalışan örneğidir.
5. **Approximation** → NP-hard pratiği: heuristik + garanti; routing, packing (optimumun sabit-çarpan yakını yeterli olduğunda).
6. **Cache / parallel model** → gerçek performans mühendisliği; multicore, distributed, GPU (word-RAM tek-işlemci modeli yerine bellek hiyerarşisi/paralellik).
7. **Quantum** → post-kuantum kriptografi farkındalığı ($P \neq NP$ güvenliğinin kırılganlığı; quantum entanglement şifre kırmayı değiştirir).

❗ Tek bir şey alıp gideceksen

6.006'nın gerçek dersi algoritma yazmak değil, **doğruluğu ve verimliliği insanlara iletmektir** — quiz'de anlaşılmayan doğru kod tam puan almaz. Tüm dönem tek bir omurgadır: **veri yapıları** (Quiz 1, bir şey bul) → **çizgeler** (Quiz 2, ilişkiler) → **DP** (Quiz 3, çizgeyi *sen kur* — SRTBOT). Buradan **6.046**'ya (OMSCS CS 6515) gidersin: ya bu hikâyeyi derinleştir (union-find, MST, max-flow, gerçek reduction'lar) ya da "doğru/verimli" tanımını gevşet (randomized = Las Vegas/Monte Carlo, numerical = kesinliği zamanla öde, approximation = optimumun sabit-çarpan yakını, model = cache/quantum/parallel).

39 Son Ders: Algoritmalar Her Yerde

Üç geometrici hocanın araştırma vitrini — origami, self-assembly, mesh geodeziği, ray casting ve redistricting; kursun kapanışı

ℹ Oturum bilgisi

- **Hocaların videosu:** YouTube — Lecture 21: Algorithms—Next Steps (final) (≈50 dk)
- **OCW sayfası:** MIT 6.006 Lecture 21: Algorithms—Next Steps
- **Seri:** MIT 6.006 — Introduction to Algorithms (Spring 2020) — Ders 32 (L21) — **SON DERS**
- **Hoca:** ÜÇ HOCA — **Jason Ku** açar, **Erik Demaine** + **Justin Solomon** devam eder
- **Okuma süresi:** ≈28 dk

Kursun FİNALİ — tema: *algorithms are everywhere* (Demaine) + *6.006 is unavoidable* (Solomon). Üç hoca da geometrici; kendi araştırmalarında 6.006 araçlarının (NP-hardness, DP, çizge, veri yapısı, approximation, hesaplama modeli) nasıl döndüğünü gösterirler.

39.1 Bu Derste Ne Var?

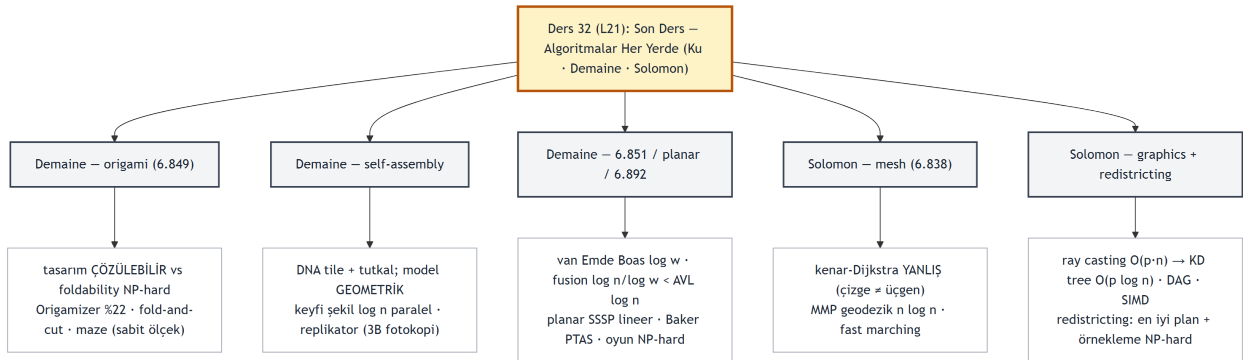
6.006'nın **son dersi** (Jason Ku açar, Erik Demaine + Justin Solomon devam eder). Üç hoca da **geometrici** — kendi araştırmalarında 6.006 materyalini nasıl kullandıklarını gösterirler. Tema tek cümle:

> *"algorithms are everywhere."* — Demaine

Ve Solomon'ın ısrarı:

> *"6.006 is unavoidable."* — Solomon

Uzmanlaşma dersleri (daha çok çizge, hesaplama modelleri, randomness, complexity) + uygulamalar (biyoloji, kriptografi, **grafik/geometri**) gezilir; sonra Demaine (computational origami, self-assembly, ileri veri yapıları, planar çizge, recreational) ve Solomon (mesh geometrisi, computer graphics, politik redistricting) somut örnekler verir.



Şekil 39.1: Ders 32'nin (L21) kavram haritası: kök = Son Ders Algoritmalar Her Yerde (Ku açar, Demaine ve Solomon devam) — kursun FİNALİ, üç geometrici hocanın araştırma vitrini. Beş dal — (1) Demaine origami 6.849: tasarım design hedef şekil to kıvrım deseni ÇÖZÜLEBİLİR vs katlanabilirlik foldability desen katlanır mı NP-hard; her şeyi katla 90 lar şerit yöntemi korkunç verimsiz; Origamizer Tachi 3B model to kare yüzde 22 alan çelik tavşan; fold-and-cut tek düz kesik herhangi şekil; maze folding köşe tipi gadget sabit ölçek faktörü. (2) Demaine self-assembly: DNA tile dört kenar tutkal tamamlayıcı yapışır sıcaklıkla ayarlanır; hesaplama modeli GEOMETRİK word-RAM tek komuttan farklı; keyfi şekil log n paralel adım sabit tutkal; replikator bilinmeyen şekli kalıplar 3B fotokopi. (3) Demaine ileri veri yapısı 6.851 planar recreational 6.892: van Emde Boas log w fusion tree log n bölü log w min karekök log n bölü log log n AVL log n den iyi; planar SSSP lineer Baker yaklaşımı BFS katmanları sil bir artı bir bölü k PTAS; oyun NP-hard Tetris Mario Portal Witness Recurse undecidable resim asma. (4) Solomon mesh 6.838: simplicial complex düğüm kenar ÜÇGEN; kenar Dijkstra YANLIŞ çizgeler üçgenlerle konuşamaz; doğru MMP geodezik n log n Dijkstra seviye kümeleri artı pencereleme; pratik fast marching yaklaşık hızlı. (5) Solomon ray casting 6.837 artı redistricting: ray casting O p çarpı n Stanford Bunny 69 bin üçgen 2 milyon piksel; sınırlayıcı kutu KD tree uzay bölme ağacı O p log n sezgisel; scene graph DAG 100 sandalye; GPU SIMD 30 fps; redistricting bağlı partisyon en iyi plan NP-hard tek düze örnekleme Hamiltonian indirgeme Yüksek Mahkeme. Birleştirici tema: 6.006 nın altı aracı sınıfta kalmıyor origami katlamadan DNA self-assembly ye mesh geodeziğinden ray casting ve politik gerrymandering e kadar her gerçek araştırma probleminde karşına çıkar algoritmalar her yerde 6.006 kaçınılmaz buradan 6.046 CS 6515 ve uzmanlık derslerine.

39.2 1. Üç Geometrici + Jason'ın Origami Yolculuğu

Bu dönemin üç hocası da geometriyle ilgilenir. **Jason Ku** makine mühendisi olarak başladı; tutkusu **origami**ydi. Origami modelleri tasarlarken kullandığı yöntemlerin aslında **algoritma** olduğunu sonradan fark etti. Demaine ile origami + **katlanabilir yapılar** (uzay uçuşu, açılır köprü/barınak, yeniden yapılandırılabilir madde) üzerine çalışmaya başladı — "telefonun maddesini yeniden programla" hayali (yazılım gibi maddeyi katla).

Bu ders, 6.006'nın **tüm araçlarının** gerçek araştırmada nasıl döndüğünü gösteren bir vitrindir: NP-hardness (Ders 28), DP (Quiz 3), çizge (Quiz 2), veri yapısı (Quiz 1), approximation, hesaplama modeli. Üç hoca sırayla origami, self-assembly, ileri veri yapıları, mesh geodeziği, ray casting ve redistricting örneklerini açar.

39.3 2. Demaine — Computational Origami (6.849)

İki problem tipi:

- **Tasarım (design):** hedef şekil → kıvrım deseni (crease pattern). Genelde **çözülebilir**.
- **Katlanabilirlik (foldability):** verilen kıvrım deseni katlanır mı? Genelde **NP-hard** (Demaine + Ku, genel foldability'nin NP-hard olduğunu kanıtladı).

> *"most of those problems are NP-hard."* — Demaine (foldability)

"Her şeyi katlayabilirsin": yeterince büyük bir kareden herhangi bir poligon/3B yüzey katlanabilir (90'lar; ince şeride katla + zigzag — ama malzemenin tamamına yakınını çöpe atar, korkunç verimsiz). Modern hedef: **ölçek faktörünü** küçült. **Origamizer** (Tomohiro Tachi; Demaine+Ku analiz etti): 3B model → kareden kat, **%22 alan** (çelik tavşan). **Maze folding** (ilk Demaine+Ku makalesi): dikdörtgenden labirent; her köşe tipi (derece 4/3/2) için küçük gadget kıvrımları tasarla, sınırları uyumluysa yapıştır → keyfi n×n labirent **sabit ölçek faktörüyle**.

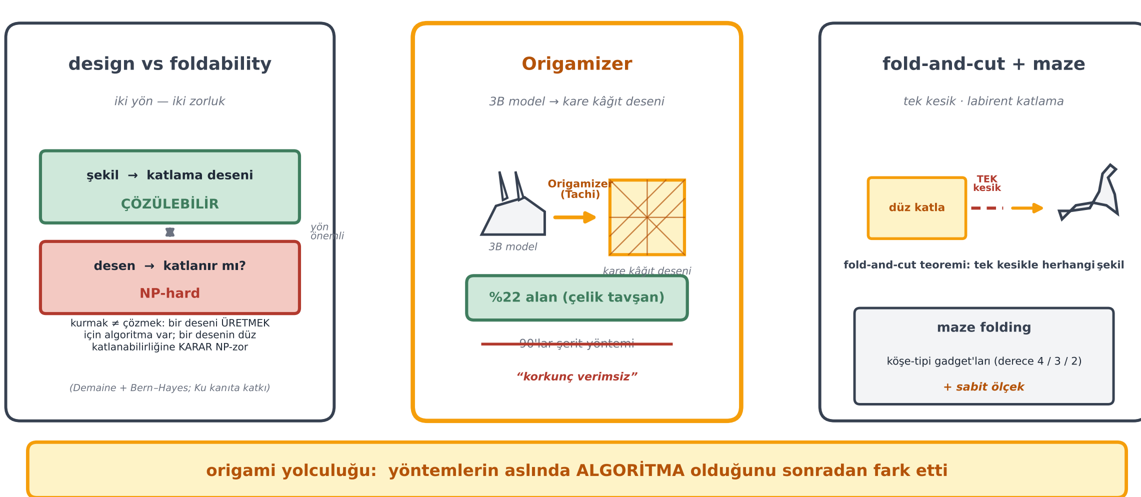
Fold-and-cut: kâğıdı düz katla + **tek düz kesik** → herhangi bir şekil (kuğu, melekbalığı, MIT logosu). Demaine'in ilk computational origami problemi; algoritma, çizdiğin herhangi bir çizgenin tüm kenarlarını hizalayan kıvrım desenini hesaplar. Şekil 39.2 bu üç fikri (tasarla vs katla, Origamizer, fold-and-cut + maze) tek panoda toplar — sayılar yalnız kaynak-alıntılıdır (%22 alan; sabit ölçek), motor rakamı değil.

Builder Notu — NP-hardness pratiği refleksi

Foldability'nin NP-hard olması, bir builder için "bu problemi her zaman verimli çözen genel algoritma arama, yön değiştir" refleksidir. **Tasarım yönü** (şekil → desen) çözülebilirken **karar yönü** (desen → katlanır mı?) NP-hard — aynı domain'in iki yönü farklı zorlukta. Bu, Ders 28'deki reduction pratiğinin gerçek araştırmadaki karşılığı: yönü değiştirip kolay tarafa odaklan. OMSCS CS 6515'te bu, "verilen problemin hangi yönünün polinom, hangisinin NP-hard olduğunu tanıma" becerisidir.

39.4 3. Demaine — Self-Assembly (Geometrik Hesaplama Modeli)

DNA şeritleriyle **kendiliğinden birleşen** kareler (tile): her karenin 4 kenarında "tutkal" (glue) deseni; yalnız tamamlayıcı desenler yapışır; sıcaklıkla ayarlanır (yüksek sıcaklık → yapışmaz, düşük → yanlış da yapışır). İyi ayarlanırsa **ikili sayaç** kuran bir sistem tasarlanır.



Şekil 39.2: Computational origami (Demaine, L21 §1-2 VİTRİN): tasarla vs katla · Origamizer · fold-and-cut + maze. KART 1 'design vs foldability' (iki yön, iki zorluk): şekil → katlama deseni ÇÖZÜLEBİLİR (yeşilimsi — verilen 3B hedefe ulaşan kıvrım desenini ÜRETMEK için algoritma var) vs desen → katlanır mı? NP-hard (kırmızı — bir desenin düz katlanabilirliğine KARAR vermek NP-zor; Demaine + Bern–Hayes, Ku kanıta katkı). KART 2 'Origamizer' (amber vurgu): 3B model (çelik tavşan) → TEK kare kâğıt kıvrım deseni (Tachi); rozet '%22 alan (çelik tavşan)' kaynak-alıntılı verim; 90'ların 'şerit yöntemi' üstü çizik 'korkunç verimsiz'. KART 3 'fold-and-cut + maze': düz katla + TEK düz kesik → herhangi düz-kenarlı şekil (kuğu silueti); maze folding köşe-tipi gadget'ları (derece 4/3/2) + sabit ölçek. Alt şerit: origami yolculuğu — yöntemlerin aslında ALGORİTMA olduğunu sonradan fark etti. KAVRAMSAL VİTRİN (sayı yalnız kaynak-alıntılı: %22 alan, sabit ölçek — motor rakamı değil).

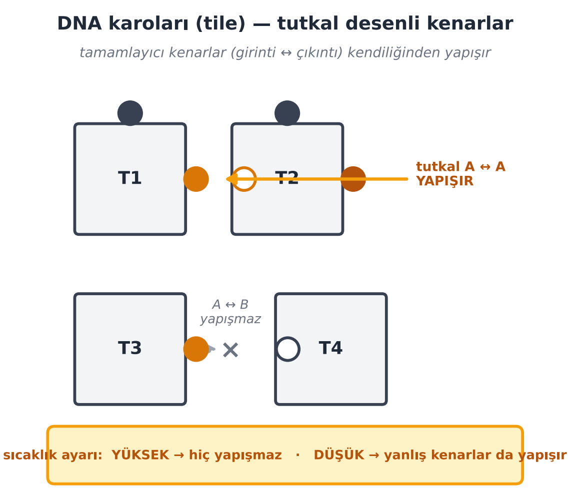
"the model of computation is geometric." — Demaine

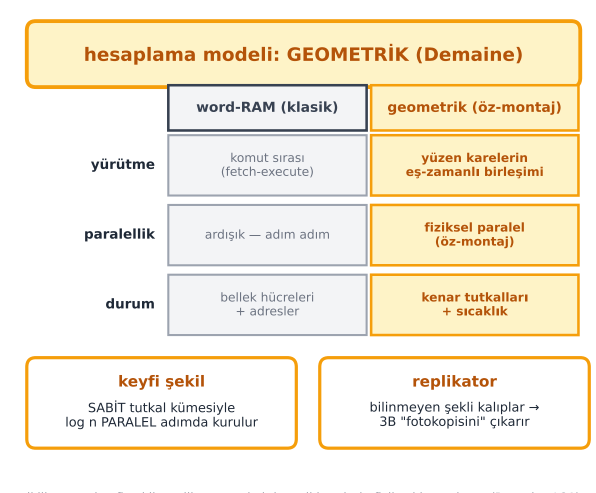
Bu, 6.006'nın "tek tek çalışan komut" modelinden **çok farklı**: program, o anda yüzen karelerin birleşimi. Bu modelde keyfi bir şekli **log n paralel adımda** (sabit sayıda tutkalla) inşa etmek kanıtlanabilir; hatta bir **replikatör** (bilinmeyen şekli kalıplayıp 3B fotokopi) kurulabilir. Şekil 39.3 DNA tile şemasını (tamamlayıcı kenarlar yapışır, sıcaklık ayarı) word-RAM vs geometrik model karşılaştırmasıyla yan yana koyar.

Builder Notu — hesaplama modeli sınıfta kalmaz

Self-assembly'nin "geometrik hesaplama modeli", word-RAM'in tek-işlemci komut sırası varsayımının **bir alternatifidir** — Ders 31'deki 6.046 köprüsünün "model değişimi" maddesi burada fiziksel forma kavuşur. Klasik analiz "adım sayısı"nı sayar; geometrik modelde program yüzen karelerin eş-zamanlı

Demaine vitrini: DNA öz-montaj (self-assembly) + GEOMETRİK hesaplama modeli (L21 §3)





Şekil 39.3: Demaine vitrini: DNA öz-montaj (self-assembly) + GEOMETRİK hesaplama modeli (L21 §3 VİTRİN). SOL panel DNA karoları (tile): 4 kenarı tutkal (glue) desenli kareler; tamamlayıcı kenarlar (girinti ↔ çıkıntı) kendiliğinden yapışır — T1.sağ (tutkal A) ↔ T2.sol (tutkal A) amber okla YAPIŞIR; farklı tutkal (A ↔ B) yapışmaz (× işareti). Alt şerit sıcaklık ayarı: YÜKSEK → hiç yapışmaz, DÜŞÜK → yanlış kenarlar da yapışır. SAĞ panel 'hesaplama modeli GEOMETRİK (Demaine)': word-RAM vs geometrik mini-tablo (yürütme komut-sırası vs yüzen karelerin eş-zamanlı birleşimi; paralellik ardışık vs fiziksel paralel; durum bellek-hücreleri vs kenar-tutkalları + sıcaklık) + iki rozet (keyfi şekil SABİT tutkal kümesiyle log n PARALEL adımda kurulur; replikatör bilinmeyen şekli kalıplar → 3B fotokopisini çıkarır). KAVRAMSAL VİTRİN (sayı yok;

39.5 4. Demaine — İleri Veri Yapıları, Planar Çizgeler, Recreational

İleri Veri Yapıları (6.851): dinamik **sıralı** küme (insert/delete + find-next/prev). 6.006'da gördüğümüz **set AVL** (Ders 10) = $O(\log n)$. Daha iyisi word-RAM'de: **van Emde Boas** = $O(\log w)$ (w = kelime boyutu), **fusion trees** = $O(\log n / \log w)$. İkisinin min'i ≈ $O(\sqrt{\log n / \log\log n})$ — AVL'nin $\log n$'inden hayli iyi (sıralı küme için sabit-zaman **imkânsız**, kanıtlanır). Bunlar **kaynak-formül**dür (Demaine L21), motor değil.

Planar Çizgeler: yolu düzlemde çizişimsiz (veya az çizişimli) çizge. Planar'da SSSP **lineer** (Dijkstra'nın $v \log v$'si yerine — Ders 19); planar Bellman-Ford ≈ **lineer** ($v \log^2 v / \log\log v$, v^2 yerine). **Baker yaklaşımı (1994):** BFS katmanlarına ayır (Ders 13'teki BFS katmanları), her k'ıncı katmanı sil (~$1/k$ kayıp) → $1 + 1/k$ yaklaşım; kalan sabit-katmanlı yapı "birkaç döngü" → fancy DP polinom-zamanda çözer → **PTAS** (her ε için $1 + \varepsilon$, ama büyük ε daha yavaş).

Recreational (6.892): oyun/bulmaca zorluk kanıtları. Tetris, Super Mario, Portal, The Witness **NP-hard**; "Recurse" **undecidable** (mükemmel oynayan algoritma yok). Ayrıca balon büküm, **resim-asma problemi** (iki çividen herhangi biri çıkınca resim düşsün; monoton Boole fonksiyonları; Rivest ile sonuç).

39.6 5. Solomon — Mesh Üzerinde En Kısa Yol (Dijkstra Neden Yanlış)

Solomon uygulamalı geometri/graphics çalışır (6.837 Computer Graphics, 6.838 geometri işleme). Ana nesne: **simplicial complex** — düğüm + kenar + **üçgen** kümesi. 6.006'da çizge sadece düğüm+kenardır; graphics'te bu bir **yüzey**dir.

Tuzak: üçgen mesh'te iki köşe arası en kısa yol için kenarlarda Dijkstra (Ders 19) koşturmak **yanlış** (sık yapılan hata). Çapraz geçen gerçek geodezik daha kısadır:

> *"graphs don't know how to talk to triangles."* — Solomon

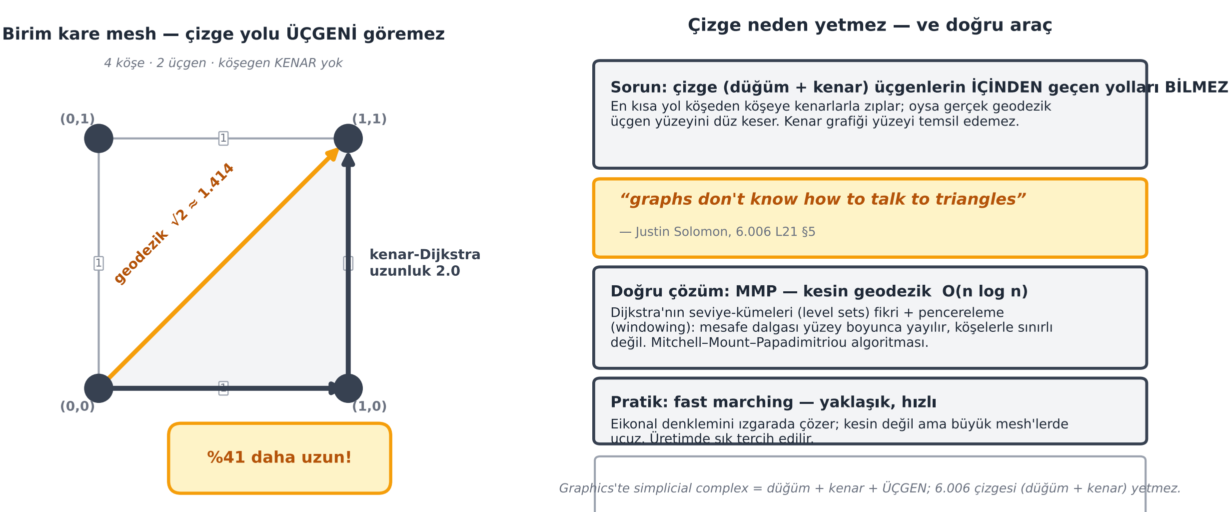
Şekil 39.4 bunu motor tanığıyla gösterir: birim kare mesh'te (4 köşe, 2 üçgen, köşegen **kenar yok**) karşı köşeye kenar-Dijkstra iki eksen-paralel kenar boyunca 2.0 verir; gerçek geodezik üçgen yüzeyini çaprazlar ve $\sqrt{2} \approx 1.4142$ olur. Oran $\sqrt{2}$ — kenar-Dijkstra **%41 daha uzun**. Tüm bu sayılar `mesh_edge_vs_geodesic()`'ten **canlı** gelir (kenar yolu $= 2.0$, geodezik $= \sqrt{2}$, oran $= \sqrt{2}$; assert ile); `_verify` D32 koşusunda PASS.

Doğru algoritma **MMP** (üçgen alan üzerinde geodezik, $O(n \log n)$) — Dijkstra'nın mesafe-fonksiyonu **seviye kümeleri** fikrini genişletir (ama pencereleme ile). Pratikte **fast marching** (geodeziğin yaklaşığı; daha hızlı, kolay, neredeyse ayırt edilemez) tercih edilir.

39.7 6. Solomon — Ray Casting & Stanford Bunny

Ray casting (6.837): her ekran pikseli için, gözden bir ışın gönder, çarptığı **ilk** nesneyi bul → renk. Maliyet: piksel sayısı × nesne sayısı = $O(p \cdot n)$. **Stanford Bunny** (69.000 üçgen) × 1080p (~2 milyon piksel) = iki büyük sayının çarpımı, çok yavaş. Bu rakamlar (69.000 üçgen, ~2 milyon piksel, ~30 fps bütçesi) **kaynak verisidir** (L21 birebir), motor değil. Her graphics özelliği (saydamlık, yansıma, parçalanma) maliyeti artırır.

Çıkış: veri yapıları + algoritmalar. Tavşanı bir kutuya koy; ışın kutuya değmiyorsa tavşana değmez ($O(1)$ hızlanma). Kutuyu özyinelemeli ikiye böl → **uzay-bölme ağacı** (KD tree). İdealde $O(p \log n)$ (ağaçta gez) — ama bu bir **sezgisel**; veriye bağlı, ortalama $\log n$. **Scene graph:** sahnedeki 100 özdeş sandalye için 100



Şekil 39.4: Çizgeler üçgenlerle konuşamaz: kenar-Dijkstra (2.0) vs gerçek geodezik (√2) — Solomon L21 §5 İMZA, motor-tanıklı. SOL birim kare mesh: 4 köşe (0,0)(1,0)(1,1)(0,1), iki üçgen yüzey, köşegen KENAR yok; eksen-paralel kenarlar w=1 ince slate; kenar-Dijkstra yolu (0,0)→(1,0)→(1,1) kalın slate KIRIK yol ‘uzunluk 2.0’; gerçek geodezik (0,0)→(1,1) düz amber çapraz ‘√2 ≈ 1.414’; ‘%41 daha uzun!’ rozeti. Tüm sayılar mesh_edge_vs_geodesic()‘ten CANLI: kenar yolu 2.0, geodezik √2, oran √2, yüzde (√2−1)·100 ≈ 41 (assert). SAĞ kural kutuları: Sorun çizge (düğüm+kenar) üçgenlerin İÇİNDEN geçen yolları BİLMEZ; Solomon alıntısı ’graphs don’t know how to talk to triangles’; Doğru çözüm MMP kesin geodezik O(n log n) — Dijkstra’nın seviye-kümeleri + pencereleme (Mitchell–Mount–Papadimitriou); Pratik fast marching eikonal denklem ızgarada yaklaşık hızlı. Alt not: graphics’te simplicial complex = düğüm + kenar + ÜÇGEN; 6.006 çizgesi yetmez.

model saklama; bir sandalye + her kopya için bir dönüşüm → **DAG** (yönlü çevrimsiz çizge). GPU = **SIMD** paralellik; numerical + approximation; algı koruma (~30 fps bütçesi).

Şekil 39.5 bu hızlandırmayı Solomon örneğinin 1B analoğuyla motor-tanıklı gösterir: aynı sahnede (seed-32 ile 30 nesne × 25 ışın) naif `ray_cast_naive` **750 karşılaştırma**, indexed `ray_cast_indexed` (sırala + ikili arama) **409 karşılaştırma** yapar — ve **hit'ler birebir aynı** (doğruluk korunur, yalnız iş azalır). Bu sayaçlar motordan canlı gelir (assert ile); `_verify` D32 koşusunda daha geniş 40-rastgele sahnede de naif toplamı indexed toplamından daima büyüktür (motor tanığı; örn. 5526 > 4494 türünden — naif > indexed daima).

> Builder Notu — KD tree / collision gerçek-sistem
>
> Ray casting'in $O(p \cdot n) \to O(p \log n)$ kazancı, bir builder için **uzay-bölme veri yapısı** (KD tree, BVH, octree) refleksidir: "her nesneyi her sorgu için tarama, sahneyi hiyerarşik kutula". Aynı yapı **çarpışma tespiti** (collision detection), 3B tarama, oyun navigasyonu ve nearest-neighbor aramada döner. Dikkat çekici nokta Solomon'ın "**sezgisel** — $\log n$ veri dağılımına bağlı" uyarısıdır: gerçek sistemde garanti değil, ortalama-durum performans mühendisliğidir. Scene graph'ın **DAG**'ı (Ders 19'daki DAG soyutlaması) ise tekrar eden geometriyi bir kez saklayıp dönüşümle paylaştırır — bellek mühendisliğinin temel deseni.

39.8 7. Solomon — Politik Redistricting (Gerrymandering)

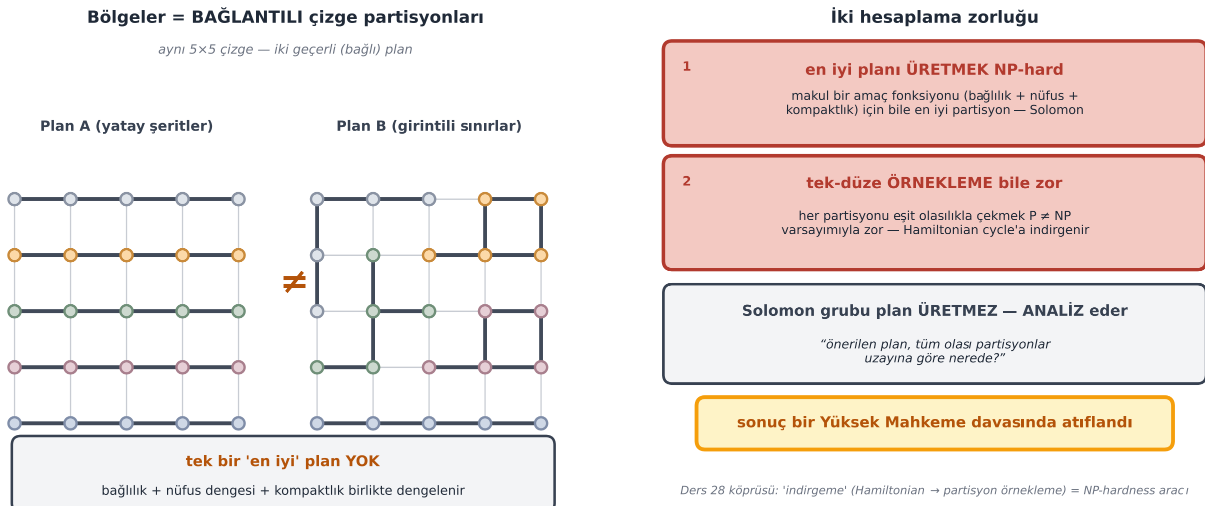
ABD'de Kongre seçim bölgelerini çizmek (gerrymandering: sınırları çizerek sonucu mühendislik). Bölgeler = **bağlantılı çizge partisyonları** (düğümleri bağlı kalacak şekilde kümele). Sorunlar: (1) tek bir "en iyi" plan yok — birden çok kriter (bağlantılılık, nüfus dengesi, kompaktlık) dengelenir; (2) makul bir amaç fonksiyonu için bile **en iyi planı üretmek NP-hard**.

> *"generating the best possible districting plan is NP-hard."* — Solomon

Solomon'ın grubu plan **üretmek** yerine **analiz** eder: "önerilen plan, tüm olası partisyonlar uzayına göre nerede?" Ama **tek-düze örnekleme** (her partisyon eşit olasılık) bile **hesaplama olarak zor** (P ≠ NP varsayımıyla; **Hamiltonian cycle**'a indirgenir — Ders 28'deki reduction aracı). Bu sonuç bir **Yüksek Mahkeme** davasında atıflandı. Şekil 39.6 bu iki zorluğu (en iyi plan + örnekleme) iki geçerli bağlı partisyon planının yanında gösterir.

> Builder Notu — MCMC / hukuk + algoritma kesişimi
>
> Redistricting analizi, NP-hardness'ın sosyal-bilim ve **hukuk** kesişimindeki en somut örneğidir: "en iyi planı üretemiyoruz, ama önerilen bir planı tüm partisyon uzayına göre **örnekleyerek** konumlandırabiliriz". Tek-düze örnekleme bile zor olduğundan (Hamiltonian indirgemesi), pratikte **MCMC** (Markov Chain Monte Carlo) ile yaklaşık örnekleme kullanılır — Ders 31'in 6.046 köprüsündeki "randomized + sampling" maddesinin gerçek araştırma karşılığı. Bir builder için ders ikiyönlü: (1) örnekleme araçlarına temkinli yaklaş (yanlılık olabilir), (2) algoritmik analiz mahkeme delili olabilir — hesaplamalı sosyal bilim gerçektir.

Şekil 39.5: Ray casting hızlandırma: naif O(p·n) → sıralı + ikili arama (uzay-bölme ağacı) — Solomon L21 §6 İMZA, motor-tanıklı. ÜST 1B model şeması (deterministik seed-32 sahne: 30 nesne × 25 ışın): sayı doğrusu üzerinde aralıklar (nesneler) + ışın noktaları (amber); NAİF her ışın × her nesne çizgi-demeti (yoğun slate, O(p·n)); INDEXED sırala + ikili arama (seyrek amber, ~O(p log n)). Sayaç rozetleri MOTORDAN: naif karşılaştırma 750 vs indexed 409 (kazanç 750→409); 'hit'ler BİREBİR aynı' doğruluk tanığı (yeşil). Tüm sayaçlar ray_cast_naive / ray_cast_indexed'ten CANLI (assert: c1==750, c2==409, hit'ler eşit). ALT Stanford Bunny anlatım kutusu (kaynak L21 §6): Naif O(p·n) = 69.000 üçgen × ~2 milyon piksel (1080p) iki dev sayının çarpımı → sınırlayıcı kutu O(1) eleme → uzay-bölme ağacı (KD) → ideal O(p log n) SEZGİSEL veriye bağlı (Solomon). Scene graph: 100 özdeş sandalye = 1 model + 100 dönüşüm = DAG. Donanım: GPU SIMD paralellik + numerical/approximation + ~30 fps render bütçesi. Bunny rakamları kaynak L21 (motor değil); sayaçlar motordan.



Şekil 39.6: Politik redistricting (Solomon, L21 §7 VİTRİN): bağlı partisyonlar · en iyi plan + örnekleme NP-hard. SOL panel 5×5 ızgara-çizge, iki BAĞLI partisyon örneği yan yana: Plan A (yatay şeritler) ≠ Plan B (girintili sınırlar) — AYNI çizge, FARKLI bağlı kümeleme (her bölge = kenar-komşu çizgede bağlı düğüm kümesi, 5 bölge × 5 düğüm; bölge-içi kenarlar kalın koyu, kesilen sınırlar ince soluk). Not: tek bir 'en iyi' plan YOK — bağlılık + nüfus dengesi + kompaktlık birlikte dengelenir. SAĞ panel iki hesaplama zorluğu: (1) en iyi planı ÜRETMEK NP-hard (makul amaç fonksiyonu için bile — Solomon, kırmızı); (2) tek-düze ÖRNEKLEME bile zor (her partisyonu eşit olasılıkla çekmek P ≠ NP varsayımıyla zor — Hamiltonian cycle'a indirgenir, kırmızı). Analiz notu: Solomon grubu plan ÜRETMEZ — ANALİZ eder ('önerilen plan tüm olası partisyonlar uzayına göre nerede?'). Rozet (amber): sonuç bir Yüksek Mahkeme davasında atıflandı. Alt not: Ders 28 köprüsü — 'indirgeme' (Hamiltonian → partisyon örnekleme) = NP-hardness aracı. KAVRAMSAL VİTRİN (sayı yok; NP-hard/örnekleme/Yüksek Mahkeme iddiaları kaynak-alıntılı).

39.9 Bu Dersin Özeti

1. **Üç hoca, tek tema:** 6.006 araçları gerçek araştırmada her yerde.
2. **Demaine — origami:** design (çözülebilir) vs foldability (**NP-hard**); Origamizer; fold-and-cut; maze folding.
3. **Demaine — self-assembly:** DNA tile, **geometrik** hesaplama modeli (log n paralel).
4. **Demaine — 6.851/planar/6.892:** van Emde Boas (log w) < AVL (log n); planar SSSP lineer; Baker PTAS; oyun NP-hardness.
5. **Solomon — mesh:** Dijkstra kenarlarda **yanlış** (çizge üçgenle konuşmaz); MMP geodezik (n log n).
6. **Solomon — graphics:** ray casting $O(p \cdot n)$ → uzay-bölme ağacı ≈ $O(p \log n)$; scene graph **DAG**; SIMD/GPU.
7. **Solomon — redistricting:** gerrymandering; en iyi plan + tek-düze örnekleme **NP-hard** (Hamiltonian indirgemesi).

! Tek Bir Cümle

6.006'nın araçları — NP-hardness, DP, çizge, veri yapısı, approximation, hesaplama modeli — origami katlamadan DNA self-assembly'ye, mesh geodeziğinden ray casting ve politik gerrymandering'e kadar her gerçek problemde karşına çıkar: algoritmalar her yerde, 6.006 kaçınılmaz.

39.10 Kontrol Soruları

i Soru 1: Origami'de «tasarım» ve «katlanabilirlik» problemleri nasıl farklı, ve hangisi NP-hard?

Cevap: Tasarım (design): hedef şekilden başlayıp onu üreten kıvrım desenini (crease pattern) bulmak — şekil → desen. Genelde **çözülebilir** (algoritmalar var; örn. Origamizer 3B modeli kareden katlar, %22 alan). **Katlanabilirlik (foldability):** verilen bir kıvrım deseninin düz katlanıp katlanamayacağını belirlemek — desen → "katlanır mı?". Demaine ve Ku bu genel problemin **NP-hard** olduğunu kanıtladı. Bu yüzden araştırma daha çok (daha kolay olan) tasarım tarafına odaklanır. Şekil 39.2 iki yönü ("kurmak ≠ çözmek") yan yana gösterir.

i Soru 2: Üçgen mesh'te iki köşe arası en kısa yol için kenarlarda Dijkstra koşturmak neden yanlıştır?

Cevap: Çizge (düğüm+kenar) **üçgenlerin içinden geçen** yolları bilmez — Dijkstra yalnız kenarlar boyunca gidebilir. Sol-üstten sağ-alta giden gerçek geodezik bir üçgenin **yüzeyini çaprazlama** kestiğinde daha kısa olur; oysa kenar-Dijkstra düğümden düğüme zikzak çizmek zorundadır ve daha uzun bir yol bulur. Şekil 39.4 motor tanığıyla gösterir: birim karede kenar-Dijkstra 2.0, gerçek geodezik $\sqrt{2} \approx 1.4142$ — %41 fark. Solomon'ın deyişiyle "graphs don't know how to talk to triangles". Doğru çözüm **MMP** algoritmasıdır (geodezik, $O(n \log n)$); Dijkstra'nın mesafe **seviye kümeleri** fikrini üçgen alana (pencereleme ile) genişletir. Pratikte **fast marching** (yaklaşık ama hızlı) sık kullanılır.

ℹ Soru 3: Stanford Bunny'yi render ederken naif ray casting neden yavaş, ve uzay-bölme ağaçları nasıl yardımcı olur?

Cevap: Naif ray casting her piksel için **tüm** nesneleri (üçgenleri) tarar → $O(p \cdot n)$. Stanford Bunny 69.000 üçgen × ~2 milyon piksel (1080p) = iki büyük sayının **çarpımı**, çok yavaş; her graphics özelliği (yansıma, saydamlık) bunu artırır. **Çözüm:** tavşanı bir sınırlayıcı kutuya koy — ışın kutuya değmezse hiçbir üçgeni kontrol etme ($O(1)$ eleme). Kutuyu özyinelemeli ikiye böl → **uzay-bölme ağacı** (KD tree); ışın ağaçta gezerek çoğu üçgeni atlar, ideal $O(p \log n)$. Şekil 39.5 bunu motor-tanıklı 1B sahnede gösterir (naif 750 → indexed 409 karşılaştırma, hit'ler birebir aynı). Dikkat: bu bir **sezgisel** — gerçek $\log n$ garantisi değil, veri dağılımına bağlı ortalama davranış. (Ayrıca özdeş nesneler **scene graph** = DAG ile bir kez saklanır.)

ℹ Soru 4: Politik redistricting'te «en iyi plan» ve «tek-düze örnekleme» neden hesaplama açısından zor?

Cevap: İki katman: (1) **En iyi plan NP-hard:** seçim bölgeleri bağlantılı çizge partisyonlarıdır; makul herhangi bir amaç fonksiyonu (bağlantılılık + nüfus dengesi + kompaktlık) için en iyi partisyonu bulmak NP-hard'dır — üstelik kanun zaten "en iyi"yi şart koşmaz. (2) **Tek-düze örnekleme zor:** bir planı, tüm olası partisyonlar uzayından **rastgele eşit-olasılıkla** çekilen bir planla kıyaslamak için, her partisyonu $1/N$ olasılıkla (N = toplam partisyon sayısı) üreten bir araç gerekir; bu **P ≠ NP varsayımıyla zordur** (Hamiltonian cycle'a indirgenir — Ders 28 reduction aracı). Yani gerrymandering analizinde kullanılan örnekleme araçlarına temkinli yaklaşmak gerekir — sonuç bir Yüksek Mahkeme davasında bile atıflandı.

39.11 Egzersizler

Egzersiz 1. Bu derste geçen her araştırma örneğini bir 6.006 kavramıyla eşle (foldability→NP-hard, mesh path→Dijkstra/geodezik, ray casting→ağaç/log n, scene graph→DAG, redistricting→Hamiltonian reduction).

Egzersiz 2. van Emde Boas ($\log w$) ile fusion tree ($\log n / \log w$) sınırlarının min'ini al; $w = \log n$ ve $w = n$ için hangisi kazanır?

Egzersiz 3. Baker yaklaşımında her 4. BFS katmanını silmek neden $1 + 1/4$ (yani %25 içinde) bir çözüm verir? k arttıkça runtime'a etkisi?

Egzersiz 4. Ray casting'i scene graph (DAG) ile birleştir: 100 özdeş sandalyeli bir sahnede bellek ve render maliyeti nasıl değişir?

Egzersiz 5. Self-assembly'nin "geometrik hesaplama modeli"ni word-RAM ile karşılaştır: hangi işlemler paralel, hangi maliyet farklı?

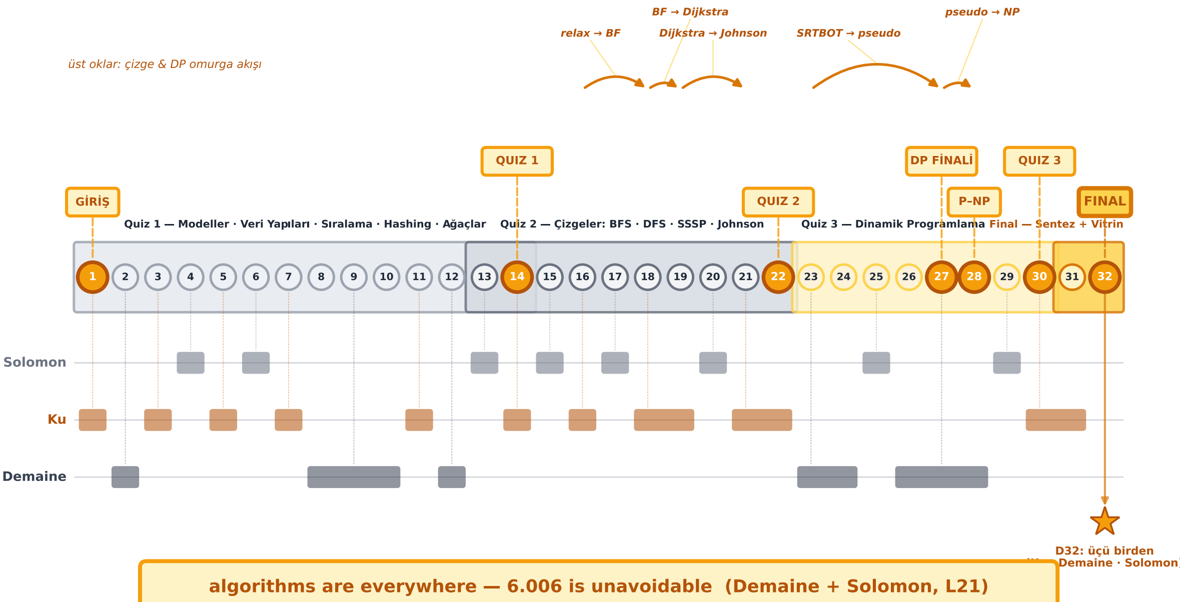
39.12 6.006 Tamamlandı — Kurs Kapanışı

Bu, 6.006'nın **son dersiydi**. 32 video boyunca: hesaplama modeli → veri yapıları → sıralama → hashing → ağaçlar/yığınlar → çizgeler/BFS/DFS → en kısa yollar (BFS/DAG/Dijkstra/Bellman-Ford/Johnson) → dinamik programlama (SRTBOT) → hesaplama karmaşıklığı (P/NP) → sentez → araştırma vitrini. Şekil 39.7

bu 32-ders yolculuğunu tek bir nehir-harita olarak — dört Quiz bloğu, üç hoca, yedi kilometre taşı — gösterir; kitabın kapanış görseli.

> *"I hope you enjoyed this class… algorithms are everywhere."* — Demaine (kapanış)

Sonraki adım (Builder/OMSCS): 6.046 (CS 6515 Graduate Algorithms) ve uzmanlık dersleri (6.849 folding, 6.851 ileri veri yapısı, 6.837/6.838 graphics/geometry, 6.892 recreational).



Şekil 39.7: 6.006 — 32 dersin tam haritası (İMZA KURS KAPANIŞI): dört Quiz bloğu, üç hoca, tek nehir. 32 ders-düğümü soldan sağa DÖRT blok-bandında dizilir: Quiz 1 (D1-12 + D14: modeller · veri yapıları · sıralama · hashing · ağaçlar, slate-soluk); Quiz 2 (D13-22: çizgeler BFS/DFS/SSSP/Johnson, slate); Quiz 3 (D23-30: dinamik programlama, amber-soluk); Final (D31-32: sentez + vitrin, amber). ÜÇ HOCA şeritleri (altta): Solomon / Ku / Demaine hangi dersleri taşır; D32 ÜÇÜ BİRDEN (amber yıldız: Ku · Demaine · Solomon). Yedi kilometre taşı rozeti: D1 GİRİŞ · D14 QUIZ 1 · D22 QUIZ 2 · D27 DP FİNALİ · D28 P–NP · D30 QUIZ 3 · D32 FINAL. Üst köprülü oklar (çizge & DP omurga akışı): D16 relax→BF, D18 BF→Dijkstra, D19 Dijkstra→Johnson, D23 SRTBOT→pseudo, D27 pseudo→NP. Kapanış rozeti: 'algorithms are everywhere — 6.006 is unavoidable (Demaine + Solomon, L21)'. KAVRAMSAL KURS-YAPISI figürü (ders numaraları + blok yapısı; sayısal motor iddiası yok).

> ⚠ Kurs Kapanışı + Buradan Nereye (Ders 32 — SON DERS)
>
> **Ders 32 (L21) kursun SON dersiydi** — sonraki ders yok. Üç geometrici hoca (Ku · Demaine · Solomon) 6.006 araçlarının gerçek araştırmadaki vitrinini kapattı. Ders 31'in (L20) omurga görüşü (veri yapısı → çizge → DP) ve 6.046 köprüsü, burada nihai bağlamına oturdu: algoritmalar her yerde.
> **Buradan nereye:**
>
> - **6.046** (Design and Analysis of Algorithms) = **OMSCS CS 6515** (Graduate Algorithms) — doğal

devam dersi; Ders 31'deki 6.046 köprüsü (union-find, MST, max-flow, randomized, approximation, model değişimi) bunun haritasıdır.

- **Uzmanlık dersleri:** 6.849 (computational origami / folding), 6.851 (ileri veri yapısı: van Emde Boas, fusion tree), 6.837 (computer graphics: ray casting, scene graph), 6.838 (geometri işleme: mesh, geodezik), 6.892 (recreational: oyun NP-hardness).
- **Geriye köprüler:** Dijkstra (Ders 19) mesh geodeziğinde, NP-hard/reduction (Ders 28) foldability ve redistricting'te, AVL (Ders 10) van Emde Boas karşılaştırmasında, BFS katmanları (Ders 13) Baker PTAS'ında karşına çıktı.

39.13 Anahtar Kavramlar (Cheat Sheet)

Hoca / Alan	Örnek	6.006 bağlantısı
Demaine — Origami (6.849)	design vs foldability; fold-and-cut	NP-hardness, reduction (Ders 28)
Demaine — Self-assembly	DNA tile, ikili sayaç	Geometrik hesaplama modeli, paralel
Demaine — İleri DS (6.851)	van Emde Boas log w, fusion tree	AVL (log n) sınırını aşma (Ders 10)
Demaine — Planar / Recreational	Baker PTAS; oyun NP-hard	BFS (Ders 13), approximation, hardness
Solomon — Mesh (6.838)	MMP geodezik (Dijkstra yetmez)	Dijkstra seviye kümeleri (Ders 19)
Solomon — Graphics (6.837)	ray casting; scene graph	$O(p \cdot n) \to$ ağaç $\log n$; DAG
Solomon — Redistricting	uniform sampling zor	Hamiltonian reduction, P≠NP (Ders 28)

39.14 Builder ve OMSCS Bağlantıları

7 köprü

Bu vitrin dersi, 6.006'nın altı aracının gerçek araştırma ve sistem mühendisliğine nasıl bağlandığını gösterir; köprülerin özeti:

1. **NP-hardness pratiği** → foldability, redistricting, oyun zorluğu: "bu problem verimli çözülemez" refleksi (OMSCS CS 6515).
2. **İleri veri yapısı (6.851)** → van Emde Boas/fusion tree; gerçek sistemlerde $\log n$'i kırma.
3. **Planar/approximation (PTAS)** → routing, harita, ağ optimizasyonu: "yapı varsa daha hızlı".
4. **Mesh geodezik (MMP/fast marching)** → robotik yol planlama, 3B tarama, oyun navigasyonu.
5. **Uzay-bölme ağaçları + scene graph (DAG)** → render, çarpışma tespiti, GPU; gerçek performans mühendisliği.

6. **Geometrik/self-assembly model** → DNA computing, moleküler nano-üretim farkındalığı.
7. **Redistricting/sampling** → MCMC, hesaplamalı sosyal bilim, hukuk+algoritma kesişimi.

❗ Tek bir şey alıp gideceksen

6.006'nın altı aracı — **NP-hardness, DP, çizge, veri yapısı, approximation, hesaplama modeli** — sınıfta kalmıyor; origami katlamada (design çözülebilir, foldability NP-hard), DNA self-assembly'de (geometrik hesaplama modeli), mesh üzerinde en kısa yolda (Dijkstra yetmez → MMP geodezik), ray casting'de ($O(p \cdot n)$ → uzay-bölme ağacı + scene graph DAG) ve politik gerrymandering'de (en iyi plan + örnekleme NP-hard) gerçek araştırma problemlerine dönüşüyor. Üç geometrici hocanın ortak mesajı tek: **algoritmalar her yerde, 6.006 kaçınılmaz.** Buradan 6.046 (CS 6515) ve uzmanlık derslerine.