WEB & YAZILIM GELİŞTİRME SERİSİ · MODÜL 6

Algoritma

Algoritmik düşünme, sözde kod ve akış şemaları; koşullar, döngüler ve mantık; diziler, yığın-kuyruk ve ağaçlar; arama, sıralama, özyineleme ve Big O. Özgün diyagramlar ve sözde kodla, dilden bağımsız.

Akış şeması · Veri yapıları · Big O · Eğitim amaçlıdır

Bu Kitap Hakkında

Bu modül, herhangi bir programlama dilinden bağımsız olan algoritmik düşünmeyi sıfırdan öğretir. Dört seviye ve yirmi altı bölüm boyunca algoritma kavramından sözde koda ve akış şemalarına, koşul-döngü-mantıktan veri yapılarına (dizi, yığın, kuyruk, sözlük, ağaç), arama-sıralama-özyinelemeden Big O ile verimliliğe ve problem çözme stratejilerine kadar her şeyi adım adım ele alır.

Her bölümde konuyu görselleştiren özgün bir diyagram (gerçek akış şemaları, ağaç grafikleri, yığın-kuyruk, Big O eğrileri), Türkçe sözde kod örnekleri, 'adım adım nasıl düşünürsün?' kartı ve bir alıştırma yer alır. Bu, on altı modüllük 'Web & Yazılım Geliştirme' serisinin altıncı modülüdür ve birinci fazı (web temelleri) tamamlar. Burada kazandığın düşünme biçimi, sonraki tüm dil ve sistem modüllerinde işine yarayacaktır. Bu seri eğitim amaçlıdır.

Web & Yazılım Geliştirme Serisi · Modül 6

İçindekiler

SEVİYE 1

Algoritmik Düşünme

Temeller: algoritma nedir, problemi anlamak ve bölmek, sözde kod, akış şemaları, girdi-işlem-çıktı, adım adım düşünmek, kalıpları görmek ve soyutlama.

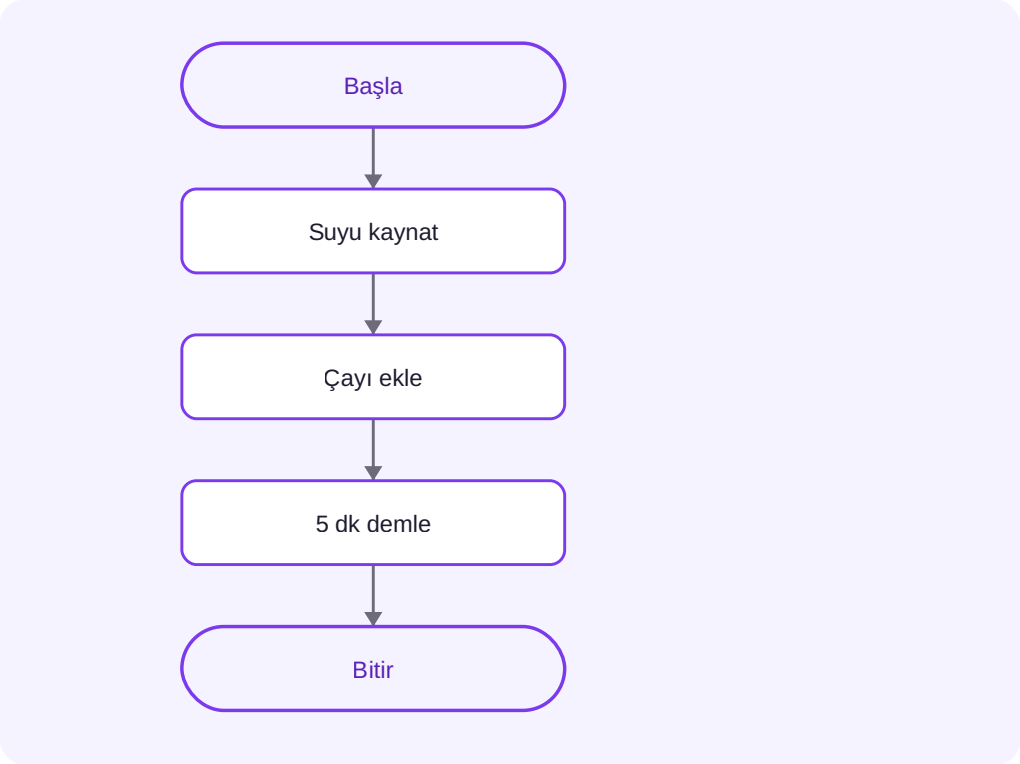
BÖLÜM 01

Algoritma Nedir?

Algoritma, bir işi yapmak için izlenen adımların net ve sıralı listesidir. Bir yemek tarifi, yol tarifi veya çay demleme adımları birer algoritmadır. Programlama, problemleri bu şekilde adımlara dökmektir.

Algoritmanın özellikleri

- **Net:** her adım açık ve anlaşılırdır.
- **Sıralı:** adımlar belirli bir düzende yapılır.
- **Sonlu:** bir başlangıcı ve sonu vardır.
- **Belirli bir çıktısı** vardır (işi tamamlar).



Şema 1.1 — Gündelik bir algoritma: çay demleme adımları.

Sözde kodla bir algoritma

```
ALGORİTMA: Çay demle
1. Suyu kaynat
2. Demliğe çay koy
3. Kaynar suyu ekle
4. 5 dakika bekle
5. Bardağa doldur
```

İPUCU

Programlamanın özü dil değil, **algoritmik düşünmedir**: bir problemi net adımlara bölebiliyorsan, onu herhangi bir dilde kodlayabilirsin. Bu modül, herhangi bir programlama dilinden bağımsızdır; öğrendiğin düşünme biçimi JavaScript'te de Python'da da işine yarayacak.

Adım adım nasıl düşünürsün? PLAN

Bir algoritma tasarlarken kendine sor: "Bu işi bilmeyen birine adım adım nasıl anlatırdım?" Cevabını sırayla, net cümlelerle yaz; işte algoritman bu.

Alıştırma 8 dk

Algoritma yaz:

1. Diş fırçalama ya da sandviç yapma adımlarını sırayla yaz.
2. Adımlardan birini atlarsan ne olur, düşün.
3. Algoritmanın 4 özelliğini kendi örneğinde göster.

BÖLÜM 02

Problemi Anlamak ve Bölmek

Büyük bir problem korkutucu görünebilir. Çözümün sırrı onu küçük, yönetilebilir parçalara bölmektir (decomposition). Her küçük parçayı çözmek kolaydır; sonra birleştirirsin.

Önce anla, sonra böl

- **Anla:** ne isteniyor? Girdi ne, çıktı ne?
- **Böl:** problemi alt problemlere ayır.
- **Çöz:** her alt problemi tek tek çöz.
- **Birleştir:** parçaları bir araya getir.

Şema 2.1 — Büyük problemi bölerek çözmek.

Bir problemi bölmek

```
PROBLEM: Bir alışveriş listesinin toplam tutarını bul
  alt problem 1: her ürünün fiyat × adet değerini hesapla
  alt problem 2: bu değerleri topla
  alt problem 3: sonucu göster
```

İPUCU

Bir problemi çözemiyorsan, genelde sebep onu yeterince **bölmemiş** olmandır. "Bunu daha küçük bir adıma indirebilir miyim?" diye sormaya devam et. En küçük adımlar o kadar basit olmalı ki, çözümü apaçık görünsün.

Adım adım nasıl düşünürsün? PLAN

Bir problemle karşılaşınca önce "tam olarak ne isteniyor?" diye netleştir, sonra "bunu hangi 2-3 küçük parçaya bölebilirim?" diye sor. Bu iki soru, çözümün yarısıdır.

Alıştırma 10 dk

Problemi böl:

1. "Bir sınıfın not ortalamasını bul" problemini alt problemlere böl.
2. Her alt problemi tek cümleyle yaz.
3. Hangi sırayla çözülmeleri gerektiğini belirt.

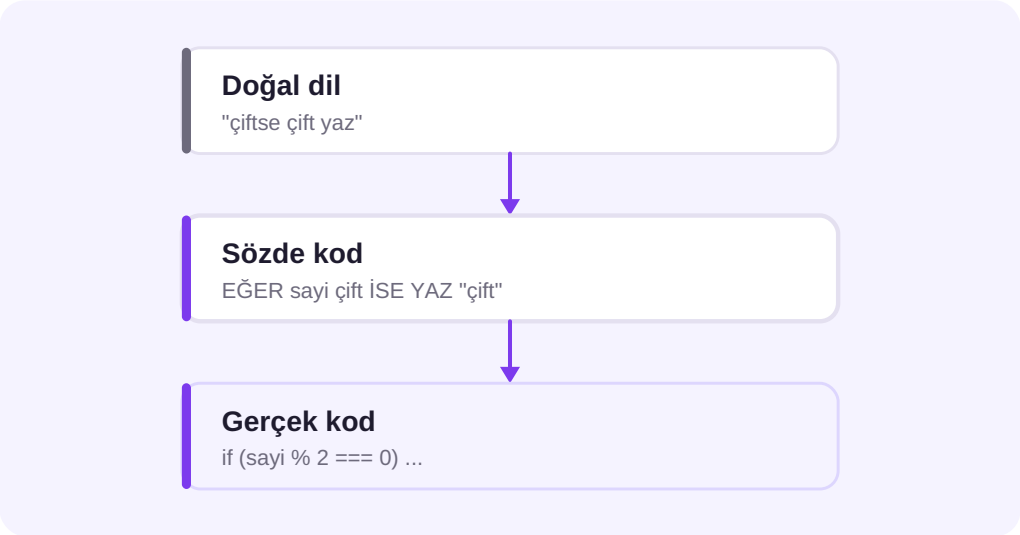
BÖLÜM 03

Sözde Kod (Pseudocode)

Sözde kod, bir algoritmayı gerçek bir programlama diline çevirmeden, sade ve yapılandırılmış cümlelerle yazmaktır. Mantığa odaklanmanı sağlar; söz dizimi (noktalı virgül, parantez) derdi olmadan.

Doğal dilden koda köprü

- Günlük dile yakın ama yapılandırılmış (EĞER, TEKRARLA, YAZ).
- Herhangi bir dile kolayca çevrilebilir.
- Mantığı planlamanın en hızlı yolu.



Şema 3.1 — Sözde kod, fikir ile gerçek kod arasında köprüdür.

Sözde kod örneği

```
 GİRDİ: bir sayı al
EĞER sayı 0'dan büyük İSE
    YAZ "pozitif"
DEĞİLSE EĞER sayı 0'dan küçük İSE
    YAZ "negatif"
DEĞİLSE
    YAZ "sıfır"
```

İPUCU

Kod yazmaya başlamadan önce **sözde kodla planla**; profesyoneller bile karmaşık bir işe önce kâğıt üstünde sözde kodla başlar. Mantığı netleştirdikten sonra gerçek koda çevirmek kolaydır ve çok daha az hata yaparsın. Sözde kodun "doğru" tek bir biçimi yoktur — yeter ki net olsun.

Adım adım nasıl düşünürsün? PLAN

Bir algoritmayı önce sözde kodla yaz: her satır tek bir adımı anlatsın, koşulları EĞER/DEĞİLSE, tekrarları TEKRARLA ile belirt. Mantık oturduğunda gerçek koda çevirmek mekanik bir iş olur.

Alıştırma 10 dk

Sözde kod yaz:

1. "Bir sayının çift mi tek mi olduğunu söyle" algoritmasını sözde kodla yaz.
2. EĞER / DEĞİLSE yapısını kullan.
3. Sözde kodunu bir arkadaşına okutup anlaşılır mı, test et.

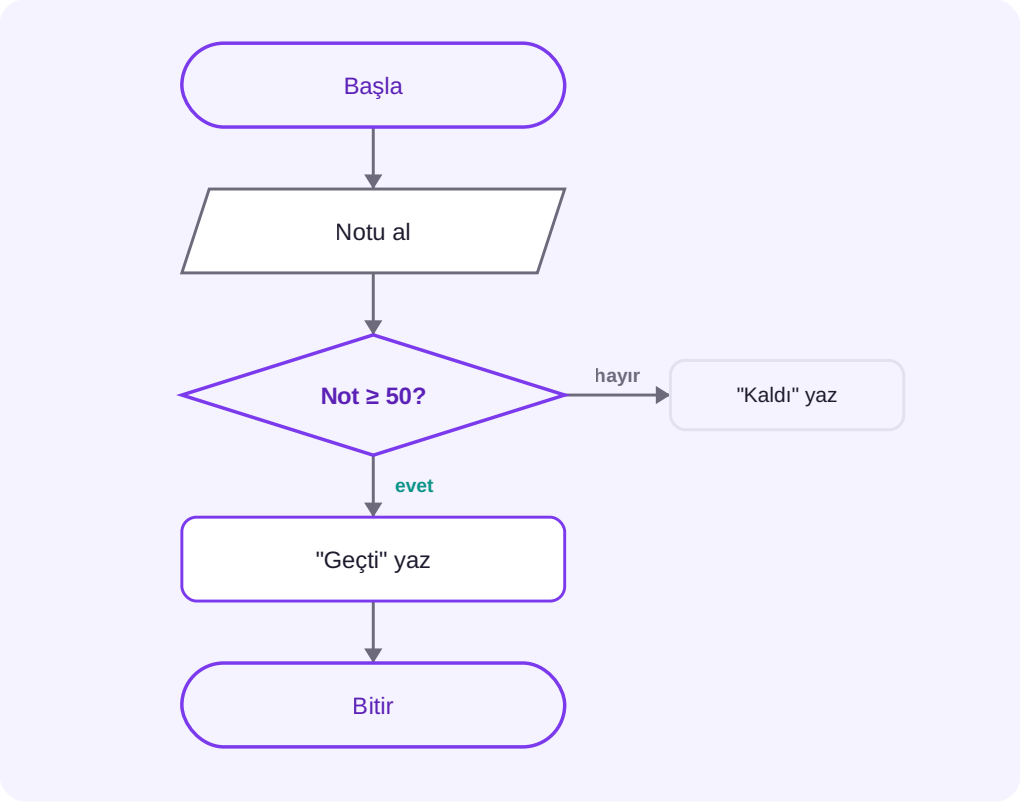
BÖLÜM 04

Akış Şemaları (Flowcharts)

Akış şeması, bir algoritmayı şekillerle görselleştirir. Her şekil bir işlem türünü temsil eder; oklar akışı gösterir. Karmaşık mantığı bir bakışta anlamanın güçlü bir yoludur.

Akış şeması sembolleri

- **Oval:** başlangıç / bitiş.
- **Paralelkenar:** girdi / çıktı (al / yaz).
- **Dikdörtgen:** işlem (bir adım).
- **Eşkenar dörtgen:** karar (evet / hayır).



Şema 4.1 — Tüm sembollerle bir akış şeması: başla, girdi, karar, işlem, bitir.

İPUCU

Karar (eşkenar dörtgen) şekli akış şemalarının kalbidir: bir koşul sorar ve akışı "evet" ile "hayır" yollarına ayırır. Bir algoritmayı kodlamadan önce akış şemasını çizmek, mantık hatalarını (eksik durum, yanlış sıra) kâğıt üstünde görmeni sağlar.

Adım adım nasıl düşünürsün? PLAN

Bir akış şeması çizerken yukarıdan aşağı düşün: nereden başlıyor, hangi kararlar veriliyor, her yol nereye gidiyor ve nerede bitiyor? Karar şekillerinden çıkan her okun bir etiketi (evet/hayır) olmalı.

Alıştırma 12 dk

Akış şeması çiz:

1. "Hava yağmurlu mu? Şemsiye al" kararını bir akış şemasıyla çiz.
2. Bir karar (eşkenar dörtgen) ve iki yol kullan.
3. Başlangıç ve bitişi oval ile göster.

BÖLÜM 05

Girdi, İşlem, Çıktı

Hemen hemen her program üç parçadan oluşur: bir girdi alır, üzerinde bir işlem yapar ve bir çıktı üretir. Bu basit model (GİÇ / IPO), bir algoritmayı düşünmenin temel çerçevesidir.

Üç temel parça

- **Girdi:** programa giren veri (kullanıcıdan, dosyadan...).
- **İşlem:** veriyle yapılan hesap/dönüşüm.
- **Çıktı:** üretilen sonuç (ekran, dosya...).

Şema 5.1 — Girdi → İşlem → Çıktı modeli.

GİÇ modeliyle düşünmek

```
 GİRDI:  kullanıcıdan bir sıcaklık (Celsius) al
İŞLEM:  fahrenheit = celsius × 9 / 5 + 32
ÇIKTI:  fahrenheit değerini yazdır
```

İPUCU

Bir probleme başlarken önce **girdi ve çıktıyı netleştir**: "Elimde ne var (girdi), ne üretmem gerekiyor (çıktı)?" Bu iki ucu bilince, ortadaki "işlem" kısmını tasarlamak çok daha kolaylaşır. Çoğu hata, girdi veya çıktının yanlış anlaşılmasından doğar.

Adım adım nasıl düşünürsün? PLAN

Her algoritmayı GİÇ olarak düşün: "Bu işlevin girdisi ne, çıktısı ne, aradaki işlem ne?" Bu üç soruya net cevap verebiliyorsan, çözümü kodlamaya hazırsın demektir.

Alıştırma 8 dk

GİÇ ile düşün:

1. Bir dikdörtgenin alanını bulan algoritmanın girdi, işlem ve çıktısını yaz.
2. Aynısını "bir metni büyük harfe çevir" için yap.
3. Hangi adımın işlem, hangisinin çıktı olduğunu ayırt et.

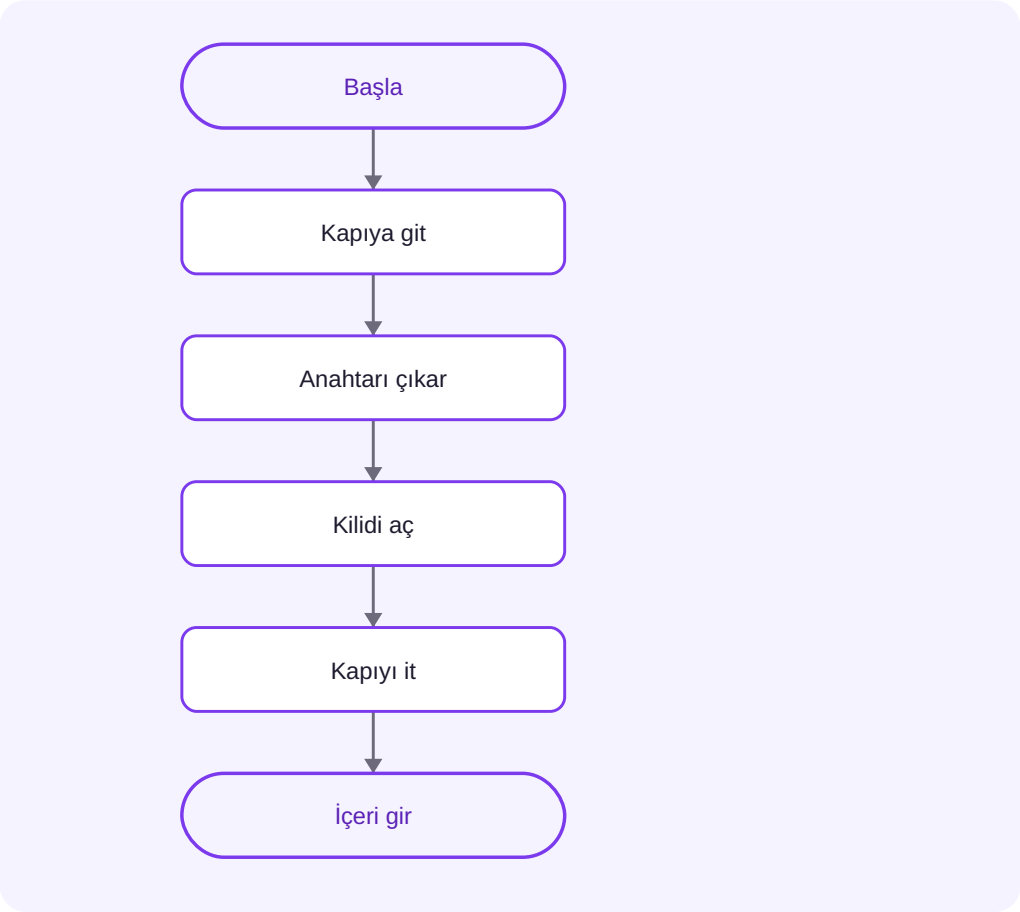
BÖLÜM 06

Adım Adım Düşünmek

Bilgisayar tam olarak söyleneni, söylenen sırayla yapar — ne eksik ne fazla. Bu yüzden adımların sırası ve eksiksizliği kritiktir. İyi bir algoritmacı, hiçbir adımı varsaymaz.

Sıralama her şeydir

- Adımlar mantıklı bir sırada olmalı (önce ne, sonra ne?).
- Hiçbir adım atlanmamalı (bilgisayar varsaymaz).
- Her adım tek ve net bir iş yapmalı.



Şema 6.1 — Doğru sıra olmazsa algoritma çalışmaz.

İPUCU

Bir algoritmayı test etmenin en iyi yolu, onu **bir bilgisayar gibi, harfiyen** takip etmektir (buna "kuru çalıştırma" / dry run denir): her adımı tam söylendiği gibi uygula, hiçbir şey varsayma. Böylece eksik veya yanlış sıralı adımları hemen yakalarsın.

Adım adım nasıl düşünürsün? PLAN

Algoritmanı yazdıktan sonra, kâğıt üstünde adım adım "oyna": her satırı bilgisayar gibi uygula. Bir yerde takılıyorsan veya bir adımı varsaymak zorunda kalıyorsan, orada eksik bir adım vardır.

Alıştırma 10 dk

Sıralamayı test et:

1. "Çay yap" algoritmanın adımlarını karıştır, sonra doğru sıraya diz.
2. Bir adımı çıkarınca ne bozuluyor, gözle.
3. Kendi bir algoritmanı kuru çalıştırmayla test et.

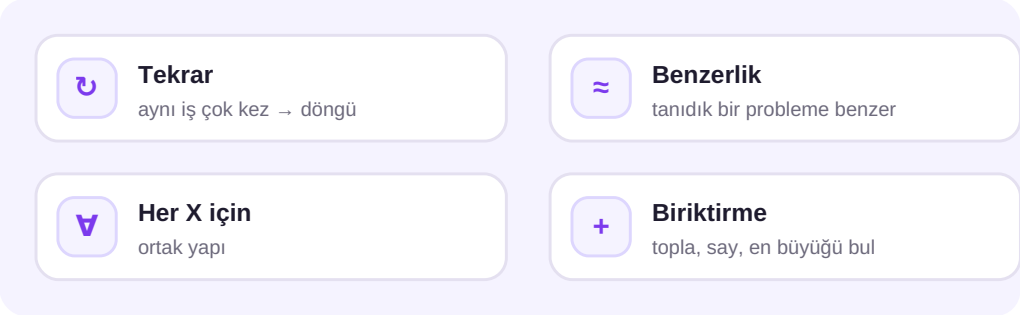
BÖLÜM 07

Kalıpları Görmek (Pattern Recognition)

İyi programcılar, farklı problemlerdeki ortak kalıpları görür. Bir kez "bu, daha önce çözdüğüm şeye benziyor" dediğinde, çözüm çok daha hızlı gelir. Kalıp tanıma, deneyimle güçlenen bir beceridir.

Hangi kalıpları ararsın?

- **Tekrar:** aynı işin birçok kez yapılması (→ döngü).
- **Benzerlik:** bu problem daha önce çözdüğüme benziyor mu?
- **Ortak yapı:** "her X için Y yap" kalıbı.



Şema 7.1 — Algoritmalarda sık görülen kalıplar.

"Biriktirme" kalıbı

```
// Bir listedeki sayıların toplamı — yaygın bir kalıp
toplam = 0
HER sayı İÇİN listede:
    toplam = toplam + sayı
YAZ toplam
```

İPUCU

Birçok problem aslında birkaç temel kalıbın birleşimidir: **biriktirme** (topla/say), **arama** (bul), **süzme** (koşulu sağlayanları seç), **dönüştürme** (her öğeyi değiştir). Bu kalıpları tanıdıkça, yeni problemler "yeni" olmaktan çıkar; tanıdık parçalara ayrılır.

Adım adım nasıl düşünürsün? PLAN

Yeni bir problemle karşılaşınca sor: "Bunda bir tekrar var mı? Daha önce çözdüğüm neye benziyor? Hangi temel kalıbı (topla/bul/süz) kullanabilirim?" Kalıbı görmek, çözümün büyük kısmını hazır getirir.

Alıştırma

10 dk

Kalıp bul:

1. "Bir listedeki en büyük sayıyı bul" probleminde hangi kalıp var?
2. "Çift sayıları say" probleminde hangi kalıplar birleşiyor?
3. Günlük hayatından tekrar eden bir işi (kalıbı) yaz.

BÖLÜM 08

Soyutlama (Abstraction)

Soyutlama, gereksiz ayrıntıları gizleyip yalnızca önemli olana odaklanmaktır. Bir arabayı kullanmak için motorun nasıl çalıştığını bilmen gerekmez; sadece direksiyon ve pedalları bilirsin. Programlamada da karmaşıklığı böyle yönetiriz.

Ayrıntıyı gizlemek

- Karmaşık bir işi tek bir isimle "paketle" (örn. fonksiyon).
- Kullanırken iç ayrıntıları düşünmek zorunda kalma.
- Genelleştir: özel bir çözümü birçok duruma uyarla.

Şema 8.1 — Soyutlama: karmaşıklığı bir isim ardına gizlemek.

Soyutlama: bir işi paketlemek

```
 // Ayrıntılar bir kez yazılır, ismi tekrar tekrar kullanılır
FONKSİYON ortalamaBul(sayılar):
    toplam = sayıların toplamı
    DÖNDÜR toplam / sayıların adedi

YAZ ortalamaBul(notlar)   // ayrıntıyı düşünmeden kullan
```

İPUCU

Soyutlama, büyük programları yönetilebilir kılan en güçlü fikirdir: bir işi bir kez doğru çöz, ona anlamlı bir isim ver (fonksiyon) ve sonra o ismi düşünmeden kullan. Fonksiyonlar, sınıflar, kütüphaneler — hepsi soyutlamanın araçlarıdır. İyi soyutlama, kodu hem kısaltır hem anlaşılır kılar.

Adım adım nasıl düşünürsün? PLAN

Bir işi birçok yerde tekrarladığını fark edince, onu soyutla: ayrıntıları tek bir fonksiyona koy, anlamlı bir isim ver ve artık o ismi kullan. Böylece "ne yaptığına" odaklanır, "nasıl yaptığını" gizlersin.

Alıştırma 10 dk

Soyutla:

1. Tekrar eden bir işi (örn. "bir kullanıcıyı selamla") tek bir fonksiyon adıyla paketle.
2. Fonksiyonun girdisini ve çıktısını belirle.
3. Aynı fonksiyonu farklı girdilerle "kullanmayı" yaz.

SEVİYE 2

Mantık ve Kontrol Akışı

Algoritmalara karar ve tekrar yeteneği: koşullu mantık, döngüler, değişkenler ve sayaçlar, iç içe yapılar, mantıksal ifadeler ve kenar durumlar.

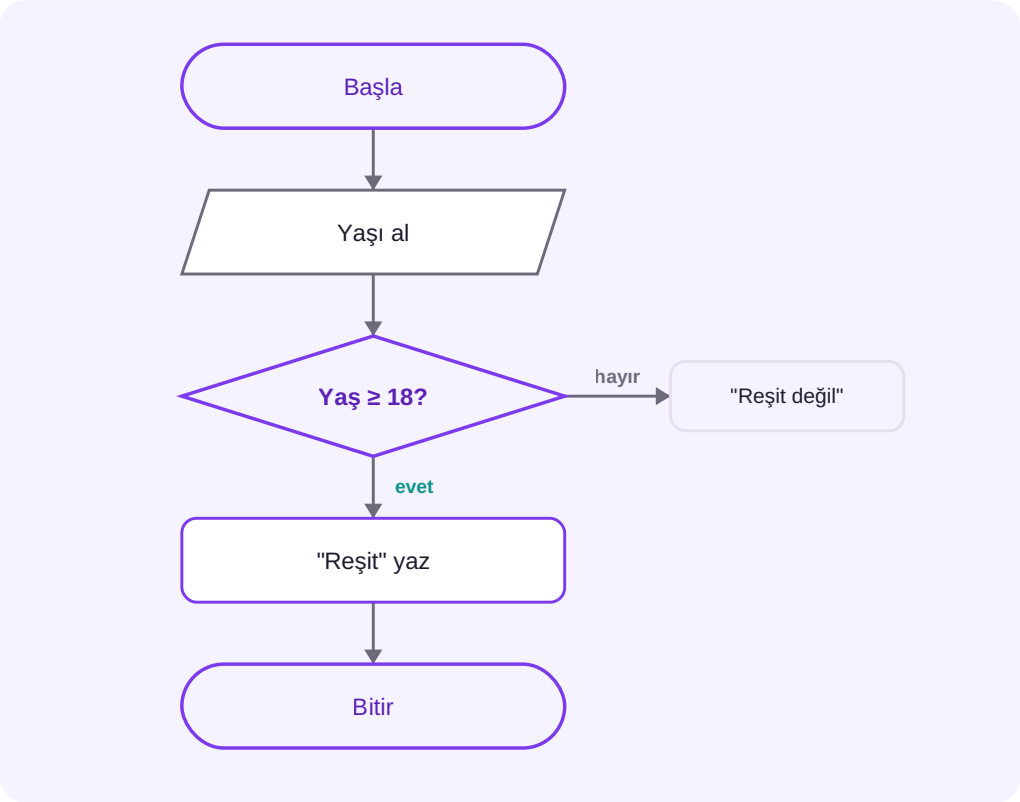
BÖLÜM 09

Koşullu Mantık

Algoritmalar karar verebilmelidir. Koşullu mantık, "eğer şu doğruysa bunu yap, değilse şunu yap" diyerek akışı koşula göre yönlendirir. Bu, programlara "zekâ" katan temel yapıdır.

Karar yapısı

- Bir **koşul** doğru (true) ya da yanlış (false) olur.
- Doğruysa bir yol, yanlışsa başka bir yol izlenir.
- Birden çok durum için koşullar zincirlenir (DEĞİLSE EĞER).



Şema 9.1 — Koşullu mantık: karar akışı evet/hayır olarak ayrılır.

Sözde kod: koşullu karar

```
 GİRDİ: not
EĞER not >= 50 İSE
    YAZ "Geçti"
DEĞİLSE
    YAZ "Kaldı"
```

İPUCU

Koşulları tasarlarken **tüm durumları** düşün: "ya tam sınırdaysa? (örn. not = 50)", "ya hiçbir koşul tutmazsa?". Eksik bir durum, gizli bir hatadır. Karmaşık koşulları akış şemasıyla çizmek, atladığın yolları görmeni sağlar.

Adım adım nasıl düşünürsün? PLAN

Bir karar yazmadan önce sor: "Hangi durumlar mümkün? Her durumda ne olmalı? Sınır değerinde (≥ mi, > mü?) ne olur?" Bütün yolları listele; sonra her birini bir koşulla ele al.

Alıştırma 10 dk

Karar tasarla:

1. Bir sayının pozitif/negatif/sıfır olduğunu söyleyen sözde kod yaz.
2. Üç durumu da (DEĞİLSE EĞER ile) ele al.
3. Sınır durumu (sıfır) doğru çalışıyor mu, kontrol et.

BÖLÜM 10

Döngüler ve Tekrar

Aynı işi defalarca yapmak (bir listeyi gezmek, 1'den 100'e saymak) için döngüler kullanılır. Döngü, "aynı kodu yüz kez yazma" derdini ortadan kaldırır ve algoritmaları kısa, güçlü kılar.

Döngü mantığı

- Bir koşul doğru olduğu sürece gövde tekrarlanır.
- Her turda bir şey değişmeli (yoksa sonsuz döngü).
- "Belli sayıda" (sayaçlı) ya da "koşula bağlı" tekrar.

Şema 10.1 — Döngü: koşul doğru oldukça gövde tekrarlanır.

Sözde kod: sayaçlı döngü

```
 i = 1
TEKRARLA i 1'den 5'e kadar:
    YAZ i
    i = i + 1
// çıktı: 1 2 3 4 5
```

İPUCU

Döngülerin en sık hatası **sonsuz döngüdür**: koşul hiç yanlışa dönmezse program kilitlenir. Her döngüde "bu koşul nasıl ve ne zaman yanlış olacak?" diye kendine sor. Sayaçlı döngüler (1'den N'e) bu hataya daha az açıktır, çünkü sonları bellidir.

Adım adım nasıl düşünürsün? PLAN

Bir döngü tasarlarken üç şeyi netleştir: **başlangıç** (sayaç nereden başlar), **koşul** (ne zamana kadar sürer) ve **değişim** (her turda ne değişir). Bu üçü doğruysa döngün de doğru çalışır.

Alıştırma 10 dk

Döngü kur:

1. 1'den 10'a kadar sayan bir döngü sözde kodu yaz.
2. Yalnızca çift sayıları yazdıracak şekilde değiştir.
3. Döngünün nasıl sonlandığını (koşulun ne zaman yanlış olduğunu) açıkla.

BÖLÜM 11

Değişkenler ve Sayaçlar

Algoritmalar çoğu zaman bir şeyi "takip eder": şimdiye kadarki toplam, en büyük sayı, kaç tane bulunduğu. Bunun için değişkenleri sayaç ve biriktirici olarak kullanırız.

Durumu takip etmek

- **Sayaç:** bir şeyi sayar (her turda +1).
- **Biriktirici:** değerleri toplar/biriktirir.
- **En iyi/en kötü:** şimdiye kadarki en büyüğü/küçüğü tutar.

Şema 11.1 — Biriktirici kalıbı: döngü boyunca değer toplanır.

Sözde kod: en büyüğü bul

```
 enBuyuk = listenin ilk öğesi
HER sayı İÇİN listede:
    EĞER sayı > enBuyuk İSE
        enBuyuk = sayı
YAZ enBuyuk
```

İPUCU

Sayaç/biriktirici değişkenlerini **döngüden önce** doğru değerle başlat: toplam için 0, çarpım için 1, "en büyük" için listenin ilk öğesi (veya çok küçük bir değer). Yanlış başlangıç, en sık yapılan algoritma hatalarındandır. Başlangıcı düşünmek, çözümün yarısıdır.

Adım adım nasıl düşünürsün? PLAN

Bir şeyi takip etmen gerektiğinde sor: "Hangi değişkeni, hangi başlangıç değeriyle tutmalıyım? Her turda nasıl güncellenecek? Döngü bitince bana ne söyleyecek?" Bu kalıp (başlat → güncelle → sonucu oku) sayısız problemde işine yarar.

Alıştırma 10 dk

Takip et:

1. Bir listedeki sayıların toplamını bulan sözde kod yaz (biriktirici).
2. Kaç tane çift sayı olduğunu sayan bir sayaç ekle.
3. En küçük sayıyı bulan mantığı yaz (başlangıcı doğru seç).

BÖLÜM 12

İç İçe Yapılar

Koşullar ve döngüler iç içe geçebilir: bir döngünün içinde başka bir döngü, bir koşulun içinde başka bir koşul. Bu, daha karmaşık problemleri çözmeni sağlar — ama dikkatli olmayı gerektirir.

İç içe döngü ve koşul

- Bir **iç içe döngü**, dış döngünün her turunda baştan sona çalışır.
- Örnek: bir tablonun her satırının her hücresini gezmek.
- İç içe koşullar, çok aşamalı kararlar kurar.

Şema 12.1 — İç içe döngü: dış döngünün her turunda iç döngü tümüyle çalışır.

Sözde kod: çarpım tablosu

```
HER i İÇİN 1'den 3'e:
   HER j İÇİN 1'den 3'e:
       YAZ i × j
   // iç döngü biter, dış döngü ilerler
```

İPUCU

İç içe döngülerde **çalışma sayısı çarpılır**: 100'lük bir dış döngü, içinde 100'lük bir iç döngüyle 10.000 işlem yapar. Bu, verimlilik (Big O) açısından önemlidir — Seviye 4'te göreceksin. İç içe yapıları çok derinleştirmek (3-4 kat) kodu hem yavaş hem anlaşılmaz kılar; mümkünse sadeleştir.

Adım adım nasıl düşünürsün? PLAN

İç içe bir yapı kurarken sor: "Dış adım her döndüğünde, iç adım kaç kez çalışacak? Toplamda kaç işlem olur?" Bu sayıyı bilmek, hem doğruluğu hem hızı kontrol etmeni sağlar.

Alıştırma 12 dk

İç içe kur:

1. 1-3 çarpım tablosunu iç içe döngüyle sözde kod olarak yaz.
2. Toplam kaç çarpım yapıldığını hesapla.
3. Dış ve iç döngünün hangi sırayla çalıştığını adım adım izle.

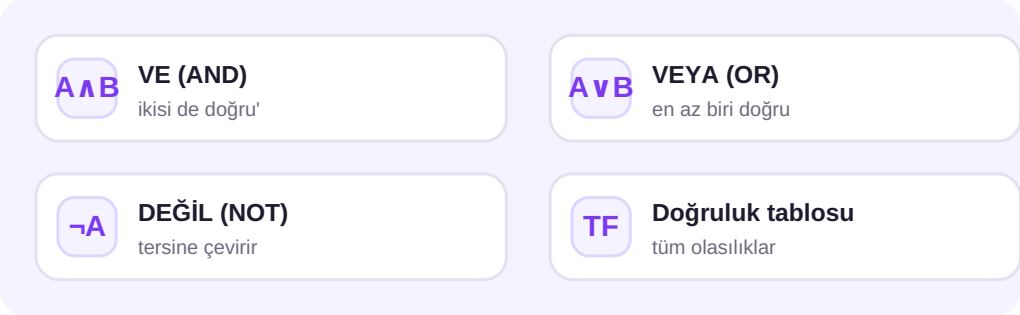
BÖLÜM 13

Mantıksal İfadeler

Karmaşık kararlar, birden çok koşulu birleştirmeyi gerektirir: "yaş 18'den büyük VE üye İSE". VE, VEYA, DEĞİL operatörleri (Boole mantığı), koşulları güçlü biçimde birleştirir.

VE, VEYA, DEĞİL

- **VE (AND):** her iki koşul da doğruysa doğru.
- **VEYA (OR):** en az biri doğruysa doğru.
- **DEĞİL (NOT):** doğruyu yanlışa, yanlışı doğruya çevirir.



Şema 13.1 — Boole (mantık) operatörleri.

Sözde kod: birleşik koşul

```
EĞER yas >= 18 VE uye = doğru İSE
    YAZ "Giriş izni var"
EĞER DEĞİL(uye) İSE
    YAZ "Önce üye ol"
```

İPUCU

Mantıksal hataların çoğu VE/VEYA karışıklığından doğar: "18 yaşından büyük **ve** küçük" gibi bir koşul asla doğru olamaz (bir sayı ikisi birden olamaz). Karmaşık koşulları parçalara ayır, her parçanın ne zaman doğru olduğunu tek tek düşün. Gerektiğinde doğruluk tablosu çiz: tüm doğru/yanlış kombinasyonlarını listele.

Adım adım nasıl düşünürsün? PLAN

Birleşik bir koşul kurarken sor: "Bu koşulun doğru olması için tam olarak ne gerekiyor — hepsi mi (VE), herhangi biri mi (VEYA)?" Şüphedeysen, birkaç örnek değer (yaş=17, üye=evet gibi) deneyip koşulun doğru sonucu verdiğini kontrol et.

Alıştırma 10 dk

Mantık kur:

1. "Hafta içi VE sabahsa: işe git" koşulunu sözde kodla yaz.
2. VEYA kullanarak "cumartesi VEYA pazar ise: dinlen" yaz.
3. Bir koşulu DEĞİL ile tersine çevir.

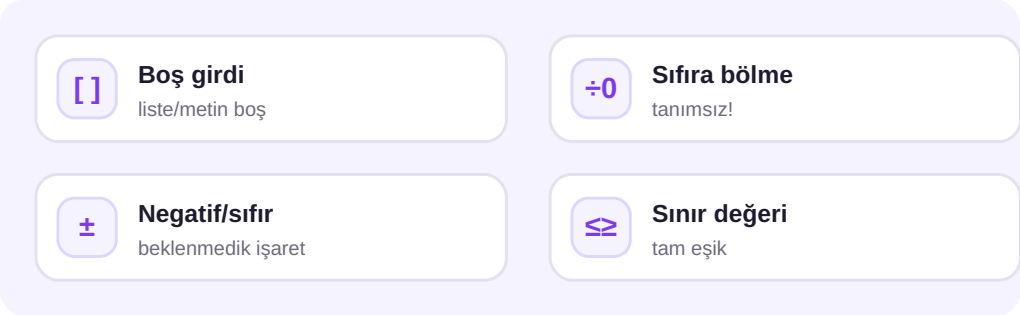
BÖLÜM 14

Hata Durumları ve Kenar Durumlar

İyi bir algoritma yalnızca "normal" durumda değil, sıra dışı durumlarda da doğru çalışmalıdır: boş liste, sıfır, negatif sayı, çok büyük değer. Bu "kenar durumları" öngörmek, sağlam kodun işaretidir.

Sık karşılaşılan kenar durumlar

- **Boş girdi:** liste boşsa ne olur?
- **Sıfır / negatif:** bölme veya sayma hataları.
- **Sınır değerleri:** tam eşik noktası (örn. = 18).
- **Çok büyük / çok küçük:** beklenmedik aşırı değerler.



Şema 14.1 — Öngörülmesi gereken kenar durumlar.

Sözde kod: kenar durumu ele al

```
 GİRDİ: sayı listesi
EĞER liste boş İSE
    YAZ "Liste boş, ortalama hesaplanamaz"
    DUR
// ... normal hesap buradan sonra
```

İPUCU

Bir algoritma yazdıktan sonra kendine sor: "Ya girdi boşsa? Ya sıfırsa? Ya negatifse? Ya tek bir öğe varsa?" Bu "ya ...?" sorularının her biri bir kenar durumudur. Profesyonel kodu amatörden ayıran en büyük fark, bu durumların öngörülüp nazikçe ele alınmasıdır — çökmek yerine anlamlı bir cevap vermek.

Adım adım nasıl düşünürsün? PLAN

Çözümünü tasarlarken normal durumun yanında "uç" örnekleri de düşün: en küçük, en büyük, boş, sıfır, negatif. Her biri için algoritman ne yapmalı? Bunları baştan planlarsan, kodun gerçek dünyada çökmek yerine doğru davranır.

Alıştırma 10 dk

Kenar durumları bul:

1. "Ortalama bul" algoritması için 3 kenar durumu listele.
2. Boş liste durumunu sözde kodda ele al.
3. "Bir sayıyı böl" işleminde sıfıra bölmeyi nasıl önlersin, yaz.

SEVİYE 3

Veri Yapıları

Veriyi doğru biçimde tutmak: veri yapısı nedir, diziler ve listeler, yığın ve kuyruk, sözlük (hash map), ağaçlar ve doğru yapıyı seçmek.

BÖLÜM 15

Veri Yapısı Nedir?

Veri yapısı, veriyi bilgisayarda düzenleme biçimidir. Doğru yapı, veriye hızlı erişmeni ve işini kolayca yapmanı sağlar; yanlış yapı ise her şeyi yavaşlatır. "Hangi yapı?" sorusu, iyi yazılımın temelidir.

Neden önemli?

- Aynı veri, farklı yapılarda farklı hızlarda işlenir.
- Doğru yapı: kod hem hızlı hem anlaşılır olur.
- Her yapının güçlü ve zayıf yanları vardır.

Şema 15.1 — Doğru veri yapısı, veriyi işlemeyi hızlandırır.

İPUCU

Bir benzetme: aynı eşyaları bir çöp poşetine de, etiketli çekmecelere de koyabilirsin. İkisi de "saklar" ama bir şey ararken çekmeceler çok daha hızlıdır. Veri yapıları, verinin "etiketli çekmeceleridir". Bu seviyede en yaygın yapıları ve hangi işe yaradıklarını öğreneceksin.

Adım adım nasıl düşünürsün? PLAN

Bir problemde veriyi tutman gerektiğinde sor: "Bu veriye nasıl erişeceğim — sırayla mı, anahtarla mı, son ekleneni mi? En sık hangi işlemi yapacağım — arama mı, ekleme mi, sıralama mı?" Cevap, doğru yapıyı işaret eder.

Alıştırma

8 dk

Yapıyı düşün:

1. Bir telefon rehberini saklamak için nasıl bir yapı düşünürsün?
2. "Sıraya girmiş insanlar" hangi yapıya benzer?
3. Doğru yapının neden hız kazandırdığını bir örnekle açıkla.

BÖLÜM 16

Diziler ve Listeler

Dizi (array) ve liste, en temel veri yapısıdır: sıralı bir öğe koleksiyonu. Her öğeye bir indeks (sıra numarası) ile erişilir. Çoğu problem dizilerle başlar.

Dizinin özellikleri

- Öğeler sıralıdır; indeks 0'dan başlar.
- İndeksle erişim çok hızlıdır (doğrudan).
- Sona ekleme kolay; ortaya ekleme/çıkarma daha yavaş.

Şema 16.1 — Dizi: indeksli, sıralı öğeler.

Sözde kod: dizide arama

```
 GİRDİ: sayilar dizisi, aranan
HER i İÇİN 0'dan dizinin sonuna:
    EĞER sayilar[i] = aranan İSE
        YAZ "Bulundu, indeks:", i
```

İPUCU

Dizinin en güçlü yanı **indeksle doğrudan erişimdir**: `sayilar[1000]`'e ulaşmak, `sayilar[0]` kadar hızlıdır. En zayıf yanı, ortadan ekleme/çıkarmadır (sonraki tüm öğeler kaymak zorunda kalır). Sıralı veri ve indeksle erişim gerektiğinde dizi ilk tercihtir.

Adım adım nasıl düşünürsün? PLAN

Bir diziyle çalışırken indeksin 0'dan başladığını ve son indeksin "uzunluk − 1" olduğunu unutma. Bir öğe ararken tüm diziyi gezmen gerekebilir; ama yerini (indeksini) biliyorsan erişim anında olur.

Alıştırma

10 dk

Diziyle çalış:

1. Bir dizideki bir değeri arayan sözde kod yaz.
2. Dizideki tüm öğelerin toplamını hesapla.
3. İlk ve son öğeye indeksle nasıl eriştiğini göster.

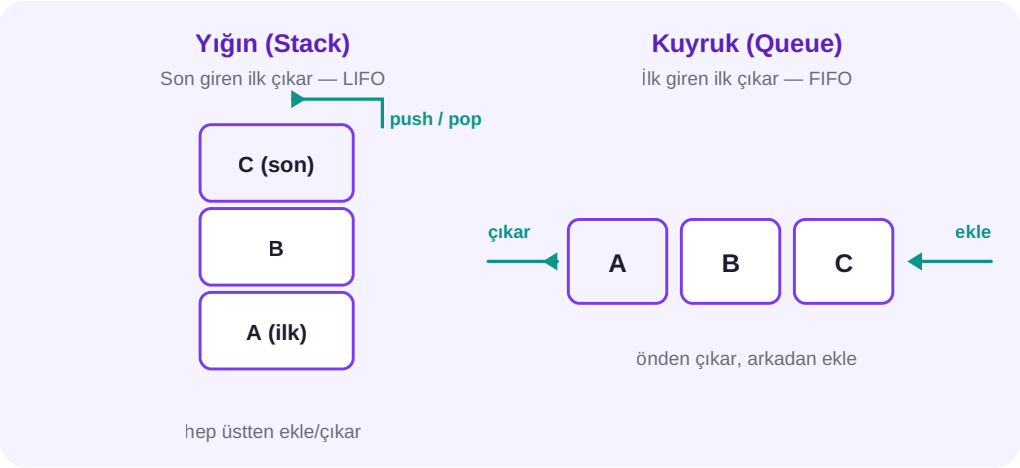
BÖLÜM 17

Yığın ve Kuyruk (Stack & Queue)

Yığın ve kuyruk, öğelerin hangi sırayla çıkacağını belirleyen iki temel yapıdır. Yığın "son giren ilk çıkar", kuyruk ise "ilk giren ilk çıkar" mantığıyla çalışır. İkisi de günlük hayattan tanıdıktır.

İki farklı sıra mantığı

- **Yığın (Stack):** LIFO — son eklenen ilk çıkar (tabak yığını gibi).
- **Kuyruk (Queue):** FIFO — ilk eklenen ilk çıkar (market sırası gibi).
- Yığın: geri alma (undo); Kuyruk: işleri sırayla işleme.



Şema 17.1 — Yığın (LIFO) ve Kuyruk (FIFO).

Sözde kod: yığın kullanımı

```
 YIĞIN olustur
yigin.ekle("A")   // push
yigin.ekle("B")
yigin.cikar()     // pop -> "B" (son giren)
```

İPUCU

Doğru mantığı seçmek kritiktir: bir "geri al" (undo) özelliği **yığın** ister (en son işlemi geri al); yazıcıya gönderilen belgeler ise **kuyruk** ister (ilk gönderilen ilk yazılır). "Son giren mi, ilk giren mi önce çıkmalı?" sorusu, hangisini kullanacağını söyler.

Adım adım nasıl düşünürsün? PLAN

Bir sıralama problemi gördüğünde sor: "Öğeler hangi sırayla işlenmeli? En son ekleneni mi (yığın), yoksa en önce ekleneni mi (kuyruk) önce almalıyım?" Bu tek soru, doğru yapıyı seçtirir.

Alıştırma 10 dk

LIFO/FIFO ayır:

1. Tarayıcının "geri" düğmesi yığın mı kuyruk mu, gerekçesiyle yaz.
2. Bir çağrı merkezi sırası hangisidir?
3. Bir yığına 3 öğe ekleyip çıkarınca hangi sırayla çıkar, göster.

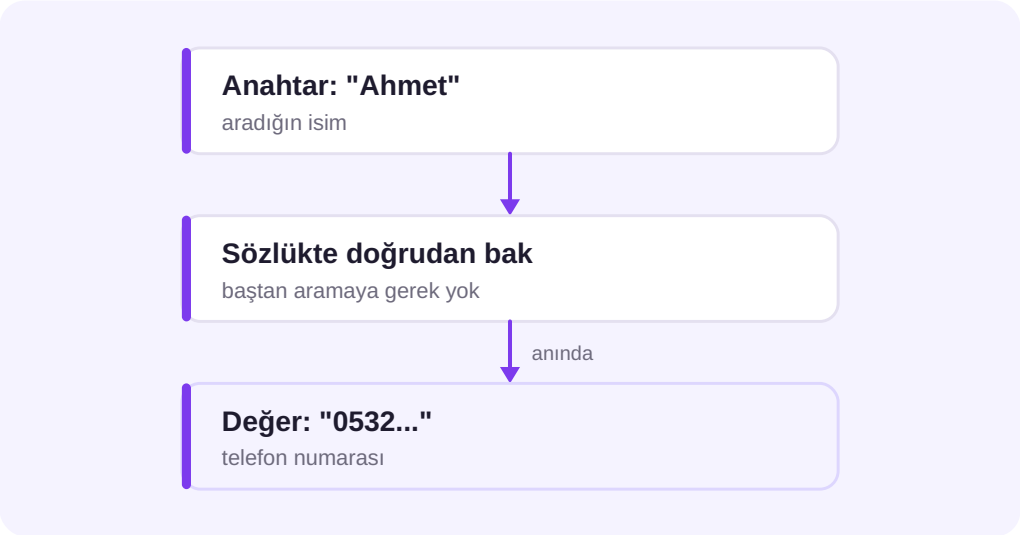
BÖLÜM 18

Sözlük / Eşleme (Hash Map)

Sözlük (hash map / dictionary), her değeri bir anahtarla eşleştirir ve o anahtarla değere neredeyse anında erişmeni sağlar. Bir gerçek sözlük gibi: kelimeyi (anahtar) ara, anlamını (değer) bul.

Anahtar → değer

- Her öğe bir **anahtar-değer** çiftidir.
- Anahtarla erişim çok hızlıdır (aramaya gerek yok).
- Anahtarlar benzersizdir; "telefon rehberi" gibi düşün.



Şema 18.1 — Sözlük: anahtarla değere doğrudan erişim.

Sözde kod: sözlük kullanımı

```
 rehber = { "Ahmet": "0532...", "Ayşe": "0543..." }
YAZ rehber["Ahmet"]      // "0532..." anında
rehber["Veli"] = "0505..."  // yeni çift ekle
```

İPUCU

Sözlüğün süper gücü **hızlı aramadır**: bir dizide "Ahmet"i bulmak için tüm listeyi gezmen gerekebilir; sözlükte anahtarla doğrudan ulaşırsın. "Bir şeyi bir kimlikle/isimle hızlıca bulmam gerekiyor" dediğin her yerde sözlük düşün: kullanıcı id'leri, kelime sayımı, önbellek...

Adım adım nasıl düşünürsün? PLAN

Bir problemde "X'e karşılık gelen Y'yi hızlıca bulmam lazım" diyorsan, sözlük doğru yapıdır. Anahtarı (X) ve değeri (Y) belirle; gerisini sözlük halleder — arama yapmadan, doğrudan erişimle.

Alıştırma 10 dk

Sözlük kur:

1. Ürün adlarını fiyatlarıyla eşleyen bir sözlük tasarla.
2. Bir ürünün fiyatına anahtarla nasıl eriştiğini yaz.
3. Bir metindeki kelime sayımının neden sözlüğe uygun olduğunu açıkla.

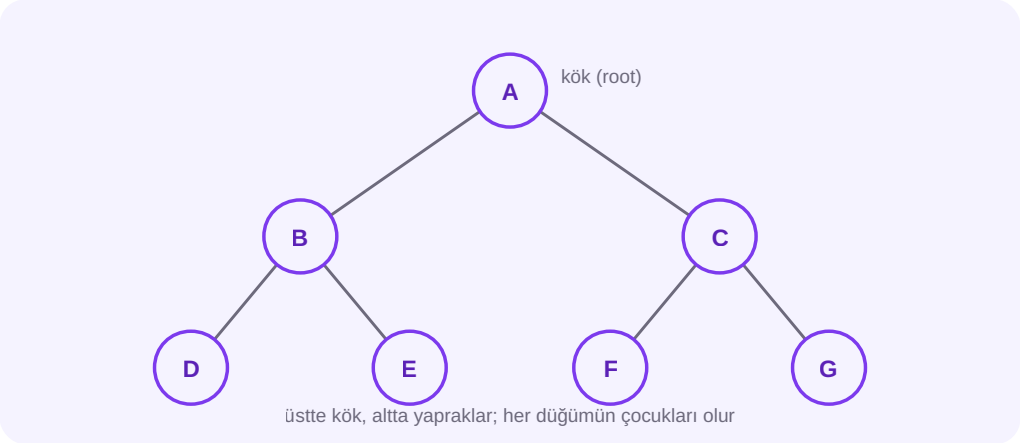
BÖLÜM 19

Ağaçlar (Trees)

Ağaç, hiyerarşik (dallı) veriyi temsil eder: bir kök, ondan dallanan düğümler ve en uçta yapraklar. Dosya sistemleri, organizasyon şemaları ve HTML'in DOM'u birer ağaçtır.

Ağaç kavramları

- **Kök (root):** en üstteki düğüm.
- **Düğüm (node):** her veri noktası.
- **Çocuk / ebeveyn:** düğümler arası bağ.
- **Yaprak (leaf):** çocuğu olmayan düğüm.



Şema 19.1 — Ağaç: kökten yapraklara hiyerarşik yapı.

Ağaç nerelerde?

```
 // Dosya sistemi bir ağaçtır:
Belgeler/            (kök)
  Resimler/          (dal)
    tatil.jpg        (yaprak)
  rapor.docx         (yaprak)
```

İPUCU

Bir veri "içinde başka şeyler barındıran" şeylerden oluşuyorsa (klasör içinde klasör, yorum altında yorum, kategori içinde alt kategori), bu büyük olasılıkla bir **ağaçtır**. Ağaçlar, hiyerarşiyi doğal biçimde temsil eder; HTML modülünde gördüğün DOM ağacı buna iyi bir örnektir.

Adım adım nasıl düşünürsün? PLAN

Veride bir hiyerarşi (üst–alt ilişkisi) görüyorsan ağaç düşün: neyin "kök", neyin "dal", neyin "yaprak" olduğunu belirle. Her düğümün bir ebeveyni (kök hariç) ve sıfır veya daha fazla çocuğu vardır.

Alıştırma 10 dk

Ağaç çiz:

1. Ailenin bir bölümünü (büyükanne → ebeveyn → çocuk) ağaç olarak çiz.
2. Kök, dal ve yaprakları işaretle.
3. Bilgisayarındaki bir klasör yapısının neden ağaç olduğunu açıkla.

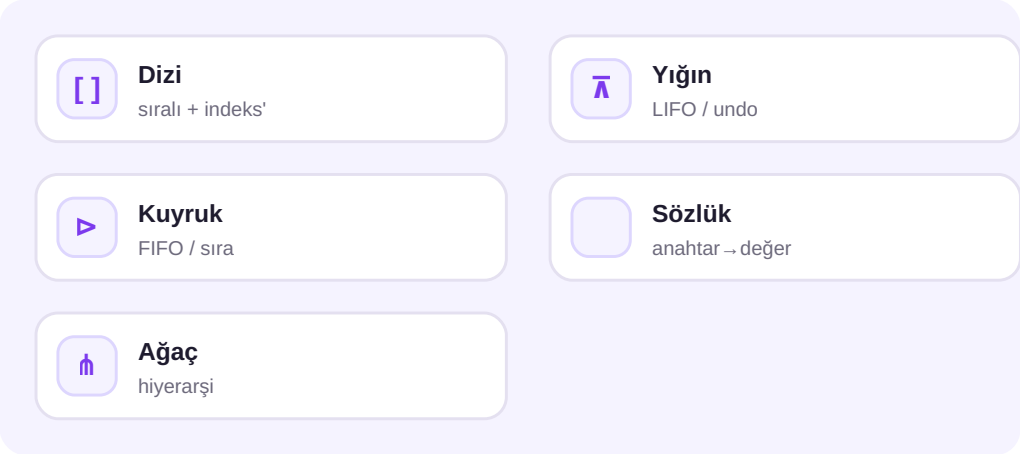
BÖLÜM 20

Veri Yapısı Seçimi

Artık birkaç veri yapısı tanıyorsun. Asıl beceri, bir problem için doğru olanı seçmektir. Bu seçim, çözümün ne kadar hızlı ve temiz olacağını belirler.

Hangi durumda hangisi?

- **Sıralı liste, indeksle erişim** → Dizi.
- **Son giren ilk çıkmalı (undo)** → Yığın.
- **İlk giren ilk çıkmalı (sıra)** → Kuyruk.
- **Anahtarla hızlı erişim** → Sözlük.
- **Hiyerarşik veri** → Ağaç.



Şema 20.1 — İhtiyaca göre veri yapısı seçimi.

İPUCU

Doğru yapıyı seçmek için "en sık hangi işlemi yapacağım?" sorusuna odaklan. Çok arama yapacaksan sözlük; sırayla işleyeceksen kuyruk; indeksle gezmek için dizi. Yanlış yapı seçmek programı çalıştırır ama yavaşlatır; doğru yapı ise hem hızı hem okunabilirliği getirir.

Adım adım nasıl düşünürsün? PLAN

Bir problemde veriyi nasıl tutacağına karar verirken: "Erişim deseni ne (indeks/anahtar/sıra)? En sık işlem ne (ara/ekle/çıkar)? Hiyerarşi var mı?" sorularını sırayla yanıtla — cevaplar doğru yapıyı işaret eder.

Alıştırma

10 dk

Yapı seç:

1. "Geri al" özelliği için hangi yapı? Neden?
2. "Kullanıcı id'sinden bilgi bulma" için hangi yapı?
3. "Bir klasör ağacı" için hangi yapı? Gerekçelerini yaz.

SEVİYE 4

Algoritma Teknikleri

Klasik teknikler ve verimlilik: arama algoritmaları, sıralama, özyineleme, Big O ile verimlilik, problem çözme stratejileri ve baştan sona bir problem çözmek.

BÖLÜM 21

Arama Algoritmaları

Bir veride bir öğeyi bulmak en sık yapılan işlerden biridir. İki temel yöntem vardır: baştan sona tek tek bakan doğrusal arama ve sıralı veride çok daha hızlı çalışan ikili arama.

Doğrusal ve ikili arama

- **Doğrusal arama:** baştan sona her öğeye bak. Basit ama yavaş.
- **İkili arama:** sıralı veride ortadan böl, yarısını ele. Çok hızlı.
- İkili arama yalnızca **sıralı** veride çalışır.

Şema 21.1 — Doğrusal arama her öğeye bakar; ikili arama her adımda yarıyı eler.

Sözde kod: ikili arama

```
 sol = 0, sag = son indeks
TEKRARLA sol <= sag iken:
    orta = (sol + sag) / 2
    EĞER dizi[orta] = aranan İSE  YAZ "bulundu", DUR
    EĞER dizi[orta] < aranan İSE  sol = orta + 1
    DEĞİLSE  sag = orta - 1
```

İPUCU

İkili aramanın gücü çarpıcıdır: 1.000.000 öğeli sıralı bir listede, doğrusal arama en kötü durumda 1.000.000 adım atarken ikili arama yalnızca ~20 adımda bulur (her adımda kalan yarıyı eler). Bu, sıralı veride neden ikili aramanın tercih edildiğini gösterir — ama önce verinin sıralı olması gerekir.

Adım adım nasıl düşünürsün? PLAN

Bir arama yapman gerektiğinde sor: "Veri sıralı mı?" Sıralıysa ikili arama (her adımda yarıyı eleyerek) muazzam hız kazandırır. Değilse ya önce sırala ya da doğrusal aramayı kullan. Veri boyutu büyüdükçe bu seçim çok önemli hale gelir.

Alıştırma 12 dk

Arama yap:

1. 1-100 arası bir sayıyı doğrusal aramayla en kötü kaç adımda bulursun?
2. Aynısını ikili aramayla kaç adımda bulursun (yaklaşık)?
3. İkili aramanın neden sıralı veri gerektirdiğini açıkla.

BÖLÜM 22

Sıralama Algoritmaları

Veriyi sıralamak (küçükten büyüğe, A'dan Z'ye) en klasik algoritma problemidir. Sıralamanın temel fikrini anlamak, hem ikili arama gibi tekniklerin kapısını açar hem de algoritmik düşünceyi güçlendirir.

Sıralamanın temel fikri

- Öğeleri karşılaştır ve gerekirse yer değiştir.
- Bunu, hepsi doğru sıraya girene kadar tekrarla.
- Basit yöntemler (kabarcık) yavaş; gelişmişler (merge/quick) hızlıdır.

Şema 22.1 — Sıralamanın özü: karşılaştır, yer değiştir, tekrarla.

Sözde kod: kabarcık sıralama (fikir)

```
TEKRARLA dizi sıralanana kadar:
   HER komşu çift (a, b) İÇİN:
       EĞER a > b İSE
           a ile b'yi yer değiştir
```

İPUCU

Kabarcık sıralama anlaması en kolay yöntemdir ama yavaştır ($O(n^2)$). Gerçek projelerde dilin **hazır sıralama fonksiyonunu** kullanırsın (genelde $O(n \log n)$'lik gelişmiş bir algoritma). Yine de temel fikri (karşılaştır–değiştir–tekrarla) bilmek, sıralamanın neden bir maliyeti olduğunu ve verimliliğin neden önemli olduğunu anlamanı sağlar.

Adım adım nasıl düşünürsün? PLAN

Sıralama gerektiğinde pratikte çoğu zaman dilin hazır `sırala()` fonksiyonunu çağırırsın. Ama "nasıl çalışıyor?" diye merak edersen, fikir hep aynıdır: öğeleri karşılaştırıp doğru sıraya gelene dek yer değiştirmek.

Alıştırma 12 dk

Sırala:

1. [3, 1, 2] dizisini kabarcık mantığıyla elle sırala, adımları yaz.
2. Kaç karşılaştırma ve kaç yer değiştirme yaptığını say.
3. Neden büyük verilerde gelişmiş sıralama gerektiğini açıkla.

BÖLÜM 23

Özyineleme (Recursion)

Özyineleme, bir fonksiyonun kendi kendini çağırarak problemi gittikçe küçülen aynı tipte alt problemlere bölmesidir. İlk başta sihir gibi görünür; ama "büyük problemi, kendinin küçük bir kopyasına indirgemek" fikrini kavrayınca güçlü bir araç olur.

İki olmazsa olmaz

- **Taban durum:** özyinelemenin durduğu en basit hal.
- **Özyineli adım:** problemi küçülterek kendini çağırma.
- Taban durum olmazsa: sonsuz döngü (çöker).

Şema 23.1 — Özyineleme: problemi küçülterek kendini çağırır, taban durumda durur.

Sözde kod: faktöriyel

```
FONKSİYON faktoriyel(n):
   EĞER n <= 1 İSE  DÖNDÜR 1      // taban durum
   DÖNDÜR n × faktoriyel(n - 1)    // özyineli adım
```

İPUCU

Her özyinelemeli çözümün bir **taban durumu** (durma koşulu) olmalı; yoksa fonksiyon kendini sonsuza kadar çağırır ve çöker. Özyineleme, doğası gereği "kendine benzeyen" problemlerde (ağaçları gezme, faktöriyel, bölerek çözme) çok zariftir. Aynı işi döngüyle de yapabilirsin; özyineleme bazı problemlerde çok daha okunur bir çözüm sunar.

Adım adım nasıl düşünürsün? PLAN

Bir problemi özyinelemeyle çözerken iki soruyu yanıtla: "Bu problemi nasıl daha küçük, aynı tipte bir probleme indirgerim? (özyineli adım)" ve "En basit hâl nedir, nerede dururum? (taban durum)". Bu ikisi varsa, özyineleme kendiliğinden işler.

Alıştırma 12 dk

Özyinele:

1. Faktöriyel(4)'ün adımlarını elle aç (4×3×2×1).
2. Taban durumu ve özyineli adımı işaretle.
3. Taban durum olmasa ne olurdu, açıkla.

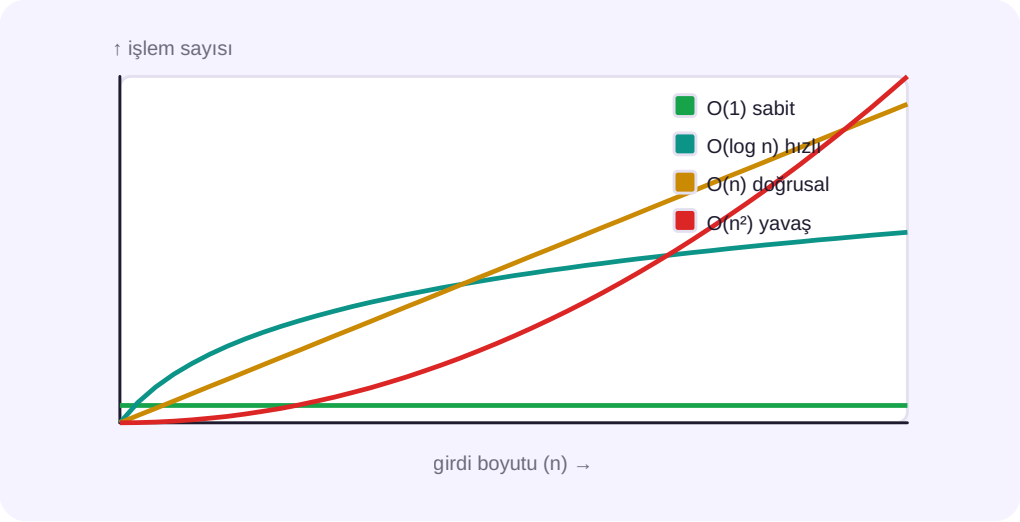
BÖLÜM 24

Verimlilik ve Big O

İki algoritma aynı işi yapabilir ama biri çok daha hızlı olabilir. Big O gösterimi, bir algoritmanın girdi büyüdükçe ne kadar yavaşladığını kabaca ölçer. "Bu yeterince hızlı mı?" sorusunun ortak dilidir.

Big O ne anlatır?

- Girdi (n) büyüdükçe işlem sayısı nasıl artıyor?
- **O(1):** sabit — girdiden bağımsız (en iyi).
- **O(log n):** çok yavaş artar (ikili arama).
- **O(n):** doğrusal (diziyi bir kez gezmek).
- **O(n²):** kareyle artar (iç içe döngü — dikkat!).



Şema 24.1 — Big O: girdi büyüdükçe işlem sayısının artış hızı.

İPUCU

Big O'da sabitler ve küçük ayrıntılar göz ardı edilir; önemli olan **büyüme hızıdır**. n=10 için O(n) ve O(n²) arasında pek fark yokken, n=1.000.000 için O(n) bir milyon, O(n²) bir trilyon işlem demektir — biri saniyeler, diğeri günler sürebilir. İç içe döngülerin (O(n²)) büyük veride tehlikeli olmasının sebebi budur.

Adım adım nasıl düşünürsün? PLAN

Bir algoritmanın hızını düşünürken sor: "Girdiyi iki katına çıkarırsam, iş kaç kat artar? Aynı mı kalır (O(1)), iki kat mı (O(n)), dört kat mı (O(n²))?" Bu soru, çözümünün büyük veride ayakta kalıp kalmayacağını söyler.

Alıştırma

12 dk

Verimliliği ölç:

1. Bir diziyi bir kez gezen döngü hangi Big O'dur?
2. İç içe iki döngü hangi Big O'dur?
3. n 1000 katına çıkınca O(n) ve O(n^2) işlemin nasıl değiştiğini karşılaştır.

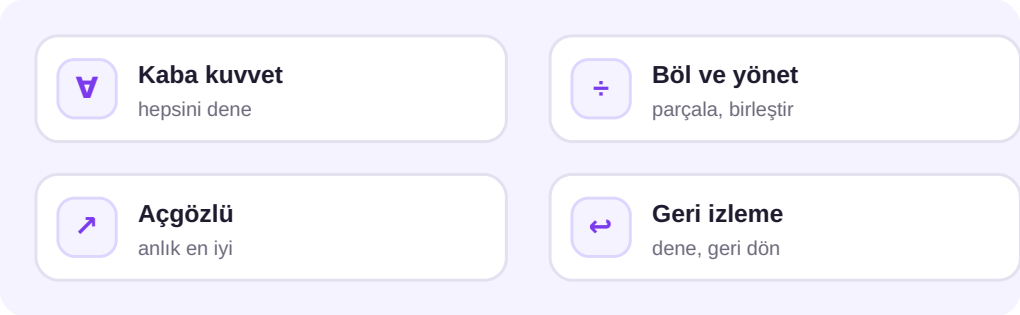
BÖLÜM 25

Problem Çözme Stratejileri

Deneyimli programcılar, problem türüne göre farklı stratejiler uygular. Bu yaklaşımları tanımak, yeni bir problemle karşılaştığında "nereden başlamalı?" sorusuna hazır cevaplar verir.

Yaygın stratejiler

- **Kaba kuvvet:** tüm olasılıkları dene. Basit ama bazen yavaş.
- **Böl ve yönet:** problemi parçala, parçaları çöz, birleştir.
- **Açgözlü:** her adımda o anki en iyi seçimi yap.
- **Geri izleme:** dene, çıkmaza girersen geri dön, başka yol dene.



Şema 25.1 — Yaygın problem çözme stratejileri.

İPUCU

Bir probleme önce **en basit yaklaşımla** (genelde kaba kuvvet) başla; çalışan bir çözüm, mükemmel ama bitmemiş bir çözümden iyidir. Çalıştıktan sonra "daha verimli olabilir mi?" diye sor ve gerekirse böl-yönet gibi daha akıllı bir stratejiye geç. Önce doğru, sonra hızlı — bu sıra önemlidir.

Adım adım nasıl düşünürsün? PLAN

Yeni bir problemde takılırsan strateji listene bak: "Tüm olasılıkları deneyebilir miyim (kaba kuvvet)? Parçalara bölebilir miyim (böl-yönet)? Her adımda yerel en iyiyi seçmek işe yarar mı (açgözlü)?" Bir strateji, çözüme giden yolu açar.

Alıştırma

10 dk

Strateji seç:

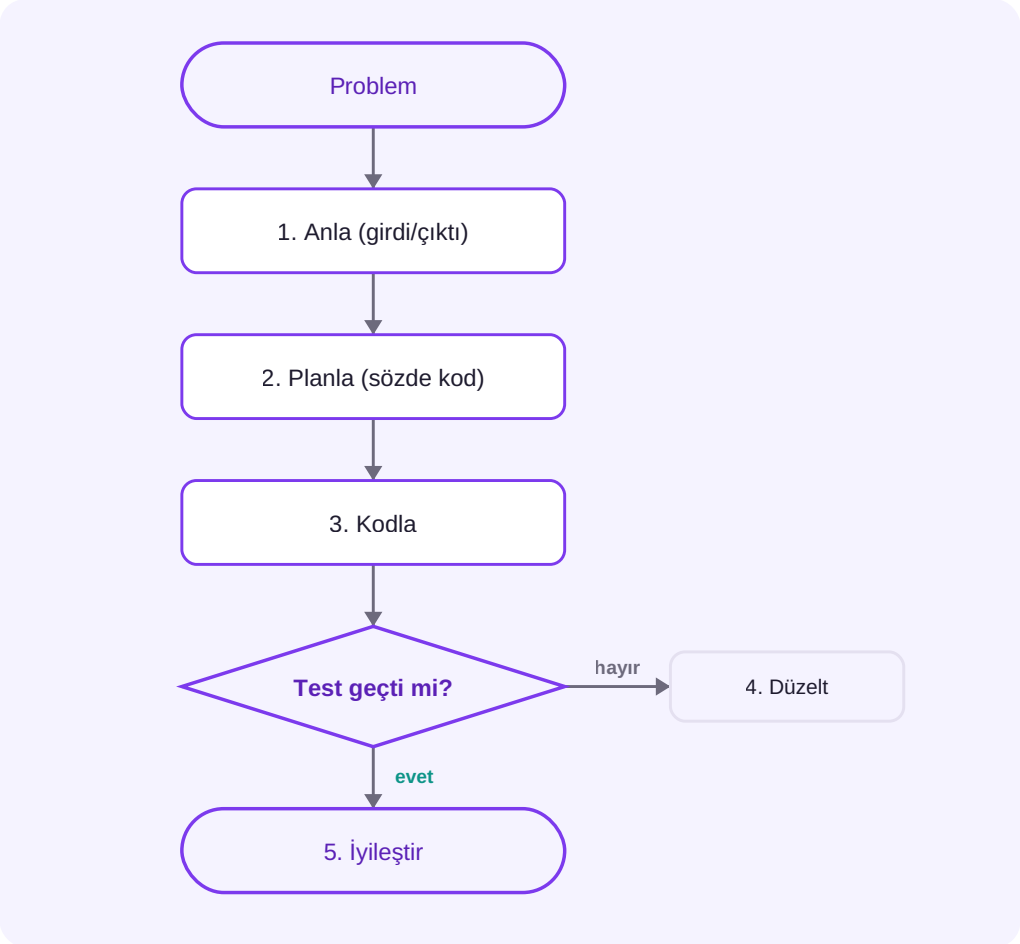
1. "Bozuk para ile en az parça üstü verme" hangi stratejiye uyar?
2. Bir labirentten çıkış hangi stratejiye benzer?
3. Bir problemi önce kaba kuvvetle, sonra daha akıllıca nasıl çözeceğini yaz.

BÖLÜM 26

Bitirme: Bir Problemi Baştan Sona Çözmek

Tüm öğrendiklerini birleştirip bir problemi profesyonelce, baştan sona çözüyorsun. Anla, planla, kodla, test et, iyileştir — bu döngü, her boyuttaki problemde işine yarayacak.

Problem çözme akışı



Şema 26.1 — Profesyonel problem çözme döngüsü.

Örnek: "en sık geçen kelimeyi bul"

```
 // 1. ANLA: girdi = metin, çıktı = en çok geçen kelime
// 2. PLANLA (sözde kod):
sayac = boş sözlük
HER kelime İÇİN metinde:
    sayac[kelime] = sayac[kelime] + 1
enSik = sayac içinde en büyük değere sahip anahtar
YAZ enSik
```

İPUCU

Bu beş adımlı döngü (anla → planla → kodla → test et → iyileştir) profesyonellerin günlük yöntemidir. En sık atlanan adım **"anla"dır**; oysa yanlış anlaşılan bir problemi mükemmel kodlamak işe yaramaz. Önce ne istendiğini netleştir, sonra sözde kodla planla — kod yazmak en kolay kısımdır. Sıradaki modülde (Arkayüz & API) bu düşünceyi gerçek sistemlere uygulayacaksın.

Adım adım nasıl düşünürsün? PLAN

Bir problemi çözerken bu sırayı izle: önce girdi/çıktıyı **anla**, sözde kodla **planla**, sonra **kodla**, kenar durumlarla **test et**, çalışınca **iyileştir**. Takıldığında hangi adımda olduğunu bilmek, çözüme dönmeni sağlar.

Alıştırma 20 dk

Baştan sona çöz:

1. "Bir listedeki tekrar eden ilk öğeyi bul" problemini seç.
2. Girdi/çıktıyı yaz (anla), sonra sözde kodla planla.
3. Hangi veri yapısının (örn. sözlük/küme) işe yarayacağını belirle.
4. Kenar durumları (boş liste, hiç tekrar yok) düşün ve ele al.

EK

Algoritma Terimleri Sözlüğü

En sık kullanılan algoritma ve problem çözme terimleri. Bir başvuru kaynağı olarak saklayabilirsin.

Algoritma	`Sıralı, net adımlar`	Sözde kod	`Sade dille plan`
Akış şeması	`Şekillerle akış`	Girdi/İşlem/Çıktı	`GİÇ modeli`
Decomposition	`Problemi bölme`	Döngü	`Tekrar yapısı`
Koşul	`EĞER / DEĞİLSE`	Dizi / Liste	`Sıralı veri`
Yığın / Kuyruk	`LIFO / FIFO`	Özyineleme	`Kendini çağırma`
Big O	`Verimlilik ölçüsü`	İkili arama	`Bölerek arama`

Algoritmik düşünmenin özeti PLAN

Algoritmik düşünme dört adımdır: **problemi anla** (girdi/çıktı), **parçalara böl** (decomposition), **kalıpları gör** ve **çözümü adımlara dök** (sözde kod / akış şeması). Bu beceri herhangi bir dilden bağımsızdır; bir kez kazanınca, JavaScript'ten Python'a her dilde aynı netlikle problem çözersin.