Giriş, Big-O ve İşaretçiler

CEN207 Veri Yapıları — Hafta 1

Dr. Öğr. Üyesi Uğur CORUH · 2026-2027 Güz

Bugünün planı (3 saat)

Saat	Konu
1	Ders planı · veri yapısı nedir · doğrusal / doğrusal olmayan
2	Big-O: arama, büyüme, döngüler, alan **Animasyon 1–5**
3	İşaretçiler, yığın/öbek, TLV/PER **Animasyon 6–13** · C atölyesi **Animasyon 14**

Öğrenme çıktıları: LO.1 (doğrusal/doğrusal olmayan yapılar) · LO.2 (zaman ve alan karmaşıklığı) · LO.7 (doğru yapıyı seçmek)

Bu haftanın kavramları — nerede

Kavram	Nerede
Veri yapısı, doğrusal / doğrusal olmayan	Bölüm 1–2
Big-O, büyüme, en iyi/en kötü/ortalama, alan	Bölüm 3
İşaretçiler, struct'lar	Bölüm 4
Yığın, öbek, TLV/PER, C atölyesi	Bölüm 5–7

Kod örnekleri nasıl çalışıyor

- Her fikrin eksiksiz bir **C** ve **Java** programı var
- C: `gcc -std=c11 -Wall -Wextra -o /tmp/x file.c && /tmp/x`
- Java: `javac -d /tmp/j File.java && java -cp /tmp/j File`
- Kaynaklar: `code/week-01/c/` ve `code/week-01/java/`
- Her programın beklenen çıktısı hafta notlarında

Ders düzeni: nasıl değerlendirileceksiniz

- **Tek bir proje**, tüm dönem boyunca: önce C, sonra Java
- **İki ara kontrol**: C'de bir vize, Java'da bir final
- **İki quiz** — ayrı bir ödev yok
- Bunun gibi haftalık notlar, İngilizce ve Türkçe
- Tam not dağılımı: ders **izlencesi**

Ders düzeni: takımlar ve kaynaklar

- Takım büyüklüğü: proje takımı başına **en çok 3** öğrenci
- **3. haftadan sonra takım değişikliği yok**
- Proje kılavuzu: `docs/project-guide/`
- Ön koşullar: `docs/prerequisites/` (C araç zinciri, alt yapı)
- Eski bir izlence PDF'i dersle çelişiyorsa: **sorun, tahmin etmeyin**

Bu haftanın — ve dersin — haritası

Bu hafta ne kuruyor	Sonraki her hafta ne ekliyor
Maliyeti ölçmenin yolu (Big-O)	Veriyi düzenlemenin bir yolu daha
Belleğin bir resmi (işaretçiler)	Aynı Big-O aracıyla değerlendiriliyor

1. Veri Yapısı Nedir?

Başlangıç sorusu

Telefonunuzun rehberi on binlerce isim arasında "Ayşe"yi anında bulur. Sırasız olsaydı, aynı telefon her ismi taramak zorunda kalırdı.

Kısa bir tarihçe

- **1960'lar** — "veri yapısı" terimi bilgisayar bilimlerinde yaygınlaşır
- **1968** — Knuth'un *TAOCP* Cilt 1'i liste, yığın, ağaçları inceler
- **1974** — Liskov & Zilles: **soyut veri tipi** (ADT)
- ADT = yapının *ne yaptığı*, *nasıl kurulduğu* değil

Sezgi: veri için mobilya

- Dosya dolabı, kitaplık, masadaki tepsi yığını
- Her biri aynı *türde* içeriği tutar
- Her biri **farklı şeylerde** iyidir
- Tek bir "en iyi" düzen yok — yalnızca ödünleşimler

Veri yapısı, tam olarak

Verileri öyle düzenlemenin bir yolu ki belirli işlemler — erişim, arama, ekleme, silme — bilinen, çözümlenebilir bir **maliyetle** yapılabilsin.

Neden tek bir tane yok

İşlem	Neden bedava değil
Erişim	Bazı düzenler konumu hesaplar; bazıları baştan yürür
Arama	Sırasız: tek tek kontrol. Sıralı: yarıya indirilebilir
Ekleme / Silme	Bazıları her şeyi kaydırır; bazıları iki işaretçiyi bağlar

Mini soru

Her şeyde diğer her yapıyı geride bırakan
tek bir "en iyi" veri yapısı neden yok?

Yanıt

Her düzen bir **ödünleşim** yapar: erişimi hızlandıran (örn. sıralı dizi), genelde eklemeyi yavaşlatır — ve tersi. İstisnasız.

2. Doğrusal ve Doğrusal Olmayan: Dersin Haritası

Başlangıç sorusu

Son beş şarkınız tek bir çizgi oluşturur.
Bir şirketin org şeması dallanır. Bu yalnızca
çizimle mi ilgili, yoksa gerçek bir yapısal fark mı?

Tanımlar

- **Doğrusal**: her elemanın tam olarak bir "sonraki"si var
- Her zaman "sırada ne var?" diye sorabilir, tek yanıt alırsınız
- **Doğrusal olmayan**: bir elemanın birden çok "sonraki"si olabilir
- Gezmek, hangi dalı izleyeceğinizi *seçmek* demektir

Ders, sıralandı: doğrusal yapılar

Yapı	Ne zaman	Belirleyici ödünleşim
Dizi (array)	Hafta 2	O(1) erişim; ortada O(n) ekleme/silme
Bağlı liste	Hafta 2	O(n) erişim; oraya varınca O(1) ekleme/silme
Yığın (stack, LIFO)	Hafta 3	Yalnızca bir uca dokunun; her işlem O(1)
Kuyruk (queue, FIFO)	Hafta 3	Bir uçtan ekle, diğerinden çıkar; her işlem O(1)

Ders, sıralandı: doğrusal olmayan ve anahtarla

Yapı	Ne zaman	Belirleyici ödünleşim
İkili ağaç, heap	Hafta 4	Dengeliyken O(log n), en çok 2 çocuk
Çizge (graph)	Hafta 5–6	Herhangi sayıda bağlantı; ağlar, haritalar
Hash tablosu (anahtarla)	Hafta 7	Ortalama O(1), konumla değil anahtarla
Dosyalar (diskte)	Hafta 13–15	Aynı ödünleşimler, disk G/Ç ile ödenir

Sık yapılan hatalar

- "Doğrusal olmayan" **"düzensiz"** anlamına gelmez
- "Sıralı" ile "doğrusal" aynı şey değil — farklı eksenler
- Bir hash tablosunun *yuvaları* doğrusaldır; anahtarla erişim yeni olan

Mini soru

Bir tabak yığını doğrusal mı, doğrusal olmayan mı? Neden?

Yanıt

Doğrusal. Her tabağın (en üsttekiler hariç) tam olarak bir üstü, (en alttaki hariç) tam olarak bir altı var — tek "sonraki".

Mini soru

Bir soy ağacı doğrusal mı, doğrusal olmayan mı?

Yanıt

Doğrusal olmayan. Bir ebeveynin birden çok çocuğu olabilir, yani belirli bir kişiden birden çok "sonraki" çıkabilir.

3. Algoritma Analizi: Saniyeyi Değil, Adımı Saymak

Başlangıç sorusu

Bir diziyi arayan bir fonksiyon yazdınız.
Çalışıyor. *Hızlı* mı? Kronometre size
bugün, laptopınız hakkında bir şey söyler.

Kısa bir tarihçe

- **1894** — Bachmann büyük-**O** gösterimini tanıtır
- **1900'lerin başı** — Landau yaygınlaştırır ("Landau sembolü")
- **1960'lar** — bilgisayar bilimi algoritma maliyeti için benimser
- **1976** — Knuth'un makalesi CS için O/Ω/Θ'yı standartlaştırır

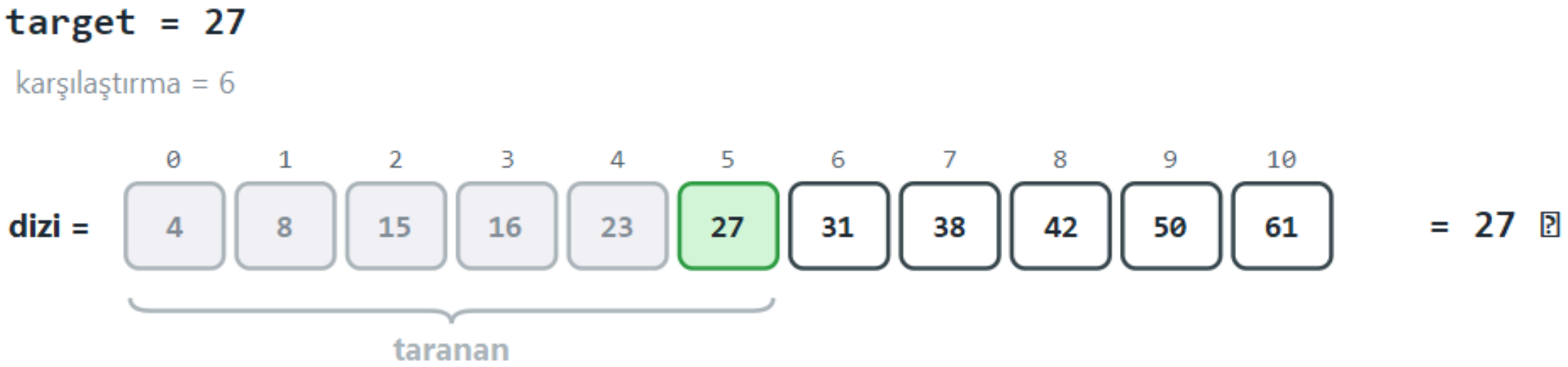
Sezgi: adımları say

Algoritmanın temel iş birimini — bir karşılaştırma, bir dizi erişimi — girdi büyüklüğü **n**'nin bir fonksiyonu olarak sayın.

Doğrusal arama: her kutuyu dene

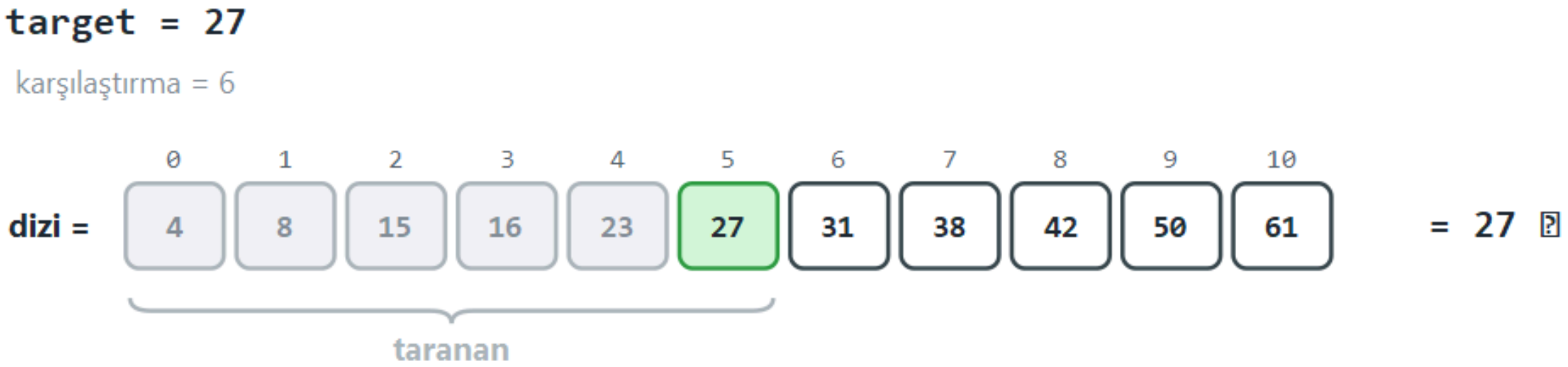
Baştan başlar, hedefi bulana ya da kutular bitene kadar elemanları tek tek dener. Veride **belirli bir sıra** gerektirmez.

Doğrusal arama, adım adım



Sonuç: arr[5] = 27, 6 karşılaştırmada bulundu. En kötü durumda (son eleman ya da hiç yoksa) doğrusal arama n = 11 karşılaştırma yapar: O(n).

Uç durum — hedef dizide yok



Sonuç: arr[5] = 27, 6 karşılaştırmada bulundu. En kötü durumda (son eleman ya da hiç yoksa) doğrusal arama n = 11 karşılaştırma yapar: O(n).

Kod — linear_search (döngü)

```
for (int i = 0; i < n; i++) {
    (*comparisons)++;
    if (arr[i] == target)
        return i;
}
```

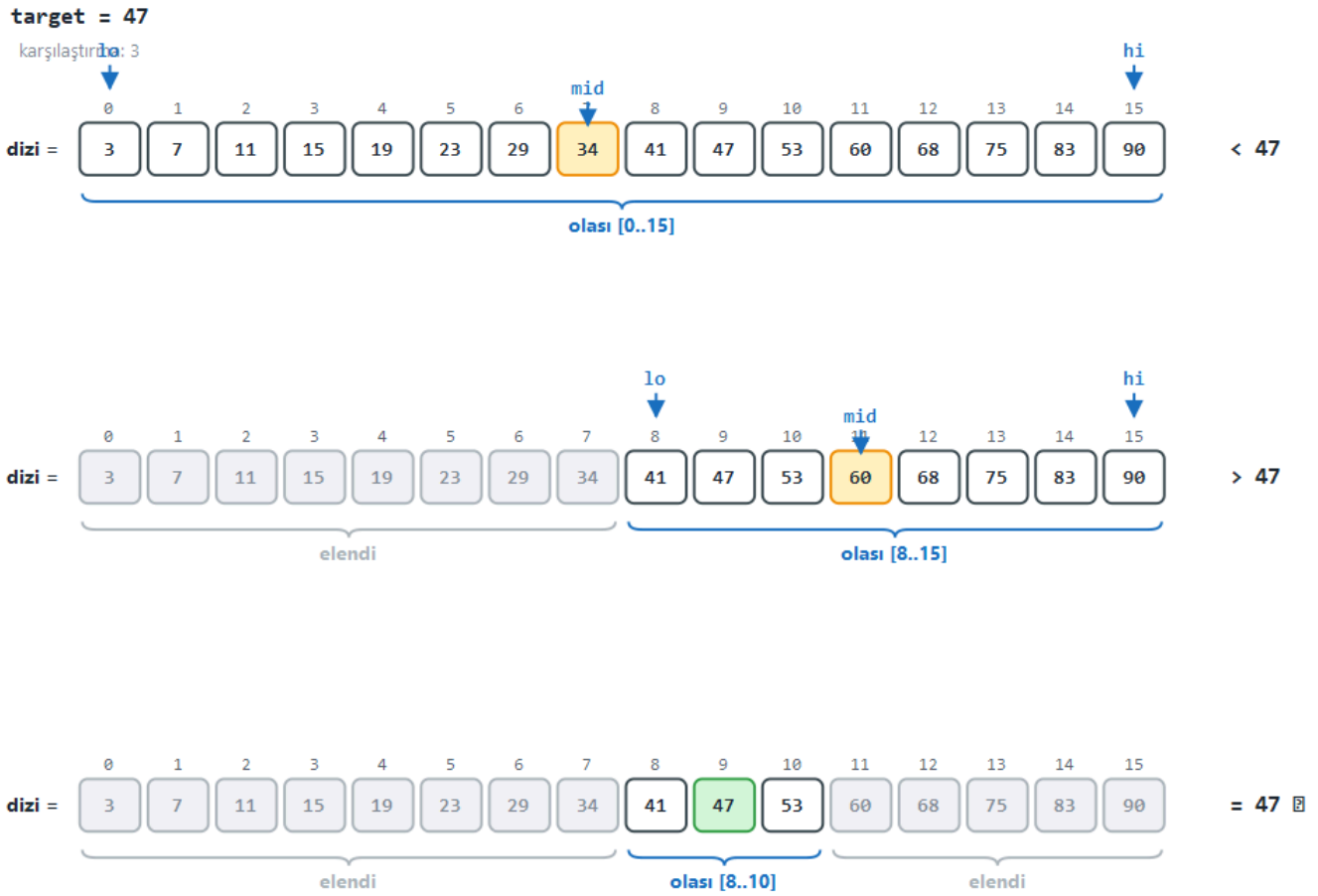
Doğrusal aramanın sonucu

- En kötü durum (son eleman ya da yok): **n** karşılaştırma
- En iyi durum (ilk eleman): **1** karşılaştırma
- Maliyet **doğrudan n ile büyür** — bu **O(n)**

İkili arama (binary search): her seferinde yarısını ele

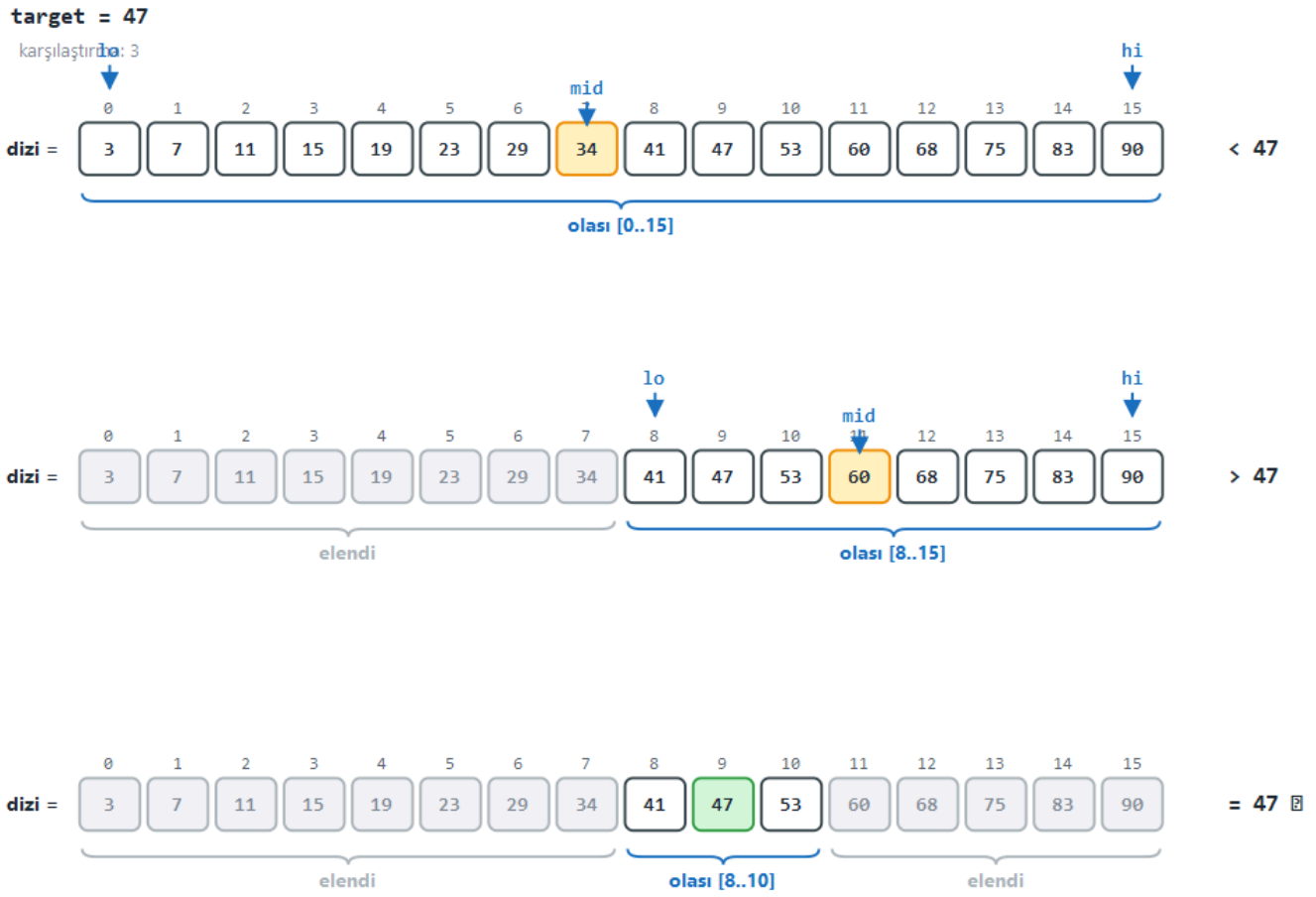
Dizi **sıralıysa**: ortaya bak. Küçükse
sol yarıyı, büyükse sağ yarıyı at.
Kalan yarıda tekrarla.

İkili arama, adım adım



Sonuç: arr[9] = 47, yalnız 3 karşılaştırmada bulundu. Her adımda arama uzayı yarıya iner: O(log n).

Uç durum — bulunamadı: lo, hi'yi geçiyor



Sonuç: arr[9] = 47, yalnız 3 karşılaştırmada bulundu. Her adımda arama uzayı yarıya iner: O(log n).

Kod — binary_search (temel adım)

```
int mid = lo + (hi - lo) / 2;
(*comparisons)++;
if (arr[mid] == target)
    return mid;
if (arr[mid] < target)
    lo = mid + 1;
else
    hi = mid - 1;
```

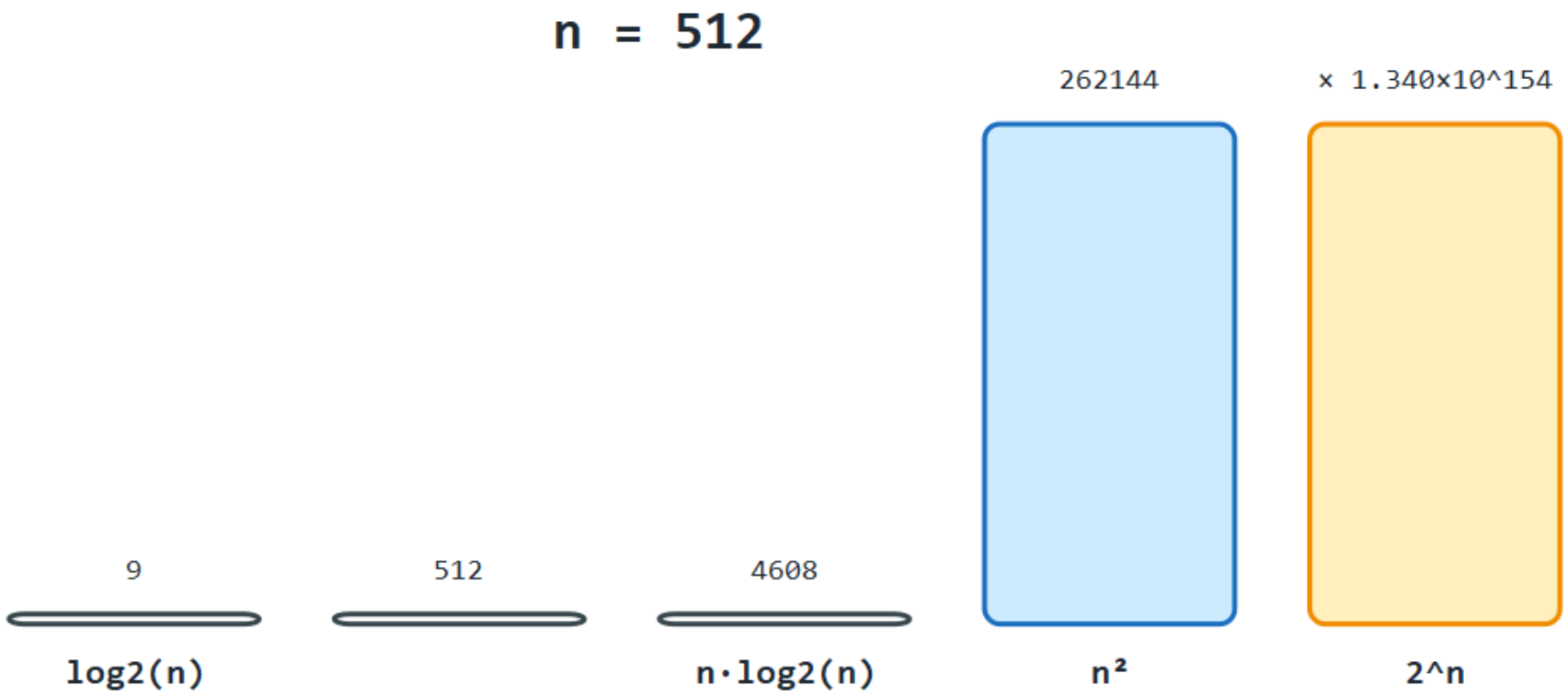
İkili aramanın sonucu

- Aynı `42`, 9 değil, yalnızca **3** karşılaştırmada bulundu
- Her karşılaştırma, kalanın **yarısını** çöpe atar
- 1'e inene kadar yarılama sayısı: **$\log_2 n$ — $O(\log n)$**

Büyüme yarışı: hız değil, şekil kazanır

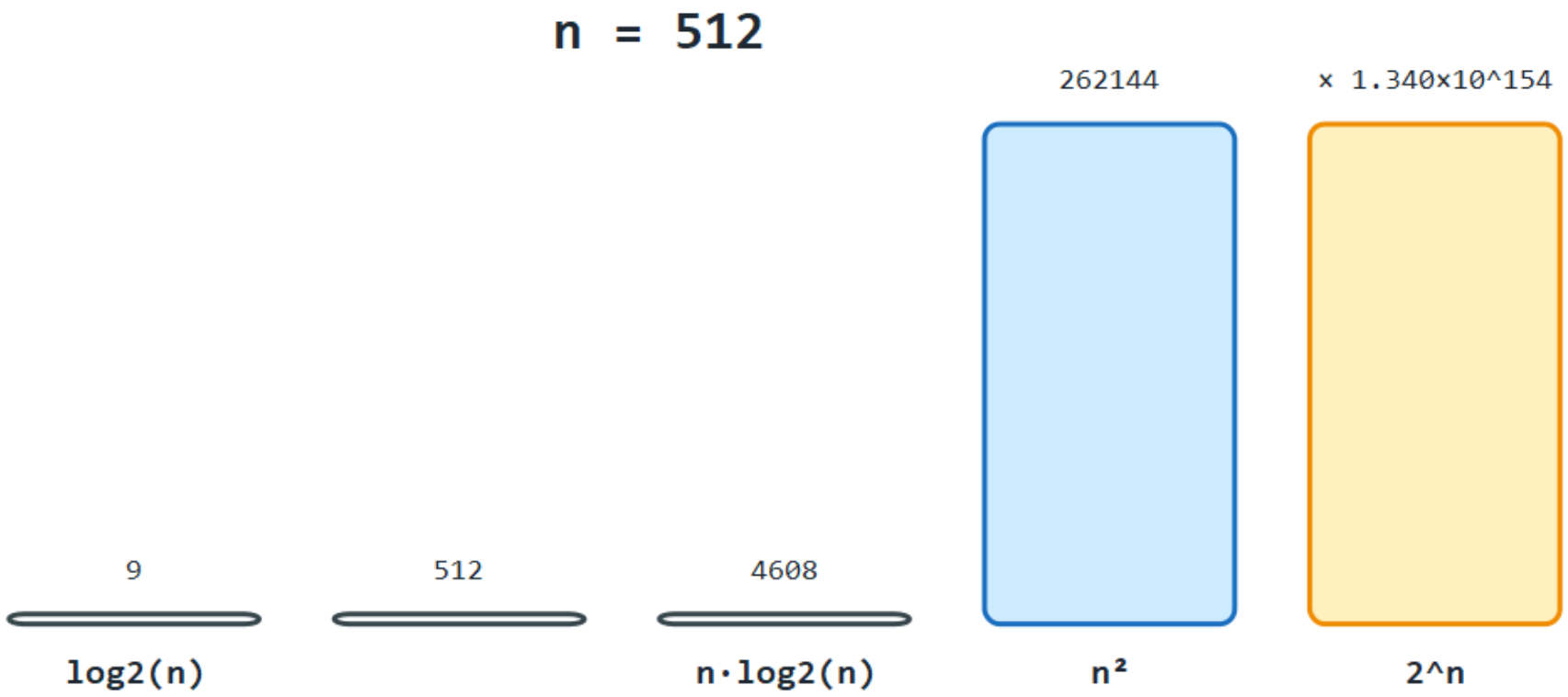
n ikiye katlanırken üç fonksiyonu yarıştıralım: düz `n`, `n·log₂n` (en iyi sıralamalar), `n²` (basit sıralamalar, iç içe döngüler).

Büyüme yarışı, adım adım



Sırayla $\log_2 n < n < n\cdot\log_2 n < n^2 < 2^n$ — her biri bir öncekini kesin olarak geride bırakır. Son n'de (512), n^2 = 262144 iken 2^n = 1.340×10^154: üstel (exponential) karmaşıklık, çok küçük girdilerde bile çok hızlı büyür.

Uç durum — 2^n neredeyse anında taşıyor



Sırayla $\log_2 n < n < n \cdot \log_2 n < n^2 < 2^n$ — her biri bir öncekini kesin olarak geride bırakır. Son n'de (512), n^2 = 262144 iken 2^n = 1.340×10^154: üstel (exponential) karmaşıklık, çok küçük girdilerde bile çok hızlı büyür.

Gerçekçi büyüklüklerde büyüme

n	$n \cdot \log_2 n$	n^2
100	664	10.000
10.000	132.877	100.000.000
100.000	1.660.964	10.000.000.000

Kod — büyüme tablosu

```
for (int i = 0; i < count; i++) {
    long n = ns[i];
    double nlogn = (double) n * log2((double) n);
    double nsq = (double) n * (double) n;
}
```

Big-O, tam olarak

`f(n)`, belirli bir noktadan sonra `f`, `g`'nin sabit bir katından hiç hızlı büyümüyorsa `O(g(n))`'dir. **Sabitleri at, en büyük terimi tut.**

En büyük terimi okumak

Sayılan adımlar	En büyük terim	Big-O
`3n + 7`	`n`	O(n)
`n² + 2n + 1`	`n²`	$O(n^2)$
`2·log₂n + 5`	`log n`	O(log n)

Sıralama, hızlıdan yavaşa

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$

İlk üçüyle bugün tanıştınız —
`O(n log n)` ve `O(2ⁿ)` bu dönem ilerde geliyor.

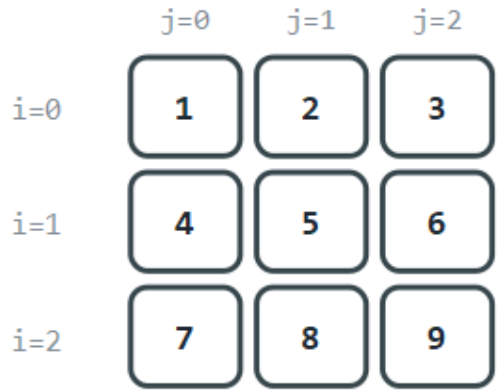
T(n)'yi elle kurmak: iç içe bir döngü

Bir döngünün içinde başka bir döngü: n dış yinelemenin her biri için, iç döngü kendi n yinelemesinin tamamını çalıştırır.

İç içe döngüyü saymak, adım adım

n = 25 (square (j < n))

çalıştırma: 9



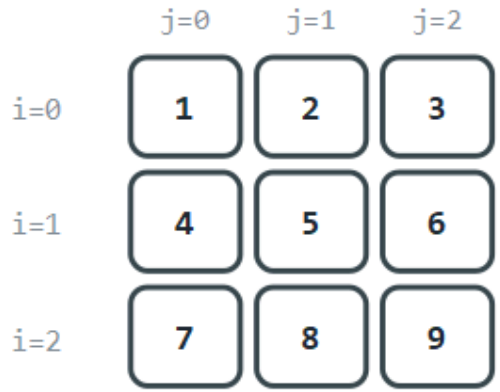
T(3) = 9
T(4) = 16
T(5) = 25
T(6) = 36
T(8) = 64
T(10) = 100
T(12) = 144
T(16) = 256
T(20) = 400
T(25) = 625

Sabitleri ve düşük dereceli terimleri atınca geriye n^2'in en büyük terimi kalır: karmaşıklık (complexity) $O(n^2)$. Bu, saymanın 3.7. bölümdeki kuralın ta kendisi.

Uç durum — farklı bir döngü şekli

n = 25 (square (j < n))

çalıştırma: 9



T(3) = 9
T(4) = 16
T(5) = 25
T(6) = 36
T(8) = 64
T(10) = 100
T(12) = 144
T(16) = 256
T(20) = 400
T(25) = 625

Sabitleri ve düşük dereceli terimleri atınca geriye n^2'in en büyük terimi kalır: karmaşıklık (complexity) $O(n^2)$. Bu, saymanın 3.7. bölümdeki kuralın ta kendisi.

Kod — iç içe döngü

```
long count = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        count++;
        (*operations)++;
    }
}
```

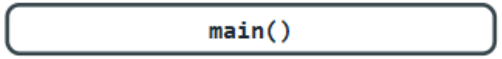
En iyi, en kötü ve ortalama durum

- **En iyi**: hedef ilk kontrol edilen — 1 karşılaştırma
- **En kötü**: hedef son ya da yok — n karşılaştırma
- **Ortalama**: `(1 + 2 + ... + n) / n = (n + 1) / 2`
- Niteliksiz "algoritmanın Big-O'su" genelde **en kötüyü** anlatır

Alan karmaşıklığı: aynı fikir, bellek için

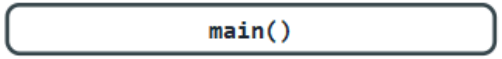
Big-O adımları sayarak **zamanı** ölçer;
alan karmaşıklığı, girdinin ötesindeki ek
depolamayı — **belleği** — ölçer.

Özyinelemeli ve döngülü toplam, adım adım



Sonuç: iki fonksiyon da aynı 550 değerini döndürür. Ama sum_recursive, n ile büyüyen O(n) yığın belleği kullandı; sum_iterative her zaman tam olarak O(1) kullandı — n ne olursa olsun tek çerçeve.

Uç durum — derin özyineleme, 22 çerçeve



Sonuç: iki fonksiyon da aynı 550 değerini döndürür. Ama sum_recursive, n ile büyüyen O(n) yığın belleği kullandı; sum_iterative her zaman tam olarak O(1) kullandı — n ne olursa olsun tek çerçeve.

Kod — özyinelemeli ve döngülü toplam

```c
int sum_recursive(const int arr[], int n) {
    if (n == 0)
        return 0;
    return arr[n - 1] + sum_recursive(arr, n - 1);
}

int sum_iterative(const int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++)
        total += arr[i];
    return total;
}
```

Sık yapılan hatalar

- Big-O'yu *koddan* değil *adımlardan* okumak
- İkili aramanın **sıralı** girdi istediğini unutmak
- O(1)'i "anlık" sanmak — yalnızca "büyümüyor" demek
- Ortalama önemliyken yalnızca en kötüyü söylemek

Mini soru

Bir algoritma tam olarak `5n + 20` temel işlem yapıyor. Big-O'su nedir?

Yanıt

O(n). Sabit çarpanı (5) ve toplamsal sabiti (20) atın; yalnızca en hızlı büyüyen terim, `n`, kalır.

Mini soru

İkili arama neden bağlı bir listede,
dizide olduğu gibi doğrudan kullanılamaz?

Yanıt

İkili arama ortaya indeksle **O(1) erişim** ister. Bağlı liste düğüm düğüm gezilmeli — zaten O(n), kazancı siliyor.

Mini soru

Niteliksiz olarak, "algoritmanın Big-O'su" genelde hangi durumu anlatır?

Yanıt

En kötü durum: garantili üst sınır —
size verilen girdi ne olursa olsun geçerli
olduğu için değerli.

4. İşaretçiler (Pointer) ve Nesneler

Başlangıç sorusu

Çağıranda iki değişkeni takas eden `swap(a, b)` yazın. Birçok dilde bu imkânsız — fonksiyon yalnızca *kopyaları* görür.

Kısa bir tarihçe

- **1966/1969** — BCPL, sonra B: adres-al, dereferans
- **1972** — Dennis Ritchie'nin C'si işaretçiyi merkeze koyar
- **1995** — Java, ham işaretçileri kaldırır, **referans**ları tutar
- Referans: paylaşılan veri, aritmetik yok, kötü adres yok

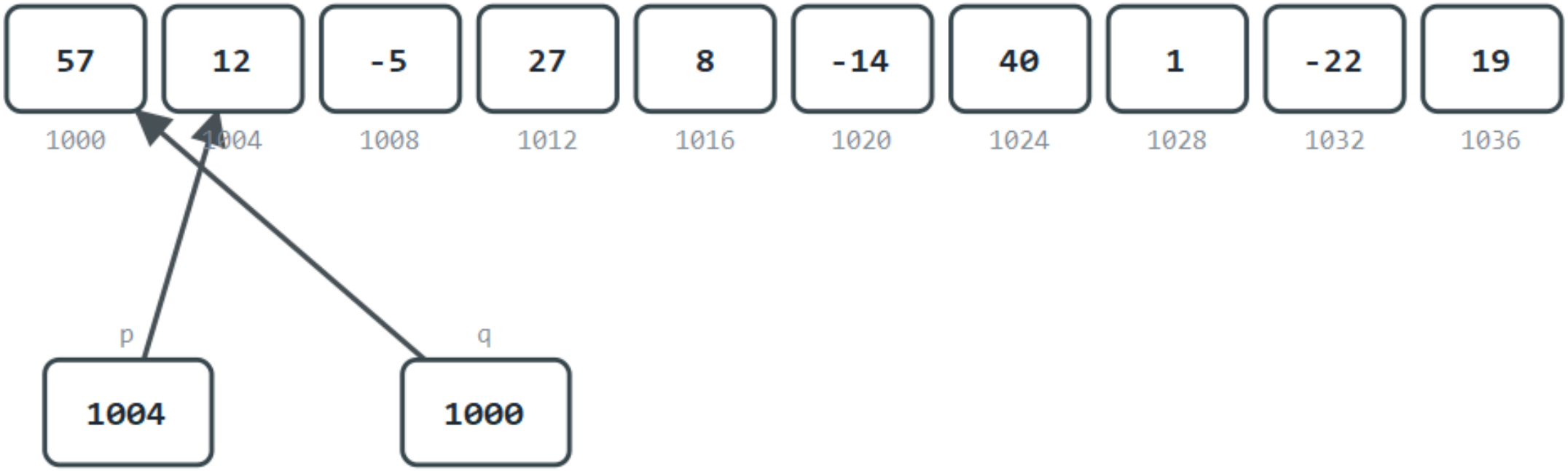
Sezgi: bellek, numaralı posta kutuları

- Bir değişken bir posta kutusu: adres + içerik
- `&x`, "x'in posta kutusu numarası ne?" diye sorar
- Bir **işaretçi**, başka bir kutunun numarasını tutar
- Dereferans (`*p`): o kutuya git, oku

İşaretçi işlemleri

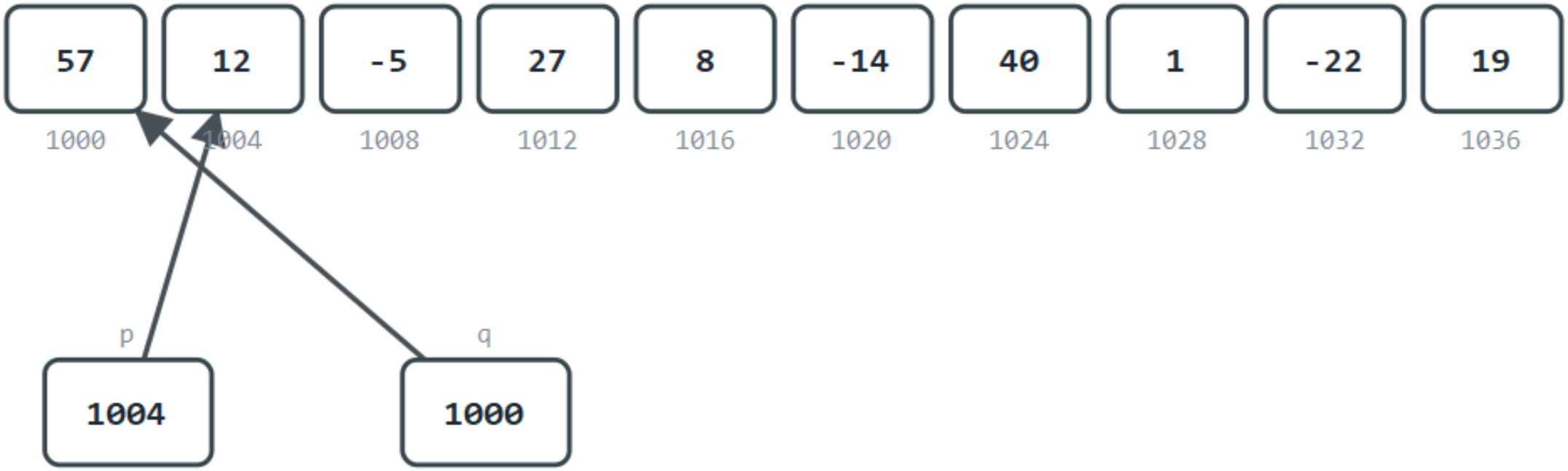
İşlem	Sözdizimi	Karmaşıklık
Adres-al	`&x`	O(1)
Bildirim	`int *p;`	O(1)
Adres ata	`p = &x;`	O(1)
Dereferans (oku/yaz)	`*p / *p = v;`	O(1)

Bir değişken, adresi, bir işaretçi



Bitti: son değerler 57, 12, -5, 27, 8, -14, 40, 1, -22, 19. İşaretçiler: p = 1004, q = 1000.

Uç durum — bir NULL işaretçi dereferansı



Bitti: son değerler 57, 12, -5, 27, 8, -14, 40, 1, -22, 19. İşaretçiler: p = 1004, q = 1000.

Kod — işaretçiler: adres, takma ad, NULL

```
int *p = NULL, *q = NULL;
p = &v[i];
*p = val;
q = p;
*q += k;
if (r != NULL)
    *r = val;
```

Java: referans ile değer, yan yana

```
int[] box = {3};
int[] alias = box;
alias[0] = 5;
int y = x;
y = 99;
```

İşaretçi aritmetiği: p + k, k bayt değil

`p + k` asla `k` bayt ilerlemez — `k × sizeof(*p)` bayt ilerler. `int` 4 bayt, yani `p + 1`, 1 değil 4 bayt atlar.

İşaretçi aritmetiği, adım adım

type = int, sizeof(int) = 4

p + 9 = 1000 + 9×4 = 1036, *(p+9) = 100

Özet: p + k, k bayt değil, her zaman k × sizeof(int) = k×4 bayt ileri gider. Aralık dışı k tanımsız davranış'tır. Java'da işaretçi aritmetiği yok — yalnızca a[k], ve aralık dışı k net bir istisna (exception) fırlatır.

Uç durum — aralık dışı offsetler

type = int, sizeof(int) = 4

p + 9 = 1000 + 9×4 = 1036, *(p+9) = 100

Özet: p + k, k bayt değil, her zaman k × sizeof(int) = k×4 bayt ileri gider. Aralık dışı k tanımsız davranış'tır. Java'da işaretçi aritmetiği yok — yalnızca a[k], ve aralık dışı k net bir istisna (exception) fırlatır.

Kod — işaretçi aritmetiği

```
int a[N];
int *p = a;
if (k < 0 || k >= N) {
    /* aralık dışı: tanımsız davranış (UB) */
} else {
    void *addr = (void *) (p + k);
    int v = *(p + k);
}
```

Struct'a işaretçi ve `->`

Bir `struct`, alanları tek bir değerde gruplar. `(*p).alan` o kadar yaygın ki C ona bir kısayol verir: `p->alan`.

Struct işaretçisi, bir diziyi gezmek


```
Bitti: 10 kayıt gezildi, 2 not güncellendi. Son kayıtlar: 101:62, 102:78, 103:95, 104:91, 105:40, 106:83, 107:67, 108:40, 109:88, 110:59.
```

Uç durum — yalnızca tek öğrenci

Bitti: 10 kayıt gezildi, 2 not güncellendi. Son kayıtlar: 101:62, 102:78, 103:95, 104:91, 105:40, 106:83, 107:67, 108:40, 109:88, 110:59.

Kod — struct dizisini gezmek

```
Student *p = students;
for (; p != students + N; p++) {
    int id = (*p).id;
    int grade = p->grade;
    p->grade = new_grade;
}
```

Sık yapılan hatalar

- `int *p;`'yi hiçbir yeri göstermeden kullanmak (vahşi işaretçi)
- `int *p, x;` — yalnızca `p` işaretçi, `x` değil
- **Bildirimde** ve **ifadede** `*`'ı karıştırmak
- Java'nın istisnası, C'nin tanımsız davranışıyla aynı değil

Mini soru

`int *p = &x;` verildiğinde, `p` ile `*p` arasındaki fark nedir?

Yanıt

`p`, işaretçide saklanan **adres**tir (x'in adresi). `*p`, o adresteki **değer**dir (x'in değeri).

Mini soru

Java'nın neden hiç `->` operatörüne ihtiyacı yok?

Yanıt

Java'da her nesne/dizi değişkeni **zaten bir referans** — `.` her zaman "önce izle, sonra alana eriş" demek.

5. Bellek Düzeni: Yığın, Öbek ve Kim Temizliyor

Başlangıç sorusu

Bir fonksiyon yerel bir dizi bildirir ve
döner. Ona ne olur? Şimdi karşılaştırın:
bir fonksiyon `malloc` yapıp serbest bırakmadan döner.

Kısa bir tarihçe

- **Yığın (stack)** — çağrı başına bir çerçeve, Algol 60'a dayanır
- `malloc` / `free` — en eski Unix C kütüphaneleri, 1970'lerin başı
- **Çöp toplama (garbage collection)** — McCarthy, Lisp, **1959**
- Java (1995), çöp toplamayı günlük programlamada yaygınlaştırdı

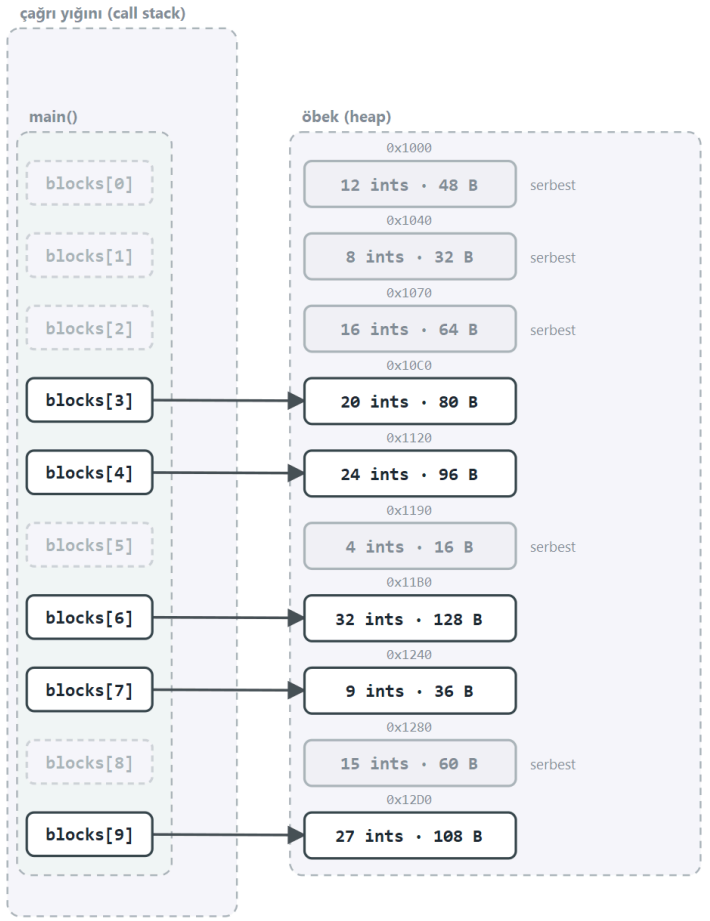
Sezgi: düzenli bir yığın ile bir depo

- **Yığın**: her çağrı bir çerçeve iter; dönüş onu çeker
- Her zaman LIFO, her zaman otomatik, her zaman hızlı
- **Öbek**: bir blok istersiniz; *geri verilene kadar* durur
- C'de bu "geri vermeyi" sizin dışınızda kimse yapmaz

Yığın ile öbek, yan yana

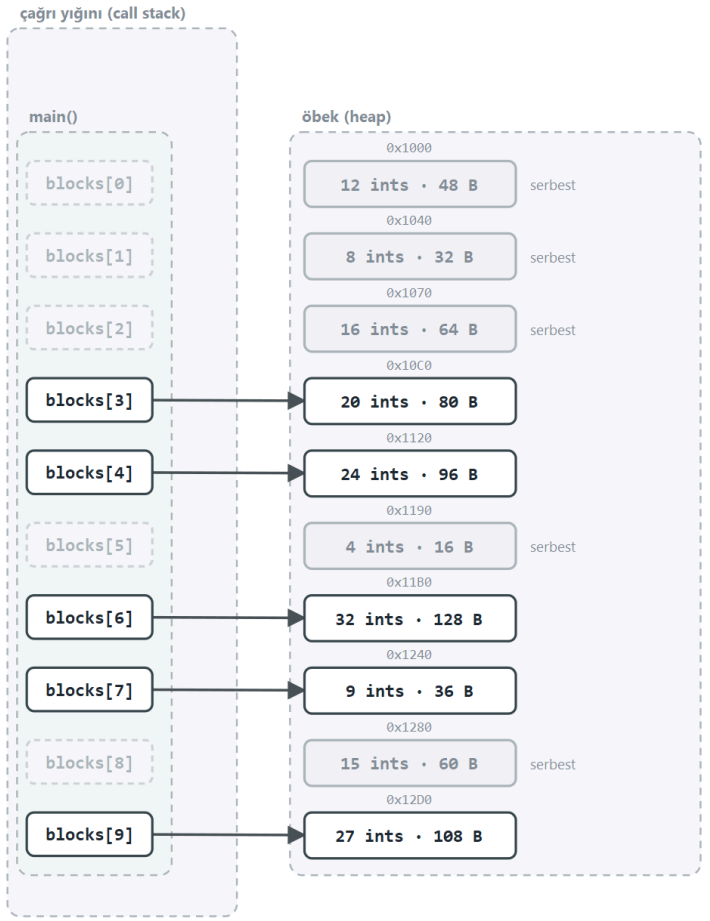
	Yığın (Stack)	Öbek (Heap)
Kim ayırır	Derleyici, otomatik olarak	Siz: `malloc` (C) / `new` (Java)
Kim serbest bırakır	Otomatik, dönüşte	C: `free()` . Java: çöp toplayıcı
Hız	Son derece hızlı	Daha yavaş: boş blok bulur

Bir yığın çerçevesi bir öbek bloğuna nasıl bağlanır



Bitti: en fazla 2 çerçeve aynı anda (main + bir çağrı), 10 blok ayrıldı. Sızıntı yok: her blok bir blocks[] sahibine sahipti.

Uç durum — sarkan bir işaretçi, işaretlendi



Bitti: en fazla 2 çerçeve aynı anda (main + bir çağrı), 10 blok ayrıldı. Sızıntı yok: her blok bir blocks[] sahibine sahipti.

Kod — alloc_block / free_block

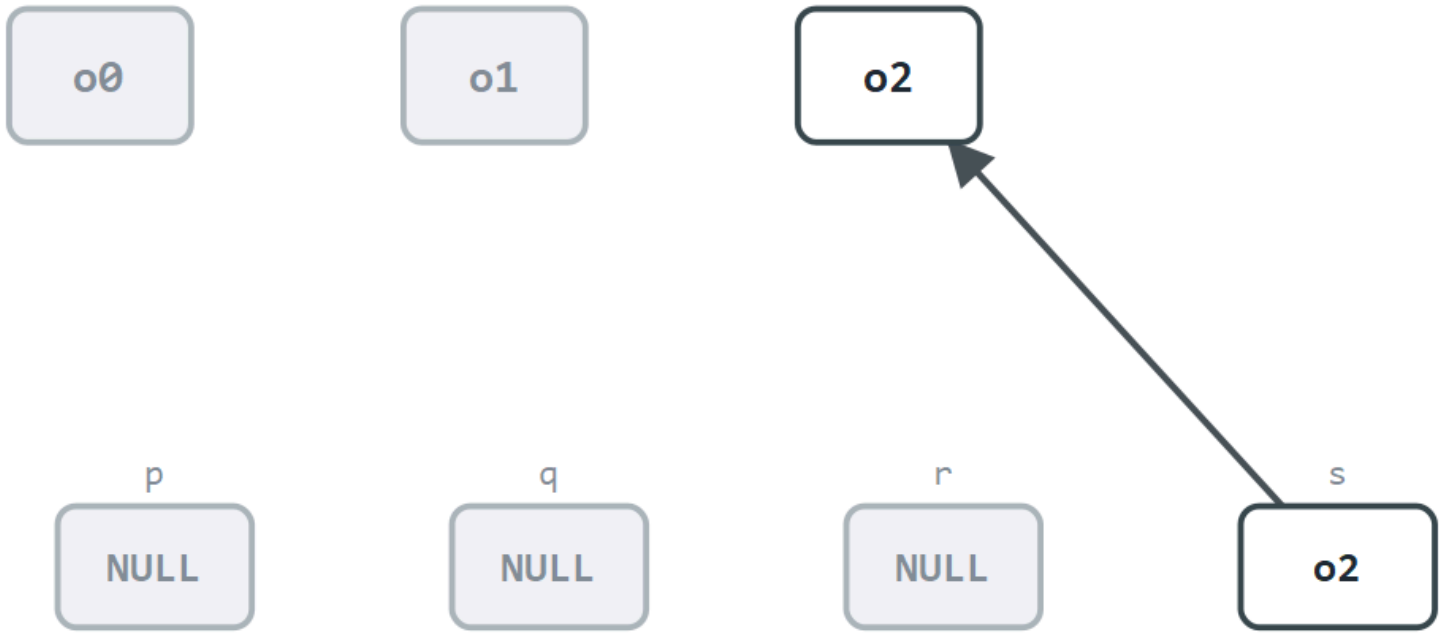
```c
static int *alloc_block(int size) {
    int *p = malloc(size * sizeof(int));
    return p;
}

static void free_block(int *p) {
    free(p);
}
```

Java: `new` ve çöp toplama

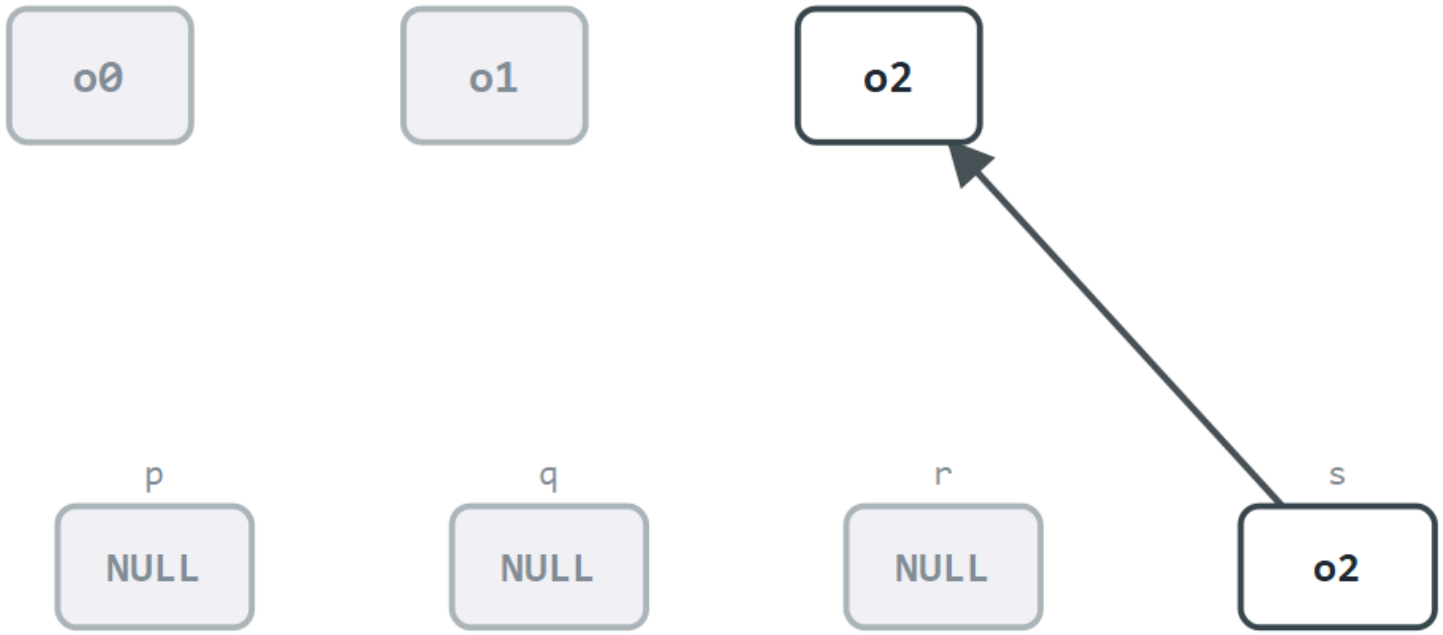
C'de `free`'yi unutmak — ya da serbest bıraktıktan sonra işaretçiyi kullanmak — tamamen sizin sorumluluğunuzda. Java `free`'yi diliden kaldırır.

Java referansları, takma adlar, çöp toplama



Bitti: 3 nesne yaratıldı. Toplanan: o0, o1. Hâlâ erişilebilir: o2.

Uç durum — bir döngü, yine de toplanır



Bitti: 3 nesne yaratıldı. Toplanan: o0, o1. Hâlâ erişilebilir: o2.

Kod — Java referansları: takma ad ve döngü

```
Node p = new Node();
Node q = p;
p.ref = q;
q.ref = p;
p = null;
q = null;
```

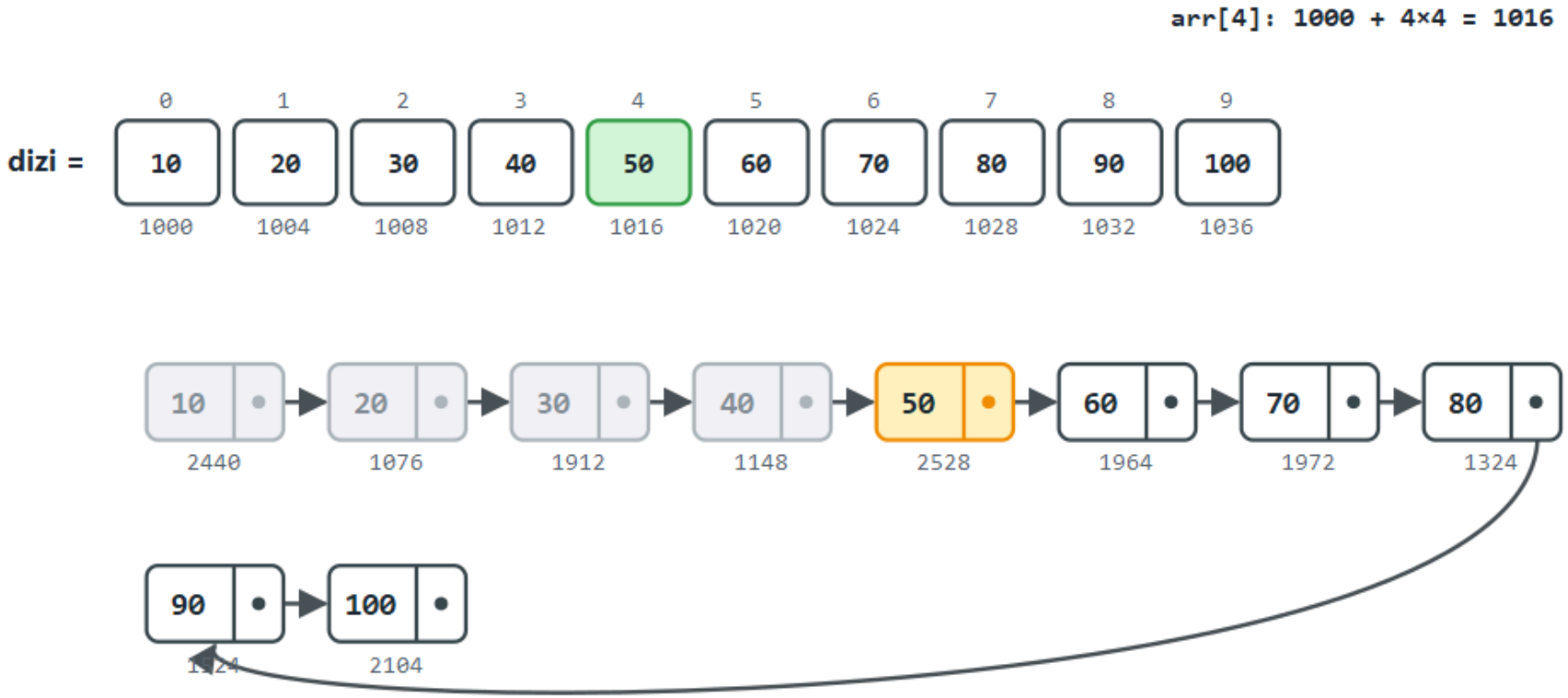
Sık yapılan hatalar

- Sarkan işaretçi: `free`'den sonra işaretçiyi kullanmak, NULL yapmadan
- Çift serbest bırakma: aynı işaretçiyi iki kez `free` etmek
- Erken dönüşte `free`'yi unutmak (bir **sızıntı**)
- Java'nın çöp toplayıcısının sızıntıyı imkânsız kıldığını sanmak (kılmaz)

Önizleme: dizi düzeni ile bağlı liste düzeni

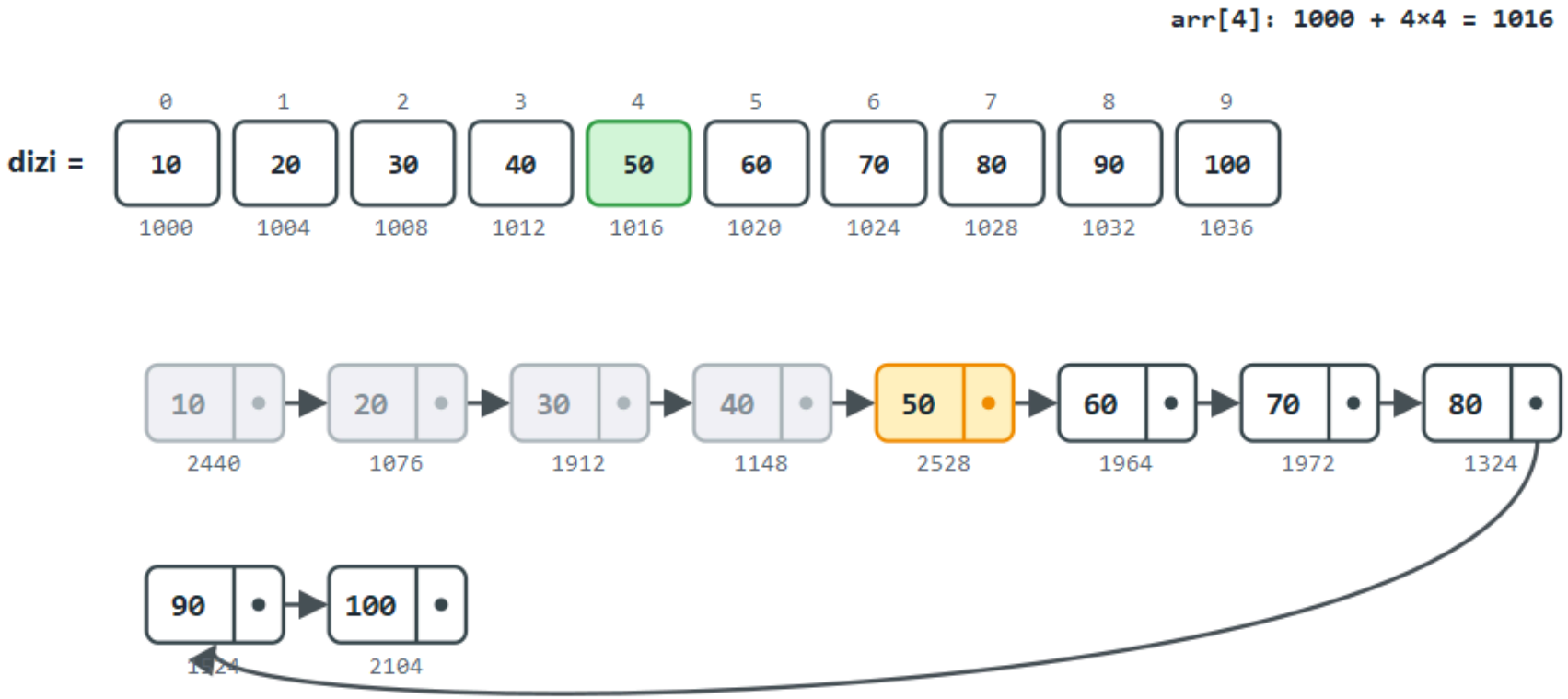
Bir dizi tek bir bitişik blok ayırır: `arr[i]` tek bir hesap, O(1). Bağlı liste düğümleri dağıtır, işaretçilerle bağlar.

Dizi ile bağlı liste düzeni, adım adım



Sonuç: aynı k = 4 için dizi 1 adımda ulaştı (adres 1016); bağlı liste baştan başlayıp next'i 4 kez izleyerek ulaştı: 4 sıçrama, O(n). Aynı veriler, TAM TERS erişim maliyeti.

Uç durum — son eleman: en çok sıçrama



Sonuç: aynı k = 4 için dizi 1 adımda ulaştı (adres 1016); bağlı liste baştan başlayıp next'i 4 kez izleyerek ulaştı: 4 sıçrama, O(n). Aynı veriler, TAM TERS erişim maliyeti.

Kod — bağlı liste kurmak

```
Node *head = NULL;
for (int i = N - 1; i >= 0; i--) {
    Node *node = malloc(sizeof(Node));
    node->data = arr[i];
    node->next = head;
    head = node;
}
```

Mini soru

Bir fonksiyon yerel olarak `int arr[100];`
bildiriyor ve ona bir işaretçi döndürüyor. Sorun ne?

Yanıt

`arr`, o çerçevede **yığında** yaşar.
Fonksiyon döner dönmez çerçeve çekilir —
işaretçi çoktan sarkan hale gelmiştir.

Mini soru

`free(block); block = NULL;` — bu sırayla

— neden önemli?

Yanıt

`free`, gerçek adrese ihtiyaç duyar. Önce NULL yapmak `free(NULL)`'ı etkisiz bırakır — blok sızar, adresi sonsuza dek kaybolur.

Mini soru

Sarkan bir işaretçi Java'da neden C'deki gibi asla oluşamaz?

Yanıt

Java'da `free` yok: bir nesne yalnızca
çöp toplayıcı hiçbir şeyin ulaşamadığını
kanıtlayınca geri alınır — erişilebilirken asla.

6. ASN.1, BER TLV ve PER TLV

Başlangıç sorusu

İki program, farklı diller, farklı
makineler, bir kaydı bayt olarak değişir.
Ham bir `struct` düzeni hiç taşınabilir değil.

Kısa bir tarihçe

- **1984** — ASN.1, ITU-T/CCITT X.409 olarak doğar
- **1995** — X.680 serisi: modern, güncel biçim
- **BER** (X.690) — kendini tanımlayan etiket + uzunluk + değer
- **PER** (X.691, 1994) — iki taraf şemayı bilince bit bit paketler

Sezgi: bir zarf, üç parça

- **Etiket (Tag)**: bu ne tür bir değer?
- **Uzunluk (Length)**: kaç bayt tutuyor?
- **Değer (Value)**: içeriğin kendisi
- Evrensel olarak **TLV** denir

TLV bayt düzeni

Parça	Boyut	Neyi kodlar
Etiket	1 bayt	sınıf · ilkel/kurulu · etiket numarası
Uzunluk	1+ bayt	kısa biçim (0–127) ya da uzun biçim (≥128)
Değer	Uzunluk bayt	içerik ya da iç içe TLV'ler

TLV kodlama, alan alan

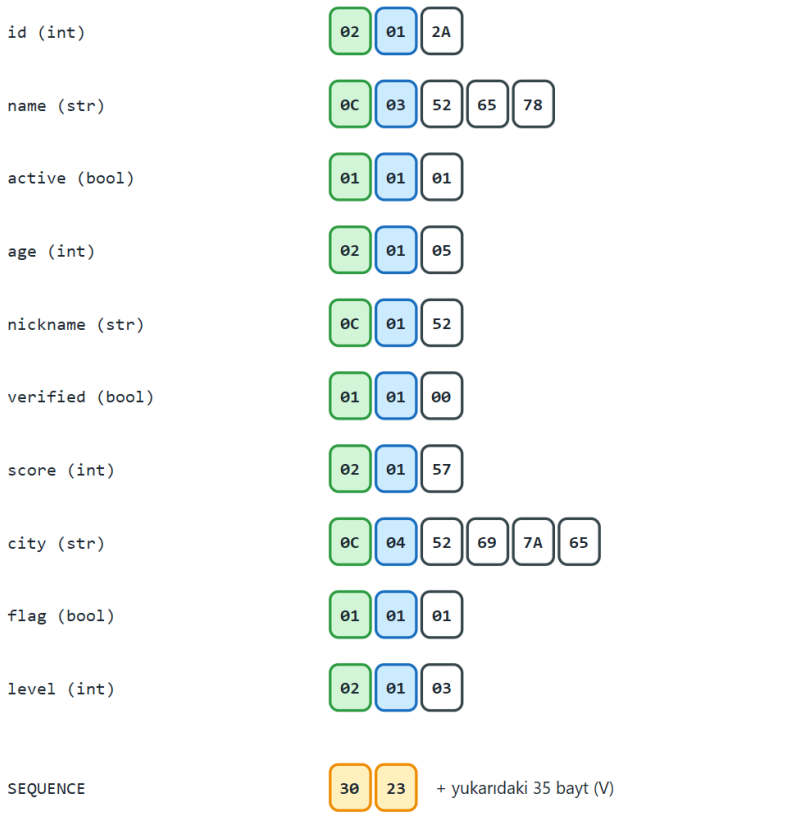
10 alanlık bir kaydı bayt dizisi olarak göndermek istiyoruz.

id (int)	02 01 2A
name (str)	0C 03 52 65 78
active (bool)	01 01 01
age (int)	02 01 05
nickname (str)	0C 01 52
verified (bool)	01 01 00
score (int)	02 01 57
city (str)	0C 04 52 69 7A 65
flag (bool)	01 01 01
level (int)	02 01 03
SEQUENCE	30 23 + yukarıdaki 35 bayt (V)

Bitti: toplam 37 bayt, tele gönderilmeye hazır. 0 boş metin alanı, 0 uzun biçim uzunluk (SEQUENCE'in kendisi).

Uç durum — uzun biçim gerektiren bir uzunluk

10 alanlık bir kaydı bayt dizisi olarak göndermek istiyoruz.



Bitti: toplam 37 bayt, tele gönderilmeye hazır. 0 boş metin alanı, 0 uzun biçim uzunluk (SEQUENCE'in kendisi).

Kod — encode_length: kısa ve uzun biçim

```
if (len < 128) {
    out[0] = (unsigned char) len;
    return 1;
}
int n = 0;
while (len > 0) {
    tmp[n++] = (unsigned char) (len & 0xFF);
    len >>= 8;
}
out[0] = (unsigned char) (0x80 | n);
```

BER ile PER

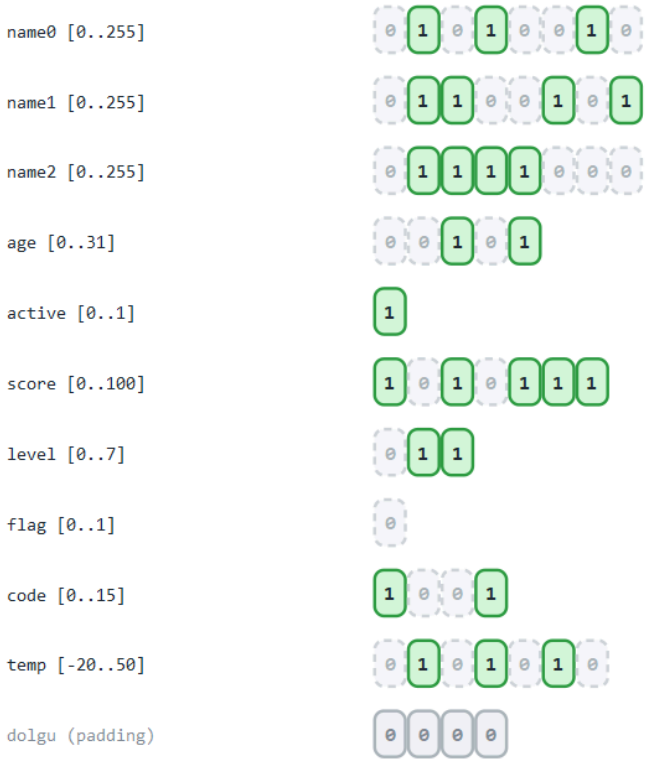
	BER	PER
Kendini tanımlar	Evet: her alanda etiket + uzunluk	Hayır: şema bilinmeli
Boyut	Daha büyük: alan başına 2+ bayt	Çok daha küçük: bayt değil bit
Tipik kullanım	X.509, LDAP, SNMP	Bant genişliği kritik (hücresel)

PER: tam olarak gerektiği kadar bit

İki taraf bir alanın `[min, max]`
aralığını biliyorsa, ne etiket ne uzunluk
gerekir — yalnızca `ceil(log2(max - min + 1))` bit.

PER kodlama, bit bit

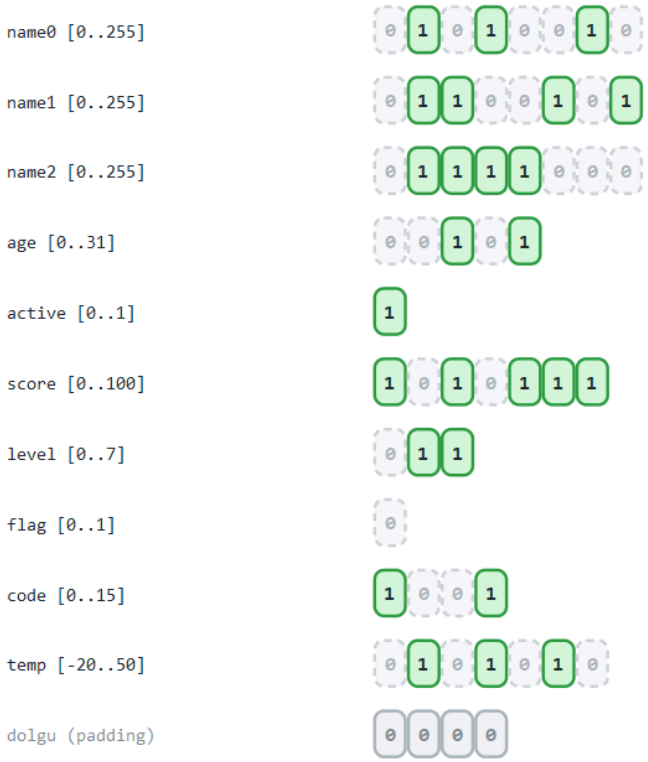
10 alanlık AYNI türden bir kayıt, bu kez bit bit paketleniyor: etiket yok, uzunluk yok.



Bitti: 52 anlamlı bit, 7 bayta paketlendi. Aynı alanlara TLV'de Tag+Length verilse kabaca 30 bayt (240 bit) gerekirdi — PER, kendini tanımlamanın çoğunu feda ederek küçülür.

Uç durum — büyüklüğü 1 olan bir aralık: sıfır bit

10 alanlık AYNI türden bir kayıt, bu kez bit bit paketleniyor: etiket yok, uzunluk yok.



Bitti: 52 anlamlı bit, 7 bayta paketlendi. Aynı alanlara TLV'de Tag+Length verilse kabaca 30 bayt (240 bit) gerekirdi — PER, kendini tanımlamanın çoğunu feda ederek küçülür.

Kod — pack_bits: bir seferde bir bit

```
int bit = (int) ((value >> i) & 1u);
int byte_index = *bitpos / 8;
int bit_index = 7 - (*bitpos % 8);
/* if (bit) buf[byte_index] |= bit (en anlamlıdan) */
(*bitpos)++;
```

Sık yapılan hatalar

- Uzunluk önekini unutup sabit bir okumanın işe yarayacağını ummak
- Uzunluğun her zaman tek bayta sığacağını sanmak (yalnız 128'in altı)
- Uzunluğu (yalnız V) tüm TLV'nin boyutuyla karıştırmak
- BER ve PER baytlarını birbirinin yerine geçer sanmak

Mini soru

TLV'deki üç harf neyin kısaltması, ve hangi sırayla gelirler?

Yanıt

Tag, Length, Value — tam olarak bu sırayla: ne tür, kaç bayt, sonra baytların kendisi.

Mini soru

SEQUENCE'in etiket baytı neden `0x10` değil (16 sayısının ikilik hâli), `0x30`?

Yanıt

En üst 3 bit etiket numarası değil: 2 sınıf biti (`00`) + 1 kurulu bit (`1`, SEQUENCE başka TLV'ler tuttuğu için) = `0x30`.

7. Uygulamalı Atölye: C Araç Zinciri

Bu neden önemli

Bundan sonra her hafta size kendi
kuracağınız C ve Java programları veriyor.
Vize projesi, kısmen temiz bir derlemeyle değerlendiriliyor.

Araç zinciri, kısaca

- **GCC** — Richard Stallman, GNU projesi, **1987**
- **GDB** — aynı GNU projesi, **1986**: adım adım gez, incele
- **CMake** — elinizdeki her araç için derleme dosyaları üretir
- Aynı `CMakeLists.txt`, Windows, WSL ve Linux'ta derlenir

Kod — ilk derlemeniz

```c
#include <stdio.h>

int main(void) {
    printf("Hello, Data Structures!\n");
    return 0;
}
```

```
gcc -std=c11 -Wall -Wextra -o /tmp/x hello_workshop.c && /tmp/x
```

Uyarıları açık derleyip okumak

`-Wall -Wextra`, kullanılmayan
değişkenleri, şüpheli karşılaştırmaları,
biçim uyuşmazlıklarını yakalar. Her program **temiz** derlenmeli.

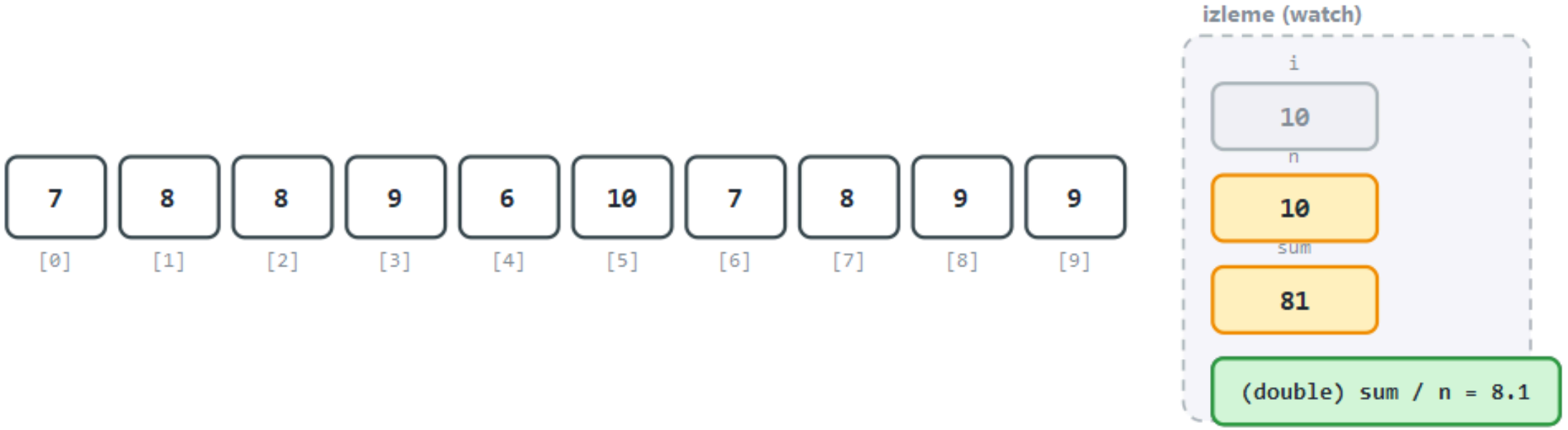
Küçük bir hata avı

```
int average_buggy(const int arr[], int n) {
    if (n == 0) return 0;
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + arr[i];
    return sum / n;
}
```

gdb ile bulmak

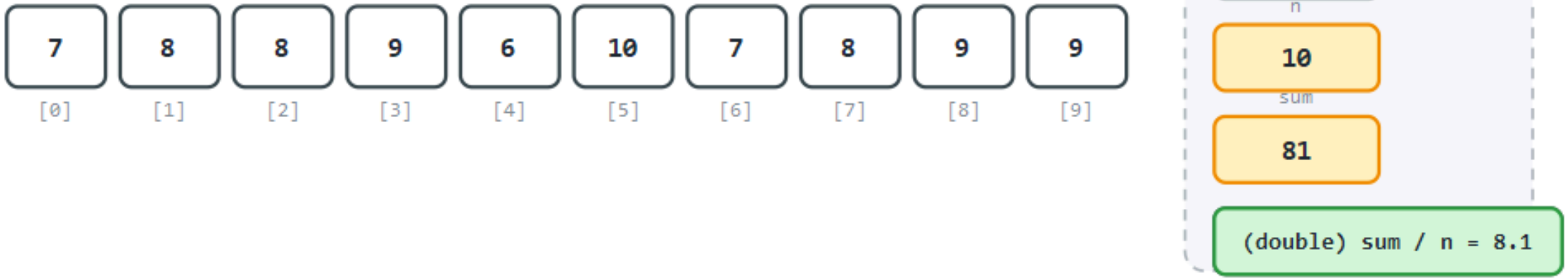
Yöntem: bir kesme noktası koy, çalıştır,
bir değişkeni incele, bir hipotezi doğrula
ya da çürüt. Canlı, adım adım izleyin.

Hata ayıklayıcıda adım adım, canlı



continue — program biter. average_buggy -> 8, average_fixed -> 8.1. Yöntem: kesme noktası koy, çalıştır, şüpheli ifadeleri yazdır, karşılaştır — kod hiç değişmedi.

Uç durum — boş dizi koruması



continue — program biter. average_buggy -> 8, average_fixed -> 8.1. Yöntem: kesme noktası koy, çalıştır, şüpheli ifadeleri yazdır, karşılaştır — kod hiç değişmedi.

gdb oturumu

```
(gdb) break debug_average.c:11
(gdb) run
(gdb) print sum
(gdb) print sum / n
(gdb) print (double) sum / n
(gdb) continue
```

Kod — CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(week1_cmake_demo C)
add_executable(week1_cmake_demo main.c)
```

```
cmake -S . -B build && cmake --build build
```

Visual Studio üzerine bir not

Build → Build Solution derler; Debug →
Start Debugging (F5) hata ayıklayıcı altında
çalıştırır; kenar boşluğuna tıklamak kesme noktası koyar.

Sık yapılan hatalar

- `-std=c11`'i unutmak: makineye göre farklı davranış
- Git'e bir `build/` klasörü ya da `.exe` dosyaları eklemek
- Başarısız bir derlemeden sonra eski bir çalıştırılabiliri koşturmak
- Yalnızca `printf`'e sarılmak, iki dakikalık bir gdb oturumuna hiç değil

Mini soru

`-Wall -Wextra` ne yapar, ve bu ders neden onun altında temiz bir derleme istiyor?

Yanıt

Geniş bir uyarı kümesini açar. Genelde gerçek hataları gösterir — temiz bir derleme istemek onları hemen düzeltir.

Mini soru

`cmake -S . -B build` ile `cmake --build build` her biri ne yapar, arada ne fark var?

Yanıt

İlki **yapılandırır**: derleme dosyaları üretir, hiçbir şey derlemez. İkincisi **derler**: o aracı çağırıp derler ve bağlar.

Özet — dönemin temelleri

Fikir	Ana gerçek
Veri yapısı	Bilinen, çözümlenebilir maliyet — her zaman bir ödünleşim
Doğrusal / doğrusal olmayan	Bir "sonraki" ya da belki birden çok
Big-O	Büyümenin üst sınırı; en büyük terimi oku
Alan karmaşıklığı	Aynı fikir, ek bellek için uygulandı

Özet — bellek ve kodlama

Fikir	Ana gerçek
İşaretçi / referans	Bir adres tutar; `&` , `*` , `->`
Yığın / öbek	Otomatik, LIFO / istenip açıkça bırakılan
TLV / BER / PER	Kendini tanımlayan baytlar, ya da şemayla paketlenen bitler
C araç zinciri	`gcc` , `gdb` , CMake — derle, çalıştır, hata ayıkla

Büyük resim

Bugün kurulan iki araç — maliyeti ölçmek
için **Big-O**, ve bir **bellek** resmi —
dönemin geri kalanındaki her yapıyı değerlendiriyor.

Öz-değerlendirme turu

Beş kısa soru. Yanıt gelmeden önce düşünün. Tam alıştırmalar ve on soruluk bir quiz hafta notlarında.

1. İki farklı, doğru algoritma neden çok farklı Big-O'ya sahip olabilir?

Big-O *adım sayısını* ölçer, yanıtın doğru olup olmadığını değil

Doğrusal arama ve ikili arama, bir değeri
sırasıyla O(n) ve O(log n)'de, ikisi de doğru bulur.

2. Şunları hızlıdan yavaşa sıralayın: O(n log n), O(1), O(n), O(log n), O(n^2)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$

3. İkili arama neden sıralı bir dizi gerektirir?

Ortayla karşılaştırmak, yalnızca bir taraf küçük diğeri büyük garantiliyse işe yarar

Bu garanti tam olarak "sıralı" demek.
Doğrusal arama hiçbir sıraya güvenmez.

4. Bir BER TLV kodlamasında Uzunluk alanı gerçekte neyi sayar?

Hemen ardından gelen Değer'deki baytları — tüm TLV'nin boyutunu değil

SEQUENCE örneğinde Uzunluk = 8'di, ama kodlanmış SEQUENCE'in tamamı 10 bayttı.

5. `p->x` neden var, ve neyin kısaltması?

`p->x`, `(*p).x`'in kısaltması: önce dereferans, sonra alana eriş

Bu tam kombinasyon C kodunda o kadar yaygın ki var olma nedeni bu.

Önümüzdeki hafta

Hafta 2 — Bağlı Listeler

Bugün tanıştığınız işaretçilerle bağlanan, bugünün öbeğinde yaşayan, tek tek ayrılmış düğümlerden bir zincir kurun.

Kaynaklar (1/2)

- Ders izlencesi, Hafta 1: `CEN207-2026-2027-Guz-Izlence.tr.md`
- Knuth. *The Art of Computer Programming, Cilt 1*, 3. baskı
- Liskov, Zilles. "Programming with Abstract Data Types," 1974
- Knuth. "Big Omicron and Big Omega and Big Theta," 1976
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3. baskı

Kaynaklar (2/2)

- Sedgewick, Wayne. *Algorithms*, 4. baskı
- Kernighan, Ritchie. *The C Programming Language*, 2. baskı
- Liang. *Introduction to Java Programming*, 10. baskı
- ITU-T X.680 / X.690 / X.691 — ASN.1, BER, PER
- GNU (GCC, GDB) ve Kitware (CMake) belgeleri