Bölüm 1: Giriş ve Algoritma Analizi

C ile Veri Yapıları ve Algoritmalar

Sistem yaşam döngüsü | Algoritma ve ADT | Seçimli sıralama | Bellek ve zaman | O, Ω, Θ | Sihirli kare | Performans ölçümü

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Süreci tanımak

Sistem yaşam döngüsünün beş aşamasını ve her aşamada veri yapısı kararlarının yerini açıklamak.

Soyutlamak

Algoritma ve soyut veri tipi (ADT) kavramlarını tanımlamak; C'de başlık dosyası ile arayüz kurmak.

Saymak

Bir C yordamının bellek gereksinimini ve adım sayısını çıkarmak (`sum` , `rsum` , `add`).

Sınıflandırmak

Adım sayısını O, Ω ve Θ ile ifade etmek; en iyi, ortalama ve en kötü durumu ayırmak.

Karşılaştırmak

Alışılmış karmaşıklık sınıflarının n büyüdükçe nasıl ayrıştığını yorumlamak.

Ölçmek

`clock()` ile işletim süresini ölçmek ve ölçüm tuzaklarını (saat vuruşu, çıktı) tanımak.

Ders Kitapları ve Kaynaklar

Başucu kitabı

Kernighan ve Ritchie, *The C Programming Language*. C'nin tasarımcısının kitabı; her an el altında bulunması gereken bir başvuru kaynağı.

Ders kitabı

Horowitz, Sahni ve Anderson-Freed, *Fundamentals of Data Structures in C*.



Aynı kitabın üç sürümü: içerik hemen hemen aynı, yalnızca kodların dili değişiyor.

Ek kaynaklar: Cormen ve diğ. (CLRS) *Introduction to Algorithms*; Sedgewick, *Algorithms in C*.

DERSE GİRİŞ · OKUMA PARÇASI

Okuma Parçası: Meslek Sevgisi

Arkeolog

Bugün yaşıyor ama aklı 5500 yıl öncesinde: “Firavunun karısı oraya bu sebeple gömülmüş olamaz” diyebiliyor.

Astronom

Kara deliği ispatlamak için yıllarca fotoğraf çekiyor; bir yıldızın bir piksel oynamasına seviniyor.

Taraftar

8 saat yol, 150 TL bilet, taş ve değnek riski… Ama konferans için aynı yolu gideni “aptal” sayıyoruz.

HOCADAN *“Meslek çok sevilmeli. Mesleğini seveni herkes dinler; elinin ucuyla anlatanı kimse dinlemez. Kodda da, kodun içinde yaşıyormuş gibi ne olup bittiği anlaşılmalı.”*

“Bir ömür mutlu olmanın yolu, işini sevmektir.” — Çin atasözü

Sistem Yaşam Döngüsü (System Life Cycle)

Yazılımlar, aşamaları aşağıdaki gibi olan bir süreçten geçer:



Gereksinimler: Proje, amacını belirten şartlarla başlar: programcıya hangi **girdiler** verilecek, hangi **çıktılar** üretilecek?

Horowitz–Sahni, Bölüm 1

TEMELLER

Sistem Yaşam Döngüsü (System Life Cycle)

Yazılımlar, aşamaları aşağıdaki gibi olan bir süreçten geçer:



Çözümleme: Problem parçalara ayrılır. İki yol var: **aşağıdan yukarı** (bottom-up) ya da **yukarıdan aşağı** (top-down).

Horowitz–Sahni, Bölüm 1

Sistem Yaşam Döngüsü (System Life Cycle)

Yazılımlar, aşamaları aşağıdaki gibi olan bir süreçten geçer:

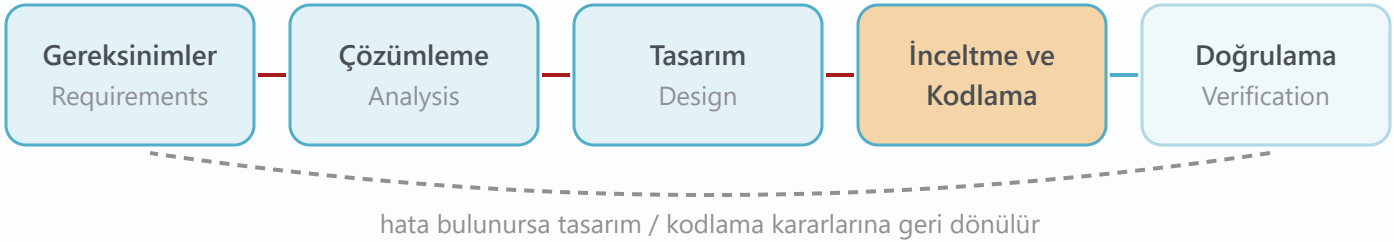


Tasarım: Veri nesneleri (→ ADT) ve işlemler (→ algoritma) tanımlanır; ikisi de **dilden bağımsızdır**. Tasarıma ayrılan zaman arttıkça hata azalır.

Horowitz–Sahni, Bölüm 1

Sistem Yaşam Döngüsü (System Life Cycle)

Yazılımlar, aşamaları aşağıdaki gibi olan bir süreçten geçer:



İnceltme ve kodlama: Seçilen veri yapıları üzerinde algoritmalar kodlanır. Veri gösterimi, algoritmanın etkinliğini belirler.

Horowitz–Sahni, Bölüm 1

TEMELLER

Sistem Yaşam Döngüsü (System Life Cycle)

Yazılımlar, aşamaları aşağıdaki gibi olan bir süreçten geçer:



Doğrulama: Doğruluk ispatı, geniş test verisiyle sınama ve hata giderme. Her biri başlı başına bir araştırma konusu.

Horowitz–Sahni, Bölüm 1

TEMELLER

Çözümleme: Aşağıdan Yukarı mı, Yukarıdan Aşağı mı?

Aşağıdan yukarı (bottom-up)

+ Problem parçalara bölünür, her parçaya ayrı odaklanılır.
+ Modüller doğrudan görülür; problem modül modül çözülür.
+ **Proje konusunda tecrübeliysek** tercih edilir.

Yukarıdan aşağı (top-down)

+ Önce bir ana plan: "Amaç ne?", "Son ürün ne olacak?"
+ Problem alt parçalara bölünür; daha çok zaman harcanır.
+ Geri dönüş ve hata olasılığı **daha azdır**; tecrübemiz yoksa tercih edilir.

Tasarım aşamasında öğrenci kütüğünün hangi alanları içereceğine karar veririz, ama onu dizi mi, bağlı liste mi, ağaç mı ile gerçekleyeceğimizi **erteleyebiliriz**. Erteleme, daha etkili bir gerçeklemeyi seçme fırsatı verir.

TEMELLER

Kodlama ve Doğrulama

Doğruluk ispatı

Matematiksel tekniklerle program doğruluğu ispatlanır; büyük projede baştan sona ispat neredeyse imkânsız. Doğruluğu ispatlanmış algoritmaları kullanmak hatayı azaltır.

Sınama (testing)

Çalışan kod ve **tüm senaryoları** kapsayan test verisi gerekir. Sözdizimi hatası vermeden çalışan program doğru demek değildir.

Hata giderme

İspat ve testler hatayı gösterirse, tasarım ve kodlama kararlarına göre hata ayıklanır.

İşletim zamanı

Hatasız ama yavaş çalışan bir programın da fazla değeri yoktur: doğruluk ile verimlilik ayrı sorulardır.

HOCADAN *"Kodlama ne kadar geciktirilirse, arka planda o kadar iyi bir yöntem yakalanabilir."*

TEMELLER

Algoritma Nedir?

Algoritma, belirli bir görevi yerine getiren **sonlu** sayıda komut dizisidir. Her algoritma şu ölçütleri sağlamalıdır:

Ölçüt	Anlamı
Girdi	Dışarıdan sıfır ya da daha fazla nicelik alınır.
Çıktı	En az bir nicelik üretilir.
Kesinlik (definiteness)	Her komut açık ve tek anlamlıdır: "x'e 6 ya da 7 ekle" olmaz.
Sonluluk (finiteness)	Her durumda, sonlu adım sonra sona erer.
Etkinlik (effectiveness)	Her komut, kâğıt kalemle yapılabilecek kadar temel ve uygulanabilirdir.

Sonlanmayan program (ör. işletim sistemi) algoritma değil, bir **hesaplama yöntemidir**. Algoritmayı doğal dil, akış şeması ya da C ile ifade edebiliriz; biz C kullanacağız.

Horowitz–Sahni, Bölüm 1

Soyut Veri Tipi (Abstract Data Type)

Veri tipi = nesneler + işlemler. **ADT**: nesnelerin ve işlemlerin **belirtiminin** (specification), gösterimden ve gerçeklemeden (implementation) **ayrıldığı** veri tipi.

Belirtim: NE yapılır?

İşlem adları, parametre türleri, sonuç türü ve işlemin ne yaptığı. Kullanıcı yalnızca bunu bilir.

Gerçekleme: NASIL?

Verinin bellekte gösterimi (struct, dizi…) ve işlemleri yapan kod. Kullanıcıdan gizlenir.

```
ADT Kesir
  nesneler: pay/payda, payda ≠ 0
  işlemler:
    olustur(p, q)  → Kesir   /* kurucu */
    topla(a, b)    → Kesir   /* dönüştürücü */
    esit(a, b)     → Boolean /* gözlemci */
    yaz(a)                   /* raporlayıcı */
    sil(a)
```

Kurucu yeni örnek yaratır, **dönüştürücü** var olandan yeni değer üretir, **gözlemci/raporlayıcı** bilgi verir. Aynı ADT farklı veri yapılarıyla gerçeklenebilir.

Horowitz–Sahni, Bölüm 1.3

C'de Soyutlama: Başlık Dosyası + struct

```c
/* kesir.h : arayuz (NE?) */
#ifndef KESIR_H
#define KESIR_H
typedef struct kesir Kesir; /* opak */

Kesir *kesir_olustur(int pay, int payda);
Kesir *kesir_topla(const Kesir *a,
                   const Kesir *b);
int  kesir_esit(const Kesir *a,
                const Kesir *b);
void kesir_yaz(const Kesir *k);
void kesir_sil(Kesir *k);
#endif
```

```c
/* kesir.c : gercekleme (NASIL?) */
#include <stdio.h>
#include <stdlib.h>
#include "kesir.h"

struct kesir { int pay, payda; };

Kesir *kesir_olustur(int pay, int payda) {
  Kesir *k;
  if (payda == 0) return NULL;
  k = malloc(sizeof *k);
  if (k == NULL) return NULL;
  k->pay = pay;
  k->payda = payda;
  return k;
}
```

`typedef struct kesir Kesir;` **eksik (opak) tip** bildirir: `kesir.h` 'yi kullanan dosya yapının alanlarını göremez, yalnızca `Kesir *` tutar. Alanlar `kesir.c` 'de gizlidir → ADT'nin C'deki karşılığı.

Gerçeklemenin Kalanı ve Kullanımı

```
Kesir *kesir_topla(const Kesir *a,
                   const Kesir *b) {
  return kesir_olustur(
      a->pay * b->payda
        + b->pay * a->payda,
      a->payda * b->payda);
}
int kesir_esit(const Kesir *a,
               const Kesir *b) {
  return a->pay * b->payda
      == b->pay * a->payda;
}
void kesir_yaz(const Kesir *k) {
  printf("%d/%d\n", k->pay, k->payda);
}
void kesir_sil(Kesir *k) { free(k); }
```

```
/* main.c : yalnizca arayuzu bilir */
#include "kesir.h"
#include <stddef.h> /* NULL */
int main(void) {
  Kesir *a = kesir_olustur(1, 2);
  Kesir *b = kesir_olustur(1, 3);
  Kesir *t = NULL;
  if (a && b) t = kesir_topla(a, b);
  if (t) kesir_yaz(t);  /* 5/6 */
  kesir_sil(a); kesir_sil(b);
  kesir_sil(t);  /* free(NULL) zararsiz */
  return 0;
}
```

Sözleşme

`kesir_olustur` payda 0 ise ya da bellek yoksa `NULL` döner; çağıran bunu denetler.

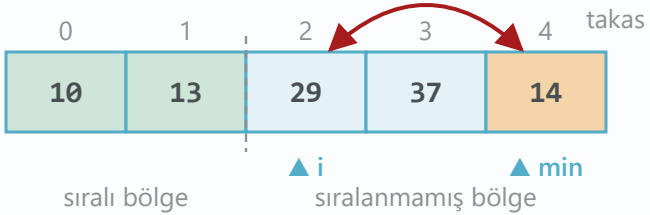
Sahiplik

Her `olustur` / `topla` sonucu çağırana aittir; `kesir_sil` ile bırakılır.

Seçimli Sıralama (Selection Sort)

Sıralı olmayan bir diziyi küçükten büyüğe sıralamak istiyoruz. Her turda **kalan kısmın en küçüğü** bulunur ve başa (i. konuma) taşınır:

```
for (i = 0; i < n; i++) {
    list[i] .. list[n-1] arasini incele;
    en kucuk tamsayi list[min] olsun;
    list[i] ile list[min]'i takas et;
}
```



HOCADAN *"Yazılmış algoritmaları beğenmeyip daha iyi alternatifler sunabilecek seviyeye gelmek gerekir."*

Seçimli Sıralama: C Gerçeklemesi

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int list[], int n);

int main(void) {
  int i, n, list[MAX_SIZE];
  printf("Dizi boyunu giriniz: ");
  if (scanf("%d", &n) != 1 || n < 1 || n > MAX_SIZE) {
    printf("Hatali n degeri!\n");
    return 1;
  }
  for (i = 0; i < n; i++) list[i] = rand() % 100;
  sort(list, n);
  for (i = 0; i < n; i++) printf("%d ", list[i]);
  return 0;
}
```

```c
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i], list[min], temp);
  }
}
```

Karşılaştırma sayısı: $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 \rightarrow \mathbf{O(n^2)}$. Takas en fazla $n-1$ kez.

SEÇİMLİ SIRALAMA · SIK YAPILAN HATA

Sık Görülen Hatalı `sort()` Nerede Ayrılıyor?

```
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = 0; j < n; j++)
      if (list[j] > list[min]) {
        min = j;
        SWAP(list[i], list[min], temp);
      }
  }
}
```

Girdi	Sürüm	Karş.	Takas*
{29,10,14,37,13}	hatalı	25	7
	seçimli	10	3
100 rastgele	hatalı	10000	2561
	seçimli	4950	94
500…1 (azalan)	hatalı	250000	124750
	seçimli	124750	250

Soldaki hatalı sürüm diziyi **yine sıralar** (n ≤ 7 için tüm permütasyonlar denendi), ama sözde koddaki seçimli sıralama **değildir**: iç döngü `j = 0` 'dan başlar, takas döngünün içinde yapılır. Doğru sürüm (`j = i + 1` , takas iç döngüden sonra) bu bölümde kullanılan sürümdür. *Takas: iki farklı hücrenin yer değiştirdiği `SWAP` 'lar (`i ≠ min`).

SWAP: Makro mu, Fonksiyon mu?

Makro

```
#define SWAP(x, y, t) \
        ((t) = (x), (x) = (y), (y) = (t))

int temp;
SWAP(list[i], list[min], temp);
```

+ Çağrı maliyeti yok: önişlemci metni yerine koyar.
+ Geçici değişken `t` dışarıdan verilir.

Fonksiyon

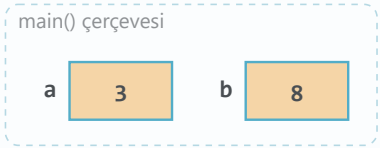
```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

swap(&list[i], &list[min]);
```

+ Tür denetimi var; `temp` yerel olur.
+ Değişkenlerin **adresleri** gönderilmelidir.

Makro, argümanlarını birden çok kez değerlendirir: `SWAP(a[i++], a[j], t)` `i`'yi iki kez artırır! Fonksiyona da `SWAP` adı verilirse makroyla çakışır; bu yüzden fonksiyonun adı `swap`.

Kod İzleme: `swap(&a, &b)`



`main`'in çerçevesinde **a = 3**, **b = 8**.

Adım 1 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
▶    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

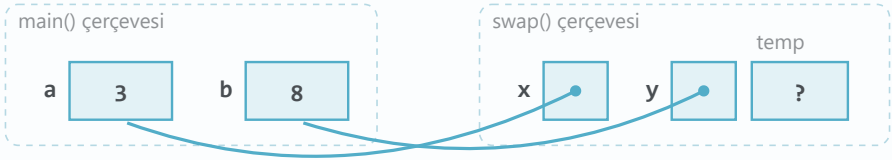
`a = 3` `b = 8`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

Kod İzleme: `swap(&a, &b)`



Çağrı: değerler değil **adresler** kopyalanır. x → a, y → b.

Adım 2 / 6

```
▶ void swap(int *x, int *y) {
2   int temp = *x;
3   *x = *y;
4   *y = temp;
5 }
6
7 int main(void) {
8   int a = 3, b = 8;
▶   swap(&a, &b);
10  printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

`x = &a` `y = &b`

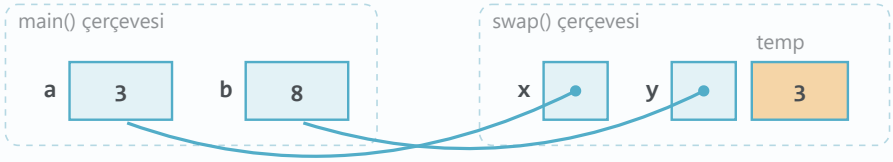
ÇAĞRI YIĞINI

`swap(&a, &b)`

`main()`

ÇIKTI

Kod İzleme: `swap(&a, &b)`



`temp = *x` : x'in gösterdiği yerdeki değer (3) okunur.

Adım 3 / 6

```
1  void swap(int *x, int *y) {
▶    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

`x = &a` `y = &b` `temp = 3`

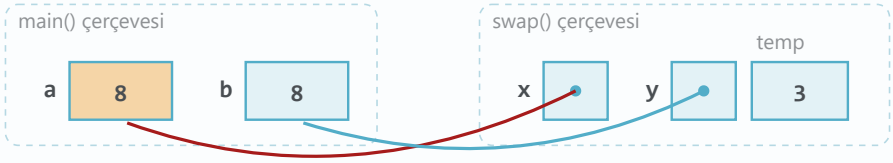
ÇAĞRI YIĞINI

`swap(&a, &b)`

`main()`

ÇIKTI

Kod İzleme: `swap(&a, &b)`



`*x = *y` : y'nin gösterdiği 8, x'in gösterdiği yere, yani **a**'ya yazılır.

Adım 4 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
▶    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

`x = &a` `y = &b` `temp = 3`

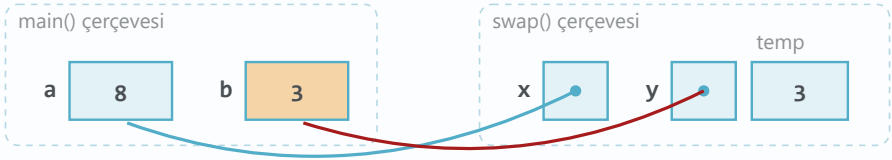
ÇAĞRI YIĞINI

`swap(&a, &b)`

`main()`

ÇIKTI

Kod İzleme: `swap(&a, &b)`



`*y = temp` : b'ye 3 yazılır. Takas **main'in değişkenlerinde** gerçekleşti.

Adım 5 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
▶    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
10   printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

`x = &a` `y = &b` `temp = 3`

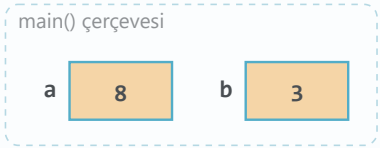
ÇAĞRI YIĞINI

`swap(&a, &b)`

`main()`

ÇIKTI

Kod İzleme: `swap(&a, &b)`



swap'ın çerçevesi yok oldu (x, y, temp gitti) ama etkisi kaldı.

Adım 6 / 6

```
1  void swap(int *x, int *y) {
2    int temp = *x;
3    *x = *y;
4    *y = temp;
5  }
6
7  int main(void) {
8    int a = 3, b = 8;
9    swap(&a, &b);
▶    printf("%d %d\n", a, b);
11 }
```

DEĞİŞKENLER

`a = 8` `b = 3`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
8 3
```

PERFORMANS ANALİZİ

Bir Programı Nasıl Değerlendiririz?

1. İsteneni karşılıyor mu?
2. Doğru çalışıyor mu?
3. Nasıl işletileceği ve kullanılacağı belgelenmiş mi (açıklama satırları dâhil)?
4. Mantıksal sonuç üretmek için fonksiyonları etkili kullanıyor mu?
5. Kod okunabilir mi?
6. **Ana ve ikincil belleği etkili kullanıyor mu?**
7. **İşletim zamanı kabul edilebilir mi?**

1 · Performans analizi

Makineden **bağımsız**: zaman ve bellek için tahmin (karmaşıklık).

2 · Performans ölçümü

Makineye **bağlı**: gerçek işletim süresi ölçülür.

Son iki soru **performans değerlendirmenin** konusudur; ilk beşi de en az onlar kadar önemlidir.

PERFORMANS ANALİZİ

Alan Karmaşıklığı (Space Complexity)

Programın işini tamamlaması için gereken bellek miktarı: **$S(P) = c + S_P(I)$**

Sabit kısım c

Girdi/çıktı büyüklüğünden bağımsız: kodun kendisi, basit değişkenler, sabitler.

Değişken kısım $S_P(I)$

I girdisinin özelliklerine (çoğunlukla boyutuna) bağlı kısım. Çözümlemede **bununla** ilgileniriz.

```
float abc(float a, float b, float c) {
  return a + b + b * c
         + (a + b - c) / (a + b) + 4.00;
}
/* yalniz sabit bellek: S_abc(I) = 0 */
```

HOCADAN *"Kod, her şey kısıtlıymış gibi yazılmalı. Bellek nasıl olsa bol diye boşa bellek harcanamaz."*

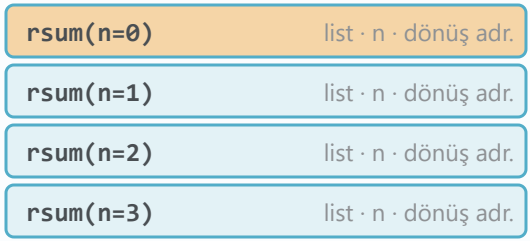
`sum` ve `rsum`: Değişken Bellek

```
float sum(float list[], int n) {
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    tempsum += list[i];
  return tempsum;
}

float rsum(float list[], int n) {
  if (n)
    return rsum(list, n - 1) + list[n - 1];
  return 0;
}
```

- `sum`'a dizinin **ilk öğesinin adresi** gider → $S_{sum}(n) = 0$. Öğeler değerle kopyalansaydı $S_{sum}(n) = n$ olurdu.
- `rsum(list, n)` toplam n+1 kez çağrılır; her çağrı yığında bir çerçeve açar → $S_{rsum}(n) \propto n$.

sistem yığını



Her çağrı bir çerçeve:

16 bitlik makinede 2 + 2 + 2 = 6 bayt
n = 1000 → en az 6000 bayt
64 bitte bir çerçeve tipik olarak ≥ 24 bayt

PERFORMANS ANALİZİ

Zaman Karmaşıklığı (Time Complexity)

T(P) = derleme zamanı + işletim zamanı. Derleme bir kez yapılır; biz **işletim zamanı** t_P ile ilgileniriz.

$$t_P(n) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n)$$

Sabitler c

Her işlemin süresi: derleyiciye ve makineye bağlı, önceden bilmek zor.

Adım sayısı

Bunun yerine **program adımlarını** sayarız: makineden bağımsız, anlamlı bir birim.

HOCADAN *"Performansı yüksek programlar yazmak neden bu kadar önemli? Bir savaş uçağının bombalama sistemi programlanıyor olsun. Bu program 3 sn'de, karşı tarafınki 2 sn'de bomba bırakıyorsa savaş baştan kaybedilmiş demektir."*

PERFORMANS ANALİZİ

Adım Sayma Tablosu: `sum()`

Deyim	s/e	Sıklık	Toplam
`float sum(float list[], int n) {`	0	0	0
`  float tempsum = 0;`	1	1	1
`  int i;`	0	0	0
`  for (i = 0; i < n; i++)`	1	n + 1	n + 1
`  tempsum += list[i];`	1	n	n
`  return tempsum;`	1	1	1
Toplam			**2n + 3**

s/e (steps per execution): deyimin bir kez işletilmesi kaç adım. `for` testi n+1 kez yapılır: n kez "devam", bir kez de "bitti". Bildirimler adım sayılmaz.

Kod İzleme: `count` ile `sum(list, 3)`



Global `count = 0`. Her **program adımında** bir artacak.

Adım 1 / 10

```
▶ float sum(float list[], int n) {
2   float tempsum = 0; count++;
3   int i;
4   for (i = 0; i < n; i++) {
5     count++;              /* for testi */
6     tempsum += list[i]; count++;
7   }
8   count++;  /* for'un son testi */
9   count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`list = [3, 1, 4]` `n = 3`

`count = 0`

Kod İzleme: `count` ile `sum(list, 3)`



Atama bir adım → count = 1.

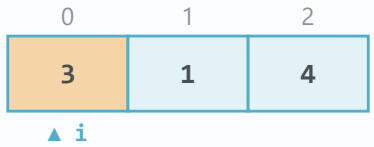
Adım 2 / 10

```
1  float sum(float list[], int n) {
▶    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `tempsum = 0` `count = 1`

Kod İzleme: `count` ile `sum(list, 3)`



for testi **0 < 3** doğru → bir adım (count = 2).

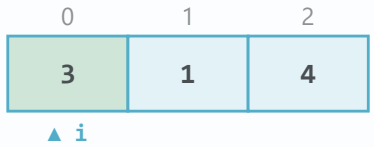
Adım 3 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 0` `tempsum = 0` `count = 2`

Kod İzleme: `count` ile `sum(list, 3)`



tempsum += list[0] → **3**; count = 3.

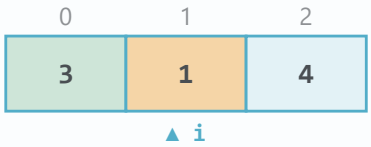
Adım 4 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 0` `tempsum = 3` `count = 3`

Kod İzleme: `count` ile `sum(list, 3)`



for testi **1 < 3** doğru → bir adım (count = 4).

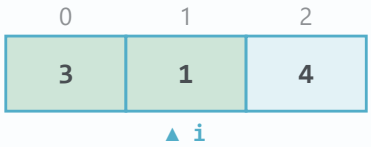
Adım 5 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 1` `tempsum = 3` `count = 4`

Kod İzleme: `count` ile `sum(list, 3)`



tempsum += list[1] → **4**; count = 5.

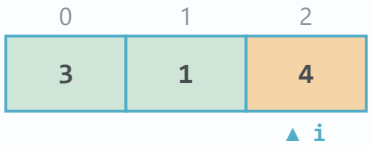
Adım 6 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 1` `tempsum = 4` `count = 5`

Kod İzleme: `count` ile `sum(list, 3)`



for testi **2 < 3** doğru → bir adım (count = 6).

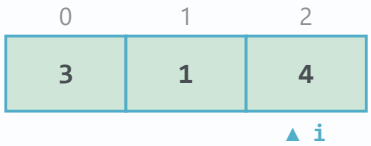
Adım 7 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
▶    for (i = 0; i < n; i++) {
▶      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 2` `tempsum = 4` `count = 6`

Kod İzleme: `count` ile `sum(list, 3)`



tempsum += list[2] → **8**; count = 7.

Adım 8 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
▶      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 2` `tempsum = 8` `count = 7`

Kod İzleme: `count` ile `sum(list, 3)`



Son test **3 < 3** yanlış; o da bir adım → count = 8.

Adım 9 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
▶    count++;  /* for'un son testi */
9    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `i = 3` `tempsum = 8` `count = 8`

Kod İzleme: `count` ile `sum(list, 3)`



return: count = **9 = 2·3 + 3**. Tablodaki 2n + 3 doğrulandı.

Adım 10 / 10

```
1  float sum(float list[], int n) {
2    float tempsum = 0; count++;
3    int i;
4    for (i = 0; i < n; i++) {
5      count++;              /* for testi */
6      tempsum += list[i]; count++;
7    }
8    count++;  /* for'un son testi */
▶    count++;  return tempsum;
10 }
```

DEĞİŞKENLER

`n = 3` `tempsum = 8` `count = 9`

PERFORMANS ANALİZİ

Adım Sayma Tablosu: `rsum()`

Deyim	s/e	Sıklık	Toplam
`float rsum(float list[], int n) {`	0	0	0
`  if (n)`	1	n + 1	n + 1
`  return rsum(list, n-1) + list[n-1];`	1	n	n
`  return 0;`	1	1	1
Toplam			**2n + 2**

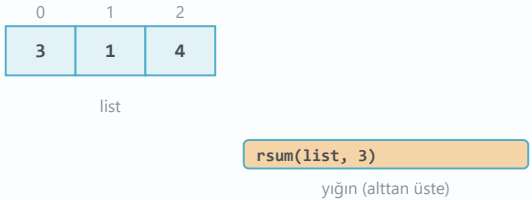
Adım sayısı yanıltabilir

rsum (2n+2) sum'dan (2n+3) "daha az adım" görünür; ama her özyinelemeli çağrının (çerçeve açma, dönüş) maliyeti bir döngü turundan fazladır.

İkisi de O(n)

Önemli olan **n büyüdükçe** adım sayısının nasıl büyüdüğüdür: ikisi de doğrusal.

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



rsum(list, 3): if testi (count = 1); n = 3 ≠ 0.

Adım 1 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if testi */
▶   if (n) {
4     count++;            /* toplama + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

DEĞİŞKENLER

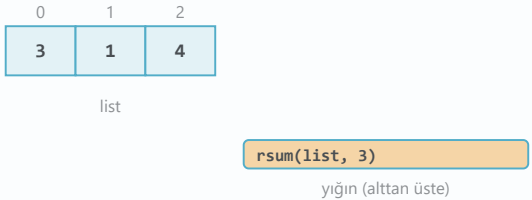
`n = 3` `count = 1`

ÇAĞRI YIĞINI

`rsum(list, 3)`

`main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



count = 2; önce **rsum(list, 2)** hesaplanmalı → yeni çerçeve.

Adım 2 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
▶     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

DEĞİŞKENLER

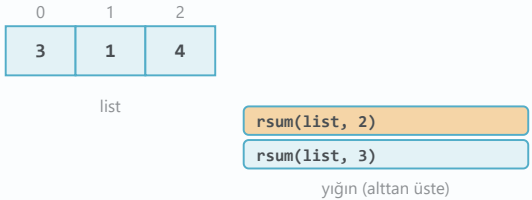
`n = 3` `count = 2`

ÇAĞRI YIĞINI

`rsum(list, 3)`

`main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



rsum(list, 2): if testi (count = 3); n = 2 ≠ 0.

Adım 3 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if testi */
▶   if (n) {
4     count++;            /* toplama + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

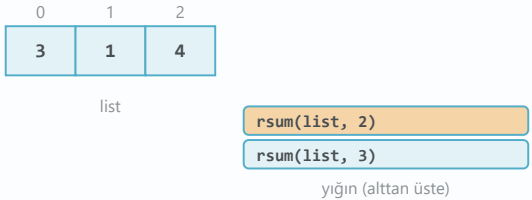
DEĞİŞKENLER

`n = 2` `count = 3`

ÇAĞRI YIĞINI

- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



count = 4; önce **rsum(list, 1)** hesaplanmalı → yeni çerçeve.

Adım 4 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
▶     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

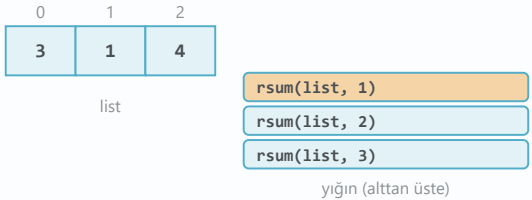
DEĞİŞKENLER

`n = 2` `count = 4`

ÇAĞRI YIĞINI

- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



rsum(list, 1): if testi (count = 5); n = 1 ≠ 0.

Adım 5 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if testi */
▶   if (n) {
4     count++;            /* toplama + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

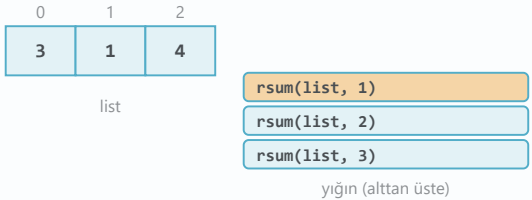
DEĞİŞKENLER

`n = 1` `count = 5`

ÇAĞRI YIĞINI

- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



count = 6; önce **rsum(list, 0)** hesaplanmalı → yeni çerçeve.

Adım 6 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
▶     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

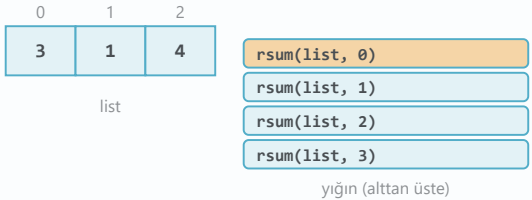
DEĞİŞKENLER

`n = 1` `count = 6`

ÇAĞRI YIĞINI

- **rsum(list, 1)**
- rsum(list, 2)
- rsum(list, 3)
- main()

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



rsum(list, 0): if testi (count = 7); n = 0 → **taban durum**.

Adım 7 / 11

```
1 float rsum(float list[], int n) {
▶   count++;              /* if testi */
▶   if (n) {
4     count++;            /* toplama + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

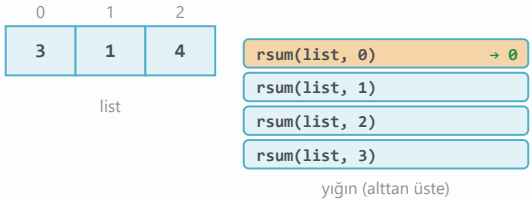
DEĞİŞKENLER

`n = 0` `count = 7`

ÇAĞRI YIĞINI

- `rsum(list, 0)`
- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



count = 8; **return 0**. Yığın en derin hâlinde: 4 çerçeve.

Adım 8 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
4     count++;              /* toplama + return */
5     return rsum(list, n - 1) + list[n - 1];
6   }
▶   count++;
▶   return 0;
9 }
```

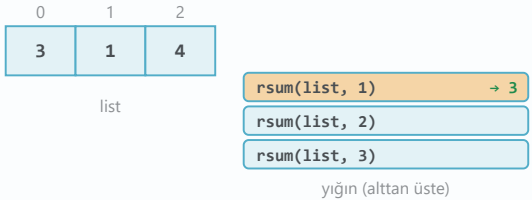
DEĞİŞKENLER

`n = 0` `count = 8`

ÇAĞRI YIĞINI

- `rsum(list, 0)`
- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



Dönen değere list[0] = 3 eklenir → rsum(list, 1) = **3**.

Adım 9 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
4     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

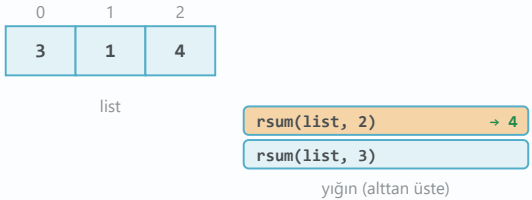
DEĞİŞKENLER

`n = 1` `count = 8`

ÇAĞRI YIĞINI

- `rsum(list, 1)`
- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



Dönen değere list[1] = 1 eklenir → rsum(list, 2) = **4**.

Adım 10 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
4     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

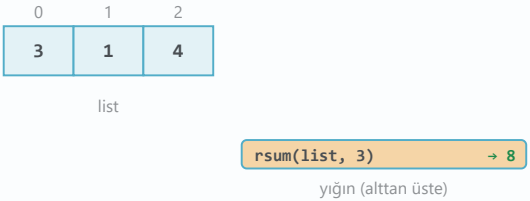
DEĞİŞKENLER

`n = 2` `count = 8`

ÇAĞRI YIĞINI

- `rsum(list, 2)`
- `rsum(list, 3)`
- `main()`

Kod İzleme: `rsum(list, 3)` ve çağrı yığını



Dönen değere list[2] = 4 eklenir → rsum(list, 3) = **8**. count = **8 = 2·3 + 2**.

Adım 11 / 11

```
1 float rsum(float list[], int n) {
2   count++;                /* if testi */
3   if (n) {
4     count++;              /* toplama + return */
▶     return rsum(list, n - 1) + list[n - 1];
6   }
7   count++;
8   return 0;
9 }
```

DEĞİŞKENLER

`n = 3` `count = 8`

ÇAĞRI YIĞINI

`rsum(list, 3)`

`main()`

PERFORMANS ANALİZİ

Adım Sayma: Matris Toplama `add()`

Deyim	Sıklık
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE], int c[][MAX_SIZE], int rows, int cols) {`	0
`int i, j;`	0
`for (i = 0; i < rows; i++)`	rows + 1
`for (j = 0; j < cols; j++)`	rows · (cols + 1)
`c[i][j] = a[i][j] + b[i][j];`	rows · cols
Toplam	**2·rows·cols + 2·rows + 1**

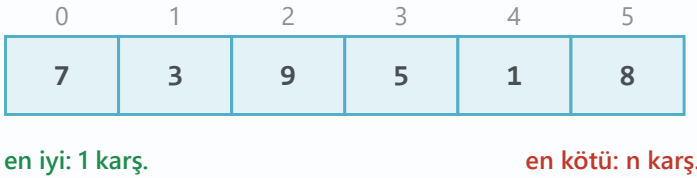
Sonuç **O(rows·cols)**; kare matriste (rows = cols = n) **O(n^2)**. Dış döngü daha küçük boyutta dönerse 2·rows terimi küçülür: rows ≫ cols iken döngüler yer değiştirilirse sayım 2·rows·cols + 2·cols + 1 olur. Matris çarpımı ise **O(n^3)**.

“O(rows, cols)” yazmak yanlıştır; doğrusu çarpım: O(rows·cols).

PERFORMANS ANALİZİ

En İyi, Ortalama ve En Kötü Durum

Aynı n için adım sayısı **girdinin kendisine** de bağlı olabilir. Doğrusal aramada:



Durum	Ne zaman?	Karşılaştırma
En iyi	Aranan ilk hücrede	$1 \rightarrow \Theta(1)$
Ortalama	Aranan var ve her konum eşit olasılıklı	$(n + 1) / 2 \rightarrow \Theta(n)$
En kötü	Aranan son hücrede ya da hiç yok	$n \rightarrow \Theta(n)$

Genellikle **en kötü durumu** veririz: her girdi için garanti bir üst sınırdır. Ortalama durum, girdinin olasılık dağılımını bilmeyi gerektirir.

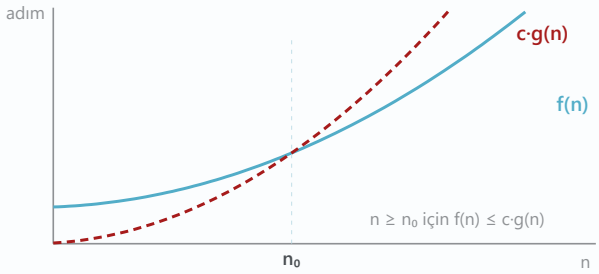
CLRS Bölüm 2.2

ASİMPTOTİK GÖSTERİM

Büyük O (Big-O) Gösterimi

Tanım: Tüm $n \geq n_0$ için $f(n) \leq c \cdot g(n)$ olacak biçimde **$c > 0$** ve **n_0** sabitleri varsa **$f(n) = O(g(n))$** yazılır: "algoritmanın karmaşıklığı g(n) derecesindedir".

- $T_{sum}(n) = 2n + 3 = O(n)$
- $T_{rsum}(n) = 2n + 2 = O(n)$
- $T_{add} = 2 \cdot rows \cdot cols + 2 \cdot rows + 1 = O(rows \cdot cols) \rightarrow O(n^2)$



HOCADAN *"Zaman, bellekten daha önemli!"*

O bir **üst sınırdır**; "=" burada "elemanıdır" anlamında tek yönlü okunur.

ASİMPTOTİK GÖSTERİM

Örnek: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (n \ge 7 \text{ için } 7n \le n^2)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (n \ge 7 \text{ için } 7n^2 \le n^3)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

Baskın olmayan 7n terimini n^2 ile sınırlarız: $n \ge 7$ olduğunda $7n \le n^2$.

Adım 1 / 4

1 · Üst sınır

İşletim zamanı hiçbir zaman $4n^3$ adımı aşmaz ($n \ge 7$).

2 · Kübik büyüme

100 sayı 2 sn'de sıralanıyorsa 300 sayı en fazla $2 \cdot 3^3$ = **54 sn** sürer (6 sn değil!).

Örnek: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (n \ge 7 \text{ için } 7n \le n^2)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (n \ge 7 \text{ için } 7n^2 \le n^3)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

$6n^2 + n^2 = 7n^2$.

1 · Üst sınır

İşletim zamanı hiçbir zaman $4n^3$ adımı aşmaz ($n \ge 7$).

2 · Kübik büyüme

100 sayı 2 sn'de sıralanıyorsa 300 sayı en fazla $2 \cdot 3^3$ = **54 sn** sürer (6 sn değil!).

Örnek: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (n \ge 7 \text{ için } 7n \le n^2)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (n \ge 7 \text{ için } 7n^2 \le n^3)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

Aynı hile bir kez daha: $n \ge 7$ için $7n^2 \le n^3$.

1 · Üst sınır

İşletim zamanı hiçbir zaman $4n^3$ adımı aşmaz ($n \ge 7$).

2 · Kübik büyüme

100 sayı 2 sn'de sıralanıyorsa 300 sayı en fazla $2 \cdot 3^3$ = **54 sn** sürer (6 sn değil!).

Örnek: $f(n) = 3n^3 + 6n^2 + 7n = O(n^3)$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 6n^2 + n^2 \quad (n \ge 7 \text{ için } 7n \le n^2)$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + 7n^2$$

$$3n^3 + 6n^2 + 7n \le 3n^3 + n^3 \quad (n \ge 7 \text{ için } 7n^2 \le n^3)$$

$$3n^3 + 6n^2 + 7n \le 4n^3 \Rightarrow f(n) = O(n^3) \quad (c = 4,\ n_0 = 7)$$

$c = 4$ ve $n_0 = 7$ bulundu: tanım sağlandı.

1 · Üst sınır

İşletim zamanı hiçbir zaman $4n^3$ adımı aşmaz ($n \ge 7$).

2 · Kübik büyüme

100 sayı 2 sn'de sıralanıyorsa 300 sayı en fazla $2 \cdot 3^3$ = **54 sn** sürer (6 sn değil!).

ASİMPTOTİK GÖSTERİM

Alt Sınır Ω ve Sıkı Sınır Θ

Gösterim	Tanım (tüm $n \geq n_0$ için)	Anlamı
f(n) = O(g(n))	$f(n) \leq c \cdot g(n)$	g, f'nin **üst** sınırı
f(n) = Ω(g(n))	$f(n) \geq c \cdot g(n)$	g, f'nin **alt** sınırı
f(n) = Θ(g(n))	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	hem O hem Ω: **sıkı** sınır

- $3n + 2 = \Theta(n)$: $n \geq 2$ için $3n \leq 3n + 2 \leq 4n$.
- $3n + 2 = O(n^2)$ de doğrudur ama bilgi vermez; mümkün olan en küçük g(n)'yi yazarız.
- Seçimli sıralama her girdide $n(n-1)/2$ karşılaştırma yapar → $\Theta(n^2)$.

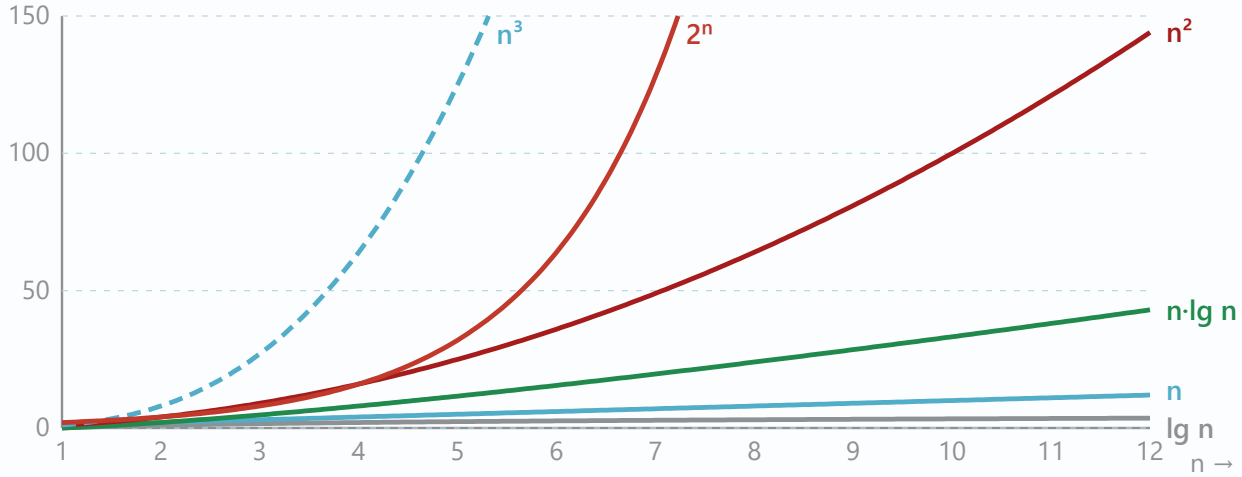
HOCADAN *"Bir sınav sorusu O(n)'de çözülmüş, sonuna da m kez dönen boş bir döngü eklenmiş: soru "O(n+m)'de çözün" diyor diye. Gerek yok. İstenen O(n+m) ise, daha iyisi yapılabiliyorsa ne âlâ!"*

Alışılmış Karmaşıklıklar (Practical Complexities)

Karmaşıklık	Adı	n=1	n=2	n=4	n=8	n=16	n=32
1	Sabit (constant)	1	1	1	1	1	1
lg n	Logaritmik	0	1	2	3	4	5
n	Doğrusal (linear)	1	2	4	8	16	32
n·lg n	Log-doğrusal	0	2	8	24	64	160
n^2	Karesel (quadratic)	1	4	16	64	256	1.024
n^3	Kübik (cubic)	1	8	64	512	4.096	32.768
2^n	Üstel (exponential)	2	4	16	256	65.536	$4{,}29 \cdot 10^9$
n!	Faktöriyel	1	2	24	40.320	$2{,}09 \cdot 10^{13}$	$2{,}63 \cdot 10^{35}$

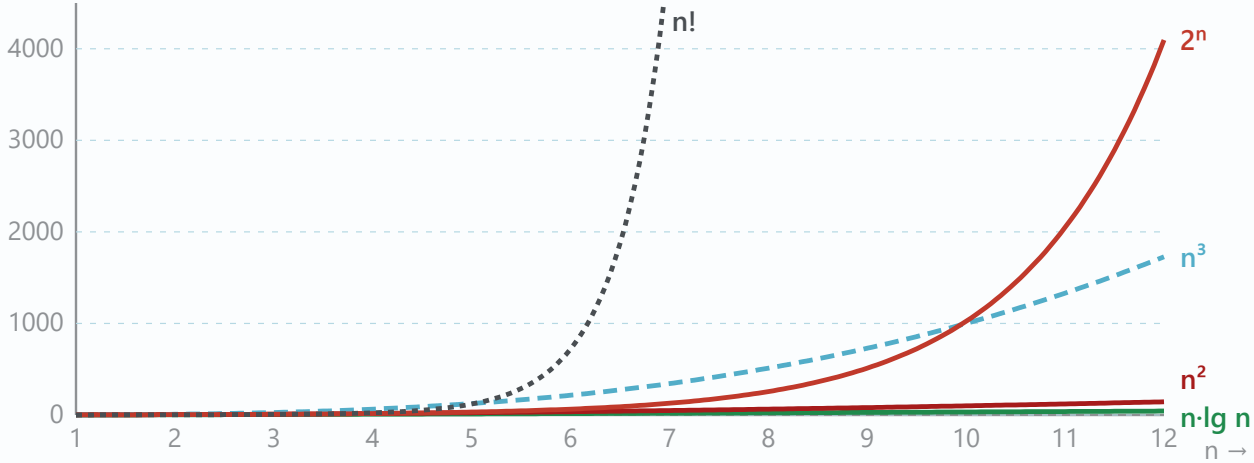
lg n = $\log_2 n$. Küçük n değerleri yanıltıcıdır: n = 4'te $2^n = n^2$, n = 8'de $2^n = 256 > n^2 = 64$. Belli bir noktadan sonra üstel ve faktöriyel terimler her şeyi ezer.

n Büyüdükçe: Büyüme Grafiği



Dikey ölçek 0–150: lg n, n ve n·lg n birbirinden ayrışıyor; n^2 hızla tırmanıyor, n^3 ve 2^n çoktan dışarıda.

n Büyüdükçe: Büyüme Grafiği



Ölçek 0–4500: $n^3 = 1728$ ($n = 12$), $2^n = 4096$ ($n = 12$); $n!$ ise $n = 7$'de 5040 ile grafikten çıkar. Doğrusal fonksiyonlar tabana yapışmış görünür.

ASİMPTOTİK GÖSTERİM

Saniyede 1 Milyar Komut İşleyen Makinede Süreler

n	n	lg n	n·lg n	n^2	n^3	n^4	n^{10}	2^n
10	0,01 µs	0,00332 µs	0,0332 µs	0,1 µs	1 µs	10,0 µs	10,0 sn	1,02 µs
20	0,02 µs	0,00432 µs	0,0864 µs	0,4 µs	8 µs	160 µs	2,84 saat	1,05 ms
30	0,03 µs	0,00491 µs	0,147 µs	0,9 µs	27,0 µs	810 µs	6,83 gün	1,07 sn
40	0,04 µs	0,00532 µs	0,213 µs	1,6 µs	64,0 µs	2,56 ms	4,05 ay	18,3 dk
50	0,05 µs	0,00564 µs	0,282 µs	2,5 µs	125 µs	6,25 ms	3,1 yıl	13,0 gün
100	0,1 µs	0,00664 µs	0,664 µs	10,0 µs	1 ms	100 ms	31,7 asır	$4{,}02 \cdot 10^{11}$ asır
1000	1 µs	0,00997 µs	9,97 µs	1 ms	1 sn	16,7 dk	$3{,}17 \cdot 10^{11}$ asır	$3{,}40 \cdot 10^{282}$ asır
10000	10,0 µs	0,0133 µs	133 µs	100 ms	16,7 dk	3,86 ay	$3{,}17 \cdot 10^{21}$ asır	$6{,}33 \cdot 10^{2991}$ asır

Her adım 1 ns kabul edildi; 1 yıl = 365 gün, 1 ay = 30 gün. n = 10000 için 2^n yaklaşık 10^{2992} asır.

ASİMPTOTİK GÖSTERİM

Amortize Analize Kısa Giriş

Bazı işlemler **bazen pahalı**, çoğunlukla ucuzdur. Amortize analiz, n işlemlik bir dizinin **toplam** maliyetini n'e böler. Örnek: doluysa kapasitesi **ikiye katlanan** dizi (`realloc`); bu dizinin adım adım izlemesi ve kopya sayımı dinamik dizilerle birlikte işlenir.

n eklemede kopyalamalar 1 + 2 + 4 + … < 2n → toplam < 3n. Tek bir ekleme O(n) sürebilir, ama **ekleme başına amortize maliyet O(1)**. (Kapasiteyi sabit sayıda artırsaydık O(n) olurdu.)

CLRS 3. baskı, Bölüm 17 (Amortized Analysis)

SİHİRLİ KARE

Sihirli Kare (Magic Square) Problemi

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Her satır, sütun ve köşegen: 65

1'den n^2'ye kadar sayılar n×n kareye, her satır, sütun ve iki ana köşegenin toplamı eşit olacak biçimde yerleştirilir: toplam $\mathbf{n(n^2+1)/2}$ (5×5 için 65).

Coxeter kuralı (n tek; Siyam ya da de la Loubère yöntemi olarak da bilinir): İlk satırın ortasına 1 yazılır. Hep **sol üst** çapraza gidilir; kare doluysa bulunulan karenin **altına** inilir. Tablonun dışına taşılırsa karşı kenardan dönülür (sarmal). n^2 adımda biter → $\mathbf{O(n^2)}$.

Kaba kuvvet (brute force): $(n^2)!$ diziliş. $25! \approx 1{,}55 \cdot 10^{25}$ olasılık; saniyede 10^9 deneme ile $\approx 1{,}55 \cdot 10^{16}$ sn ≈ **4,9 milyon asır**.

“Bilgisayar Hızlı, Hemen Çözer!” mi?

> *Sık duyulan bir itiraz: 10 GHz’lik yeni bir bilgisayar bu problemi hemen çözer; algoritmayla uğraşmaya gerek yok.*

10 kat hızlı makine

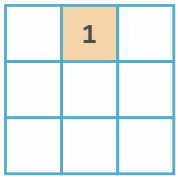
4,9 milyon asır → yaklaşık **490 bin asır**. Hâlâ umutsuz.

Doğru algoritma

Coxeter kuralı: 25 adım, mikrosaniyenin altında. **Algoritma, donanımdan önce gelir.**

Hesap: $25! / (10^{10} \cdot 3{,}15 \cdot 10^{9}) \approx 4{,}9 \cdot 10^{5}$ asır.

Kod İzleme: `sihirli(3, k)`



İlk satırın ortası: **i = 0, j = 1** → 1 yazılır.

Adım 1 / 10

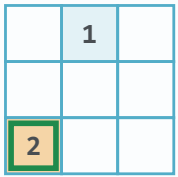
```
1  void sihirli(int n, int k[][MAX]) {
▶    int i = 0, j = (n - 1) / 2, s, r, c;
▶    k[i][j] = 1;        /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
5      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
8      else { i = r; j = c; }
9      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`n = 3` `i = 0` `j = 1`

`k = 1 yerleşti`

Kod İzleme: `sihirli(3, k)`



s = 2: sol üst çapraz (2, 0) boş (satır taştı → en alt satır) → oraya geçilir.

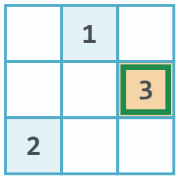
Adım 2 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 2` `i = 2` `j = 0` `r = 2` `c = 0`

Kod İzleme: `sihirli(3, k)`



s = 3: sol üst çapraz (1, 2) boş (sütun taştı → en sağ sütun) → oraya geçilir.

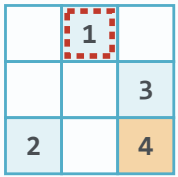
Adım 3 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 3` `i = 1` `j = 2` `r = 1` `c = 2`

Kod İzleme: `sihirli(3, k)`



s = 4: sol üst çapraz (0, 1) **dolu** (orada 1 var) → bir aşağı: (2, 2).

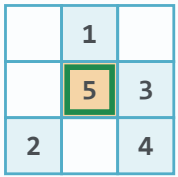
Adım 4 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
▶      if (k[r][c]) i = (i + 1) % n; /* in */
8      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 4` `i = 2` `j = 2` `r = 0` `c = 1`

Kod İzleme: `sihirli(3, k)`



s = 5: sol üst çapraz (1, 1) boş → oraya geçilir.

Adım 5 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 5` `i = 1` `j = 1` `r = 1` `c = 1`

Kod İzleme: `sihirli(3, k)`

6	1	
	5	3
2		4

s = 6: sol üst çapraz (0, 0) boş → oraya geçilir.

Adım 6 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 6` `i = 0` `j = 0` `r = 0` `c = 0`

Kod İzleme: `sihirli(3, k)`

6	1	
7	5	3
2		4

s = 7: sol üst çapraz (2, 2) **dolu** (orada 4 var) → bir aşağı: (1, 0).

Adım 7 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
▶      if (k[r][c]) i = (i + 1) % n; /* in */
8      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 7` `i = 1` `j = 0` `r = 2` `c = 2`

Kod İzleme: `sihirli(3, k)`

6	1	8
7	5	3
2		4

s = 8: sol üst çapraz (0, 2) boş (sütun taştı → en sağ sütun) → oraya geçilir.

Adım 8 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 8` `i = 0` `j = 2` `r = 0` `c = 2`

Kod İzleme: `sihirli(3, k)`

6	1	8
7	5	3
2	9	4

s = 9: sol üst çapraz (2, 1) boş (satır taştı → en alt satır) → oraya geçilir.

Adım 9 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
4    for (s = 2; s <= n * n; s++) {
▶      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
▶      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
▶      else { i = r; j = c; }
▶      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 9` `i = 2` `j = 1` `r = 2` `c = 1`

Kod İzleme: `sihirli(3, k)`

6	1	8
7	5	3
2	9	4

s = 10 > 9: döngü biter. Her satır, sütun ve köşegenin toplamı **15** = 3·(9+1)/2.

Adım 10 / 10

```
1  void sihirli(int n, int k[][MAX]) {
2    int i = 0, j = (n - 1) / 2, s, r, c;
3    k[i][j] = 1;          /* k sifirlanmis olmali */
▶    for (s = 2; s <= n * n; s++) {
5      r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
6      c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
7      if (k[r][c]) i = (i + 1) % n; /* in */
8      else { i = r; j = c; }
9      k[i][j] = s;
10   }
11 }
```

DEĞİŞKENLER

`s = 10` `i = 2` `j = 1`

Performans Ölçümü: İki Yöntem

	1 · İşleyici zamanı (clock)	2 · Takvim zamanı (time)
Başlangıç	`start = clock();`	`start = time(NULL);`
Bitiş	`stop = clock();`	`stop = time(NULL);`
Tür	`clock_t`	`time_t`
Sonuç (sn)	`(double)(stop - start) / CLOCKS_PER_SEC`	`difftime(stop, start)`

clock()

Program için harcanan işlemci saat vuruşu (clock tick) sayısı. Çözünürlüğü sistemin vuruş aralığıyla sınırlıdır.

time()

Duvar saati; saniye çözünürlüğü. Uzun işler için uygun, kısa işler için kaba.

`CLK_TCK` eski, standart dışı bir addır; standart C'de **`CLOCKS_PER_SEC`** kullanılır. Ölçerken çevre birimleriyle (ekran, disk, yazıcı) işlem yavaştır: **`printf`'ler ölçülen bölgenin dışında tutulmalıdır.**

PERFORMANS ÖLÇÜMÜ

Uygulama 1: Seçimli Sıralamayı Ölçmek

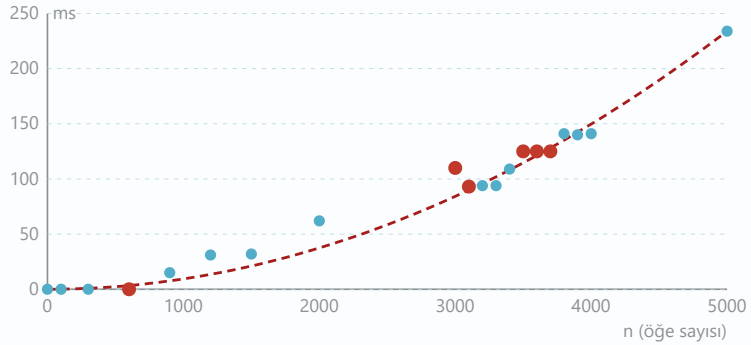
```c
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 5001
#define ITERATIONS 20
void sort(int list[], int n);   /* secmeli siralama */

int main(void) {
  int i, j, list[MAX_SIZE];
  int sizelist[] = { 0, 100, 300, 600, 900, 1200, 1500, 2000,
    3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800,
    3900, 4000, 5000 };
  clock_t start, stop;  double ms;
  for (i = 0; i < ITERATIONS; i++) {
    for (j = 0; j < sizelist[i]; j++)
      list[j] = sizelist[i] - j;   /* azalan dizi */
    start = clock();
    sort(list, sizelist[i]);
    stop = clock();
    ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
    printf("%4d eleman: %6.1f ms\n", sizelist[i], ms);
  }
  return 0;
}
```

- 20 farklı n; her biri için **büyükten küçüğe** dizilmiş dizi.
- Yalnızca `sort` çağrısı iki `clock()` arasında.
- `(int)(1000*(stop-start)/CLK_TCK)` yazılırsa tamsayı bölmesi küsuratı atar; `1000.0` ile **double** aritmetik daha doğru.

PERFORMANS ÖLÇÜMÜ

Ölçüm Sonuçları ve Tutarsızlıklar



Noktalar: örnek ölçüm · kesikli: $c \cdot n^2$ eğrisi

- 600 öğeye kadar hepsi **0 ms**?
- 3000 öğe 110 ms, 3100 ise 93 ms?
- 3500, 3600, 3700 hepsi 125 ms?

Neden? Sıralama iki saat vuruşu arasında bitince `stop - start = 0`. Vuruş aralığı ~15,6 ms olan bir sistemde sonuçlar 15–16 ms'nin katlarına yuvarlanır.

Çözüm: Her ölçüm çok kez tekrarlanır, ortalaması alınır.

Uygulama 2: Doğrusal Arama (Bekçi ile)

```
int seqsearch(int list[], int searchnum, int n) {
  int i;
  list[n] = searchnum;          /* bekci */
  for (i = 0; list[i] != searchnum; i++)
    ;
  return (i < n) ? i : -1;
}
```

```
start = clock();
for (j = 0; j < runtimes[i]; j++)
  pos = seqsearch(list, -1, sizelist[i]);
stop = clock();
total = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
ort = 1000.0 * total / runtimes[i];  /* mikro sn */
list[sizelist[i]] = sizelist[i];  /* bekciyi sil */
```

- **Bekçi** (sentinel): aranan değer `list[n]`'e konur; döngü hep durur, her turda `i < n` testine gerek kalmaz.
- Dizide **n + 1** yer olmalı (MAX_SIZE = 5001).
- −1 aranır: hiç bulunmaz → **en kötü durum** ölçülür.
- Her n için arama `runtimes[i]` kez (500000 … 5000) tekrarlanır.

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



Dizide 3 öğe var, ama **list[3] için yer ayrılmış**. 5 aranıyor.

Adım 1 / 7

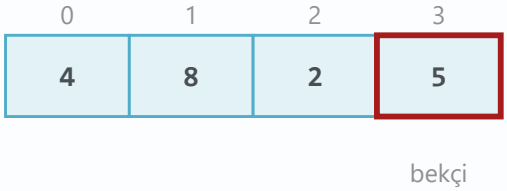
```
▶ int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, ?]`

`searchnum = 5` `n = 3`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



Bekçi: **list[3] = 5**. Artık döngü en geç burada duracak.

Adım 2 / 7

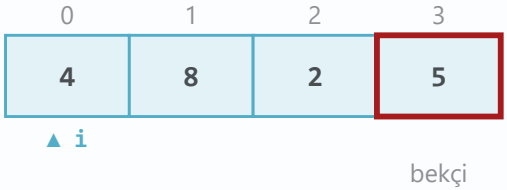
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
▶   list[n] = searchnum;          /* bekci */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



list[0] = 4 ≠ 5 → i++. (Tek test: sınır denetimi yok.)

Adım 3 / 7

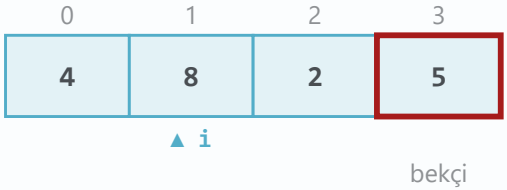
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 0`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



list[1] = 8 ≠ 5 → i++. (Tek test: sınır denetimi yok.)

Adım 4 / 7

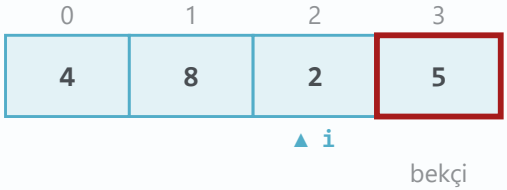
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 1`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



list[2] = 2 ≠ 5 → i++. (Tek test: sınır denetimi yok.)

Adım 5 / 7

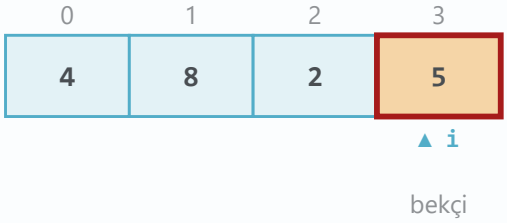
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 2`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



list[3] = 5 → döngü durdu, ama bu **bekçi**.

Adım 6 / 7

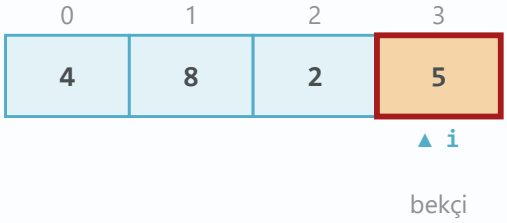
```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
▶   for (i = 0; list[i] != searchnum; i++)
5     ;
6   return (i < n) ? i : -1;
7 }
```

DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 3`

Kod İzleme: `seqsearch({4, 8, 2}, 5, 3)`



i = 3 < n değil → **return −1**: bulunamadı. n + 1 = 4 karşılaştırma.

Adım 7 / 7

```
1 int seqsearch(int list[], int searchnum, int n) {
2   int i;
3   list[n] = searchnum;          /* bekci */
4   for (i = 0; list[i] != searchnum; i++)
5     ;
▶   return (i < n) ? i : -1;
7 }
```

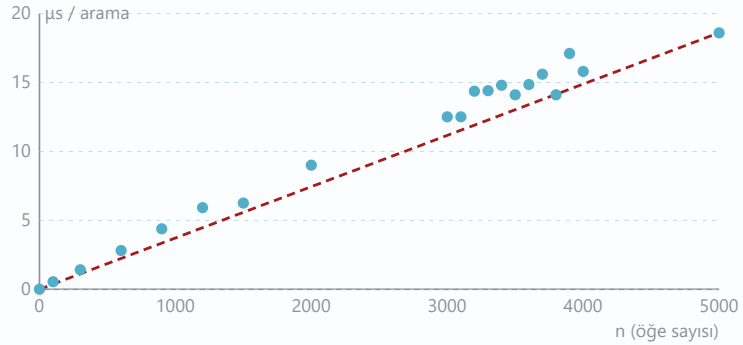
DEĞİŞKENLER

`list = [4, 8, 2, 5]`

`searchnum = 5` `n = 3` `i = 3`

PERFORMANS ÖLÇÜMÜ

Tekrarlı Ölçüm: Ortalama Arama Süresi



Noktalar: ölçülen ortalamalar · kesikli: doğrusal eğilim

- Ortalamalar artık 0 değil ve n ile **doğrusal** büyüyor: O(n) beklentisiyle uyumlu.
- 3000'den sonraki dalgalanma: tekrar sayısı (runtimes) azaldıkça ölçüm yine kabalaşıyor.
- Ölçümün yapıldığı makinede saniyede 1000 vuruş: 1 vuruş = 1 ms.

Kısa işler **çok kez** tekrarlanır, toplam süre tekrar sayısına bölünür.

ÖDEVLER

Ödev Soruları (Özyineleme)

1. n! değerini bulan fonksiyonu **özyinelemeli** olarak yazınız.
2. Fibonacci serisi $f_0 = 0$, $f_1 = 1$, $f_i = f_{i-1} + f_{i-2}$ $(i > 1)$ ise f_i'yi bulan özyinelemeli yordamı yazınız.
3. $C(n, m) = C(n-1, m) + C(n-1, m-1)$ verildiğine göre $C(n, m) = n! / (m! \cdot (n-m)!)$ değerini hesaplayan özyinelemeli yordamı yazınız.

▶ **İpuçları**

ÖDEVLER

Ödev Cevapları ve Sık Hatalar

```
long fakt(long n) {
  if (n <= 1) return 1;
  return n * fakt(n - 1);
}

long fibo(long n) {
  if (n <= 1) return n;   /* f0 = 0, f1 = 1 */
  return fibo(n - 1) + fibo(n - 2);
}

long komb(int n, int m) {
  if (m == 0 || m == n) return 1;
  return komb(n - 1, m) + komb(n - 1, m - 1);
}
```

fibo: * değil +

Sık yapılan hata: `fibo(deger-1) * fibo(deger-2)` yazmak; çarpım her zaman 0 ya da 1 verir. Tanım gereği **toplam** olmalı.

komb: parametreler

Sık yapılan hata: `long komb()` parametresiz yazmak; `n` ve `m` parametre olmalı.

Maliyet

fakt O(n); fibo ve komb bu hâliyle **üstel**: aynı alt problemler tekrar tekrar çözülür.

ÖZET

Karmaşıklık Özeti

Yordam	Zaman	Ek bellek	Not
`sort` (seçimli sıralama)	$\Theta(n^2)$	$O(1)$	$n(n-1)/2$ karşılaştırma, $\le n-1$ takas
`sum`	$\Theta(n)$	$O(1)$	$2n + 3$ adım
`rsum`	$\Theta(n)$	$O(n)$	$2n + 2$ adım, $n+1$ çerçeve
`add` (matris toplama)	$\Theta(\text{rows}\cdot\text{cols})$	$O(1)$	kare matriste $\Theta(n^2)$
`seqsearch`	$O(n)$	$O(1)$	en iyi $\Theta(1)$, en kötü $n + 1$
`sihirli` (Coxeter)	$\Theta(n^2)$	$O(1)$	kaba kuvvet: $(n^2)!$
Dinamik dizi ekleme	$O(1)$ amortize	$O(n)$	tek ekleme en kötü $O(n)$
`fibo` (özyinelemeli)	üstel	$O(n)$	yığın derinliği n

ÖZET

Sık Yapılan Hatalar

1. “Derlendi ve çalıştı, o hâlde doğru” sanmak: test verisi tüm senaryoları kapsamalı.
2. Adım sayısını sabitlerle karıştırmak: 2n + 3 ile 100n aynı sınıftadır, O(n).
3. $O(n^2)$ bulup “en iyi sınır” sanmak: mümkün olan en küçük g(n) yazılır; sıkı sınır için Θ.
4. Kısa işi tek sefer `clock()` ile ölçmek: sonuç 0 ya da vuruş katı çıkar.
5. Ölçülen bölgede `printf` bırakmak: G/Ç süresi algoritmayı gölgeler.
6. Makroya yan etkili argüman vermek: `SWAP(a[i++], …)`.

ÖZET

Alıştırmalar

1. `for (i = 0; i < n; i++) for (j = 0; j < i; j++) x++;` parçasında `x++` kaç kez işletilir? O(?)
2. $10n^2 + 100n + 1000 = O(n^2)$ olduğunu c ve n_0 vererek gösteriniz.
3. Seçimli sıralama zaten sıralı bir dizide kaç karşılaştırma yapar? En iyi durumu var mı?
4. O(n log n) bir algoritma n = 1000'de 10 ms sürüyorsa n = 10^6'da yaklaşık ne kadar sürer?
5. `rsum` 'u n = 10^6 ile çağırmak neden tehlikelidir?
6. 7×7 sihirli karede ilk satırı Coxeter kuralıyla yazınız (toplam kaç olmalı?).

▶ **Cevapları / ipuçlarını göster**

ÖZET

Bölüm Özeti

Süreç

Gereksinim → çözümleme → tasarım → kodlama → doğrulama. Gerçeklemeyi ertelemek daha iyi seçim sağlar.

Soyutlama

ADT = belirtim ⊥ gerçekleme. C'de: başlık dosyası + opak `struct`.

Sayma

Bellek: $c + S_P(I)$. Zaman: program adımları (sum $2n+3$, rsum $2n+2$).

Sınıflama

O üst, Ω alt, Θ sıkı sınır. En kötü durum garantidir; amortize analiz dizinin toplamına bakar.

Büyüme

Küçük n yanıltır; 2^n ve $n!$ kısa sürede umutsuzlaşır. Algoritma donanımdan önce gelir.

Ölçüm

`clock()` + tekrar + ortalama; G/Ç ölçüm dışında.

ÖZET

Kaynaklar

- Horowitz, Sahni, Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 1.
- Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 3. baskı: Bölüm 2–3 (analiz, asimptotik), Bölüm 17 (amortize).
- Kernighan, Ritchie, *The C Programming Language*, 2. baskı.

Ders sayfası

Bölüm 2: Diziler ve Yapılar →

Son

Bölüm 1: Giriş ve Algoritma Analizi

Sonraki bölüm: Bölüm 2: Diziler ve Yapılar · Ders sayfası