YZM 2116- VERİ YAPILARI

Ders#1: Algoritma Analizi

İÇERİK

Bu bölümde,

- Algoritma Analizi,
- Çalışma Zamanı Analizi
- Algoritma Analiz Türleri
- Asimptotik Notasyon,
- Big-O Notasyonu,
- Algoritmalar için "Rate of Growth" (Büyüme Hızı)
- Big-O Hesaplama Kuralları
- Big-O Avantajları

konularına değinilecektir.

Algoritma Nedir?

19.yy da İranlı Musaoğlu Horzumlu Mehmet (Alharezmi adını Araplar takmıştır) ***problemlerin çözümü*** *için* ***genel kurallar*** *oluşturdu. Algoritma Alharezmi'nin Latince okunuşudur.*

- Basit tanım: Belirli bir görevi yerine getiren sonlu sayıdaki işlemler dizisidir.
- Geniş tanım: Verilen herhangi bir sorunun çözümüne ulaşmak için uygulanması gerekli adımların hiç bir yoruma yer vermeksizin açık, düzenli ve sıralı bir şekilde **söz ve yazı** ile ifadesidir. Algoritmayı oluşturan adımlar özellikle **basit** ve **açık** olarak ortaya konmalıdır.

Algoritmaların Sahip Olması Gereken Özellikler

- Giriş/çıkış bilgisi
- Sonluluk
- Kesinlik
- Etkinlik
- Başarım ve performans

Algoritma Analizi

- **Aynı problemi** (örneğin sıralama) birçok algoritma ile (**insertion**, **bubble**, **quick** vs) çözmek mümkün olduğu için algoritmalar ***verimlilik*** (kullandıkları hafıza ve işlemi gerçekleştirdikleri zaman) anlamında kıyaslanmalı ve **seçim** buna göre *yapılmalıdır*.
- Bu kıyaslama algoritma analizinde ***çalışma zamanı karşılaştırması*** olarak bilinir.

Algoritma Analizi (devam…)

- Algoritma analizi yapılma nedenleri:
  - Algoritmanın performansını ölçmek için
  - Farklı algoritmalarla karşılaştırmak için
  - Daha iyisi mümkün mü? Bu mudur **en iyisi**?
- Analiz edilen özellikler aşağıdaki gibidir
  - Algoritmanın **çalışma zamanı analizi**
  - Hafızada **kapladığı alan analizi**

Çalışma Zamanı Analizi

- *Çalışma zamanı analizi* (***karmaşıklık analizi***) *bir algoritmanın* (***artan***) *"**(veri) giriş**" boyutuna bağlı olarak işlem zamanının / süresinin **nasıl arttığını** (değiştiğini) tespit etmek olarak tanımlanır.*
- Algoritmaların işlediği sıklıkla karşılaşılan *"**(veri) giriş**"* türleri:
  - Array (boyuta bağlı)
  - Polinom (derecesine bağlı)
  - Matris (eleman sayısına bağlı)
  - İkilik veri (bit sayısı)
  - Grafik (kenar ve düğüm sayısı)

Çalışma Zamanı Analizi (devam…)

- Çalışma zamanı/karmaşıklık analizi için kullanılan **<u>başlıca yöntemler</u>** aşağıdaki gibidir:
  1. **<u>Deneysel Analiz Yöntemi:</u>** Örnek problemlerde *denenmiş bir algoritmadaki* hesaplama <u>deneyimine</u> dayanır. Amacı, ***pratikte*** *algoritmanın nasıl davrandığını tahmin etmektir*. Bilimsel yaklaşımdan çok, *<u>uygulamaya yöneliktir</u>*. Programı yazan programcının **tekniğine**, **kullanılan bilgisayara**, **derleyiciye** ve **programlama diline** bağlı ***değişkenlik*** gösterir.
  2. **<u>RAM (Random Access Machine) Modeli ile Komut Sayarak Çalışma Zamanı Analiz Yöntemi</u>**

Çalışma Zamanı Analizi (devam…)

- **RAM Modeli:**
  - RAM modeli algoritma gerçekleştirimlerini ölçmek için kullanılan **basit bir yöntemdir**.
  - Genel olarak çalışma zamanı veri giriş boyutu **n'e** *bağlı* **T(n)** ile *ifade edilir*.
  - Her operasyon (+,-, * =, if, call) "bir" zaman biriminde gerçekleşir.
  - Döngüler ve alt rutinler (fonksiyonlar) basit operasyonlar ile farklı değerlendirilirler.
  - RAM modelinde her bellek erişimi yine **"bir"** zaman biriminde gerçekleşir.

Örnek 1: Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
  int toplam = 0;

  for  (i=0; i < N; i++){
    toplam += A[i];
  } //Bitti-for

  return toplam;
} //Bitti-Topla
```

Bu **fonksiyonun** **yürütme zamanı** ne kadardır?

Örnek 1: Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
   int topla = 0;

   for (i=0; i < N; i++){
      topla += A[i];
   } //Bitti-for

   return topla;
} //Bitti-Topla
```

İşlem sayısı

1

N

N

1

Toplam: $1 + N + N + 1 = 2N + 2$

- Çalışma zamanı: $T(N) = 2N+2$
  - N dizideki eleman sayısı

Örnek 2: Dizideki bir elemanın aranması

```
int Arama(int A[], int N,
                   int sayi) {
  int i = 0;

  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

Bu **fonksiyonun** **yürütme zamanı** ne kadardır?

Örnek 2: Dizideki bir elemanın aranması

```
int Arama(int A[], int N,
                 int sayi) {
  int i = 0;

  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

İşlem sayısı

1

1<=L<=N

1<=L<=N

0<=L<=N

1

1

Toplam: 1+3*L+1+1 = 3L+3

- Çalışma zamanı: T(N) = 3N+3

Algoritma Analiz Türleri

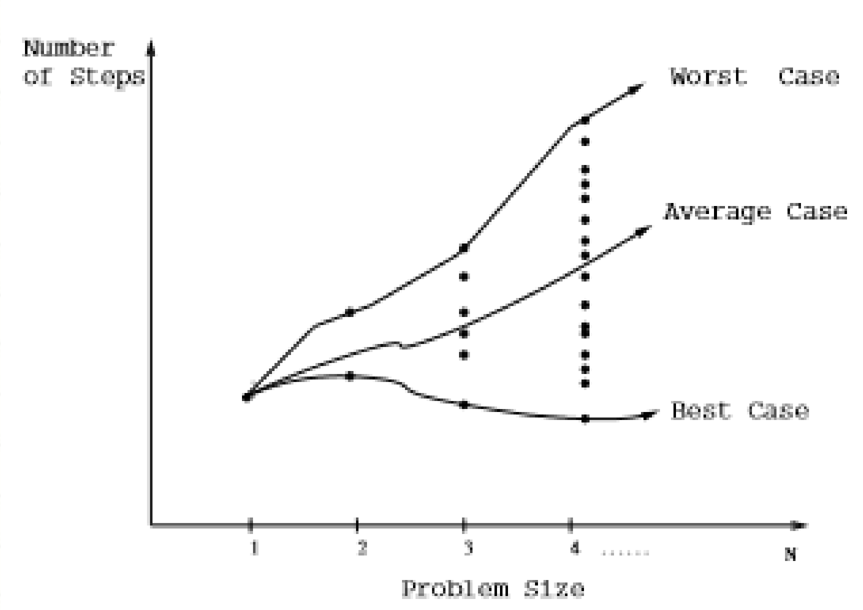
- Bir algoritmanın analizi için o algoritmanın kabaca bir polinom veya diğer zaman karmaşıklıkları cinsinden *ifade edilmesi* gerekir.
- Bu ifade üzerinden veri girişindeki değişime bağlı olarak algoritmanın *best case (en az zaman alan)* ve *worst case (en çok zaman alan)* durumları incelenerek algoritmalar arası kıyaslama yapılabilir. Bu şekilde bir algoritma üç şekilde incelenebilir:
  1. **Worst case (en kötü)**
  2. **Best case (en iyi)**
  3. **Average case (ortalama)**

Algoritma Analiz Türleri (devam…)

- **Worst case (en kötü):** Algoritma çalışmasının en fazla sürede gerçekleştiği analiz türüdür. En kötü durum, çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan *"daha uzun sürmeyeceği"* **garantisi** verir. Bazı algoritmalar için en kötü durum ***oldukça sık rastlanır***. Arama algoritmasında, *aranan öğe* genellikle **dizide olmaz** dolayısıyla **döngü N kez çalışır**.
- **Best case (en iyi):** Algoritmanın en kısa sürede ve en az adımda çalıştığı giriş durumu olan analiz türüdür. Çalışma zamanında bir alt sınırdır.
- **Average case (ortalama):** Algoritmanın ortalama sürede ve ortalama adımda *çalıştığı* giriş durumu olan analiz türüdür.

Algoritma Analiz Türleri (devam…)

- Bu incelemeler:
- ***Lower Bound (i) <= Average Bound (ii) <= Upper Bound (iii)*** şeklinde sıralanırlar.
- Grafiksel gösterimi aşağıdaki gibidir:



Algoritma Analiz Türleri (devam…)

- **Örnek 2** için en iyi, en kötü ve ortalama çalışma zamanı nedir?

```
int Arama(int A[], int N,
                 int sayi) {
  int i = 0;

  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

- En iyi çalışma zamanı
  - Döngü sadece bir kez çalıştı
    T(n) = 6
- Ortalama çalışma zamanı
  - Döngü N/2 kez çalıştı
    T(n)=3*n/2+3 = 1.5n+3
- En kötü çalışma zamanı
  - Döngü N kez çalıştı
    T(n) = 3n+3

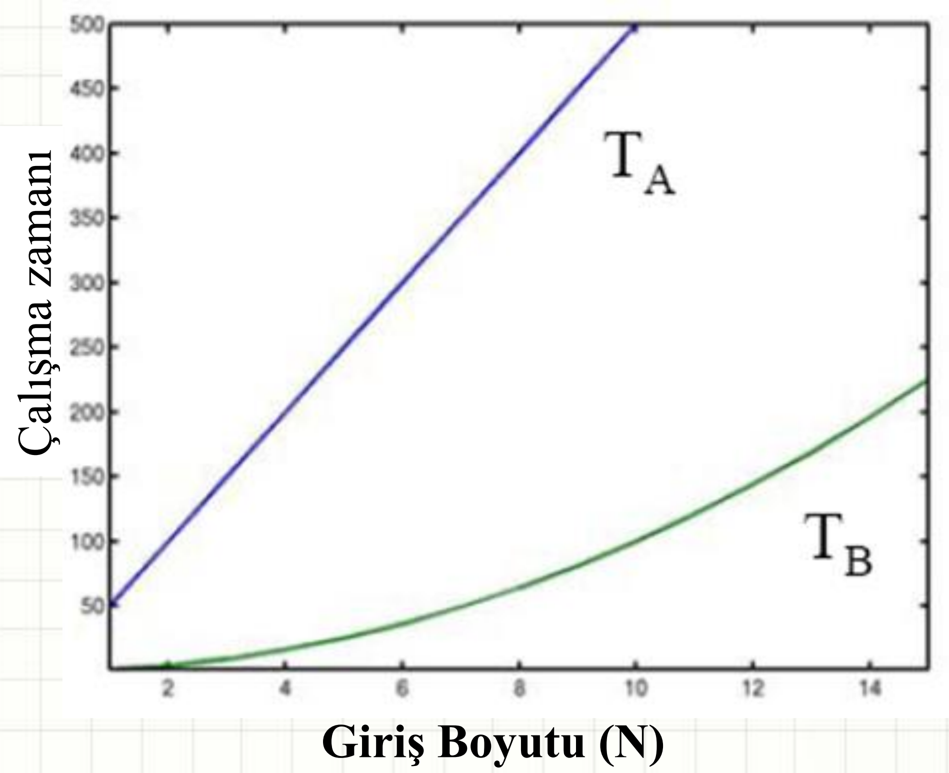
Algoritma En Kötü Durum Analizi

- Bir algoritmanın **genelde** EN KÖTÜ durumdaki çalışma zamanına bakılır. **Neden?**
  - En kötü durum çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan daha **uzun sürmeyeceği garantisi** verir.
  - Bazı algoritmalar için *en kötü durum oldukça sık rastlanır*. Arama algoritmasında, aranan öğe genellikle dizide olmaz **dolayısıyla döngü N kez çalışır**.
  - Ortalama çalışma zamanı genellikle en kötü çalışma zamanı kadardır. Arama algoritması için **hem** ortalama hem de en *kötü çalışma zamanı* doğrusal fonksiyondur.

Asimptotik Notasyon

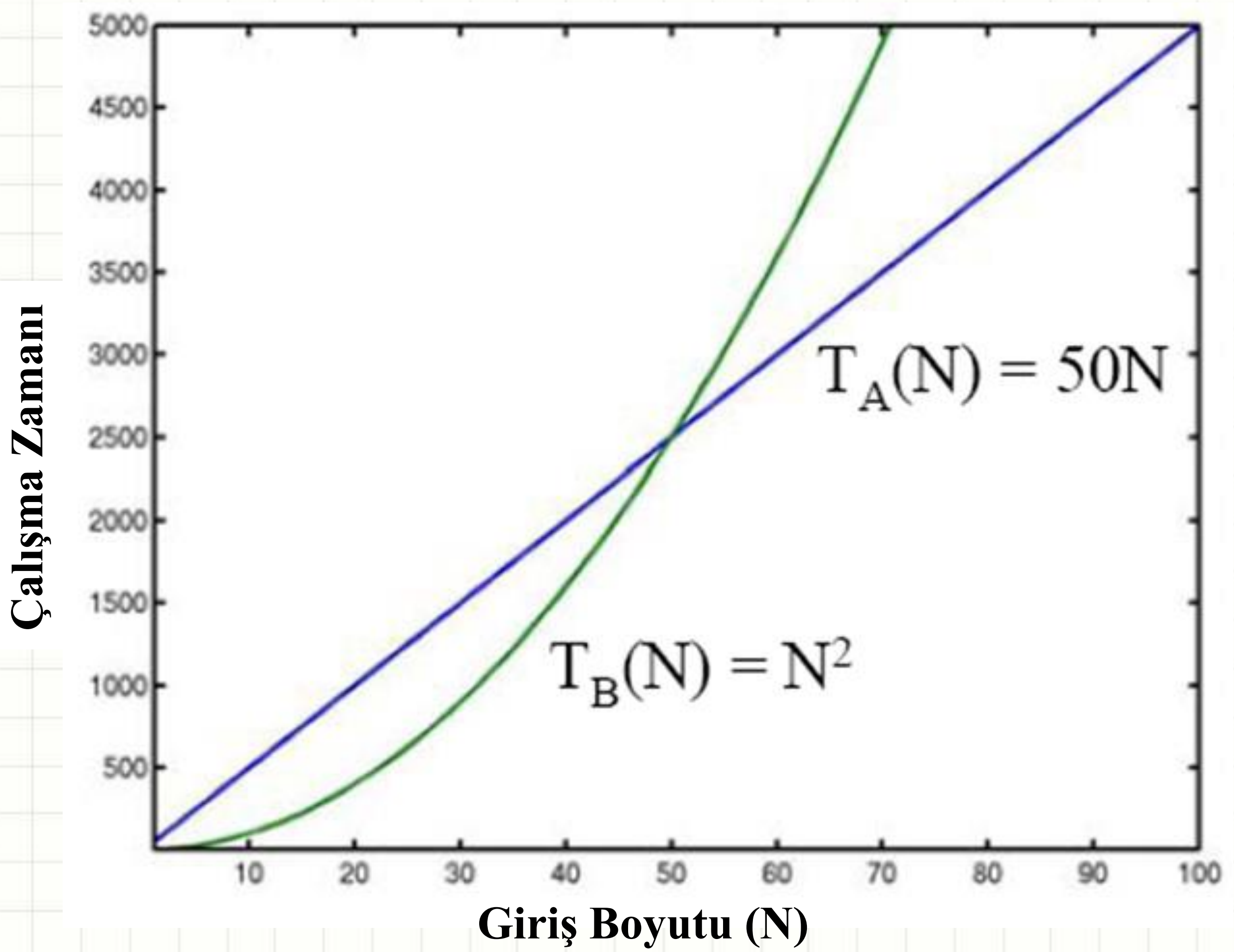
- Bir problemi çözmek için **A ve B** şeklinde iki algoritma verildiğini düşünelim.
- Giriş boyutu N için aşağıda A ve B algoritmalarının çalışma zamanı **T_A ve T_B fonksiyonları** verilmiştir.

Hangi algoritmayı seçersiniz?



Asimptotik Notasyon (devam…)

- N büyüdüğü zaman A ve B nin çalışma zamanı:



Şimdi hangi algoritmayı seçersiniz?

Asimptotik Notasyon (devam…)

- Asimptotik notasyon, **eleman sayısı n'nin sonsuza gitmesi durumunda** algoritmanın, benzer işi yapan algoritmalarla karşılaştırmak için kullanılır.
- Eleman sayısının *küçük olduğu durumlar* mümkün olabilir fakat bu birçok uygulama için geçerli değildir.
- Verilen iki algoritmanın çalışma zamanını T1(N) ve T2(N) fonksiyonları şeklinde gösterilir. Hangisinin **daha iyi** olduğunu belirlemek için bir *yol belirlememiz* gerekiyor.
  - Big-O (Big O): Asimptotik üst sınır
  - Big Ω (Big Omega): Asimptotik alt sınır
  - Big Θ (Big Teta): Asimptotik alt ve üst sınır

Big-O Notasyonu

- Algoritmanın *f(n)* şeklinde **ifade edildiğini varsayalım**.
- Algoritma, fonksiyonunun sıkı üst sınırı (tight upper bound) olarak tanımlanır.
- Bir fonksiyonun <u>sıkı üst sınırı</u> genel olarak:

  $f(n) = O(g(n))$
- şeklinde ifade edilir.
- Bu ifade <u>n'nin artan değerlerinde</u>
  - *f(n)*'nin *üst sınırı g(n)*'dir
- şeklinde yorumlanır.

Big-O Notasyonu (devam…)

- **Örneğin:**

  $f(n) = n^4 + 100n^2 + 10n + 50$ algoritma fonksiyonunda

  $g(n)= n^4$ olur.

- “Daha açık bir ifadeyle”, n’nin artan değerlerinde f(n) nin **maksimum büyüme oranı**

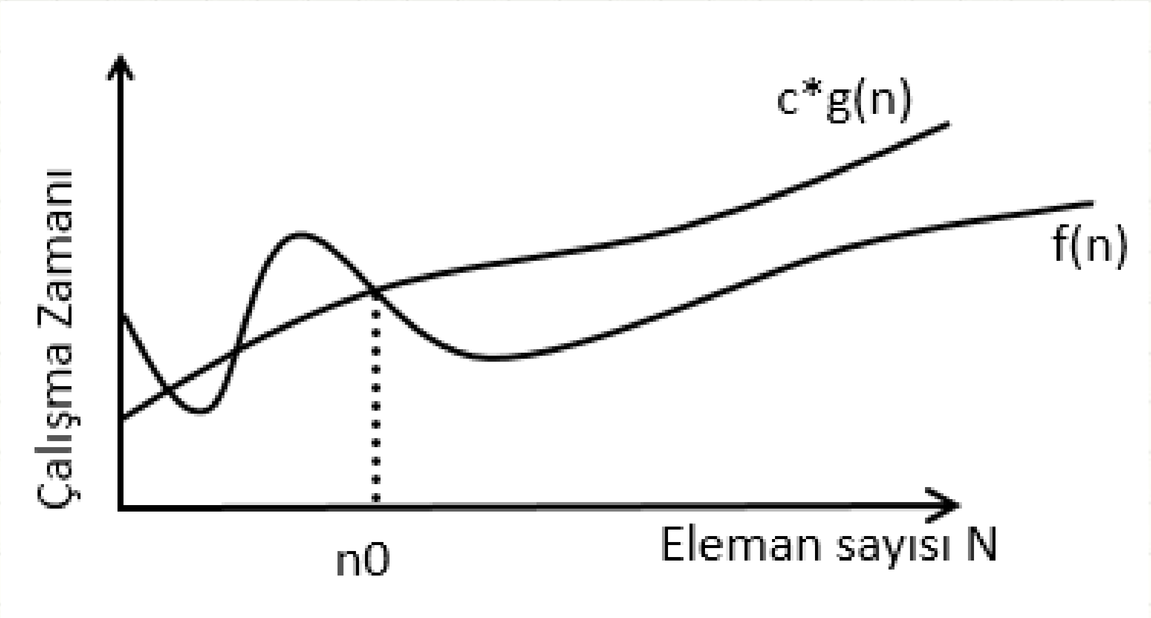
  $g(n) = n^4$

- O-notasyonu gösteriminde bir fonksiyonun *düşük n değerlerindeki* performansı **önemsiz kabul edilir**.

Big-O Notasyonu (devam…)

- O(g(n)) = {
  - f(n): tüm $n>=n_0$ için, $0 <= f(n)<= cg(n)$ olmak üzere pozitif c ve n_0 sabitleri bulunsun
- }
- Bu durumda g(n), f(n)'nin asimptotik (n sonsuza giderken) **sıkı üst sınırı** olur.
- n'nin düşük değerleri ve o değerlerdeki değişim dikkate alınmazken, n_o'dan büyük değerler için *algoritmanın büyüme oranı* **değerlendirilir**.

Big-O Notasyonu (devam…)

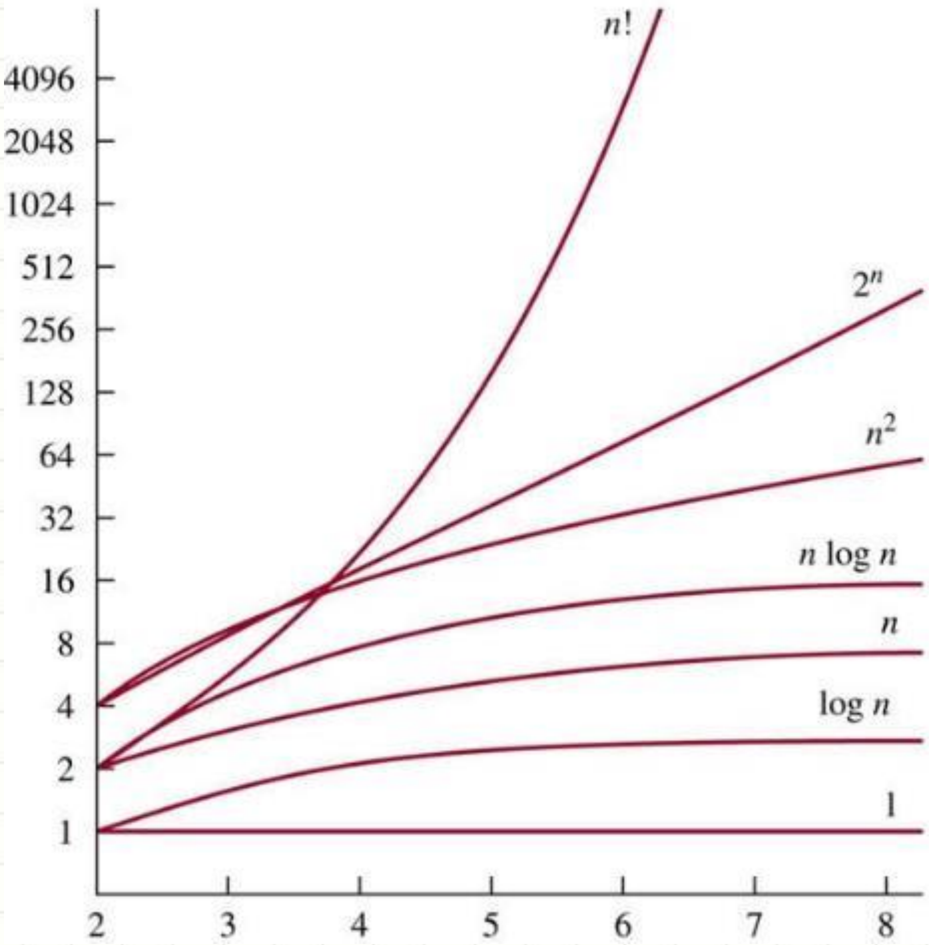


- Dikkat edilirse, n_o'dan büyük değerler için **c*g(n)**,
- f(n) için üst sınırı (asimptot) olarak görülürken,
- **n_o öncesinde** iki fonksiyonun değişimi farklı olabilir.

Büyüme Oranı (Rate of Growth)

Zaman karmaşıklığı	Açıklama	Örnek
O(1)	Sabit: Veri giriş boyutundan bağımsız gerçekleşen işlemler.	Bağlı listeye ilk eleman olarak ekleme yapma
O(log N)	Logaritmik: Problemi küçük veri parçalarına bölen algoritmalarda görülür.	Binary search tree veri yapısı üzerinde arama
O(N)	Lineer – doğrusal: Veri giriş boyutuna bağlı doğrusal artan.	Sıralı olmayan bir dizide bir eleman arama
O(N log N)	Doğrusal çarpanlı logaritmik: Problemi küçük veri parçalarına bölen ve daha sonra bu parçalar üzerinde işlem yapan.	N elemanı böl-parçala-yönet yöntemiyle sıralama. Quick Sort.
$O(N^2)$	Karesel	Bir grafikte iki düğüm arasındaki en kısa yolu bulma veya Buble Sort.
$O(N^3)$	Kübik	Ardarda gerçekleştirilen lineer denklemler
$O(2^N)$	İki tabanında üssel	Hanoi’nin Kuleleri problemi

Büyüme Oranı (Rate of Growth) (devam…)



Big-O Analiz Kuralları

- *f(n), g(n), h(n),* ve *p(n)* pozitif tamsayılar kümesinden, pozitif reel sayılar kümesine tanımlanmış fonksiyonlar olsun:

1. ***Katsayı Kuralı:*** *f(n) , O(g(n))* ise o zaman *kf(n)* yine *O(g(n))* olur. Katsayılar önemsizdir.
2. ***Toplam Kuralı:*** *f(n), O(h(n))* ise ve *g(n), O(p(n))* verilmişse *f(n)+g(n), O(h(n)+p(n))* olur. Üst-sınırlar toplanır.
3. ***Çarpım Kuralı:*** *f(n), O(h(n))* ve *g(n), O(p(n))* için *f(n)g(n)* is *O(h(n)p(n))* olur.
4. ***Polinom Kuralı:*** *f(n), k* dereceli polinom ise *f(n)* için $O(n^k)$ kabul edilir.
5. ***Kuvvetin Log'u Kuralı:*** $log(n^k)$ için *O(log(n))* dir.

Big-O Hesaplama Kuralları

- Programların ve algoritmaların Big-O yaklaşımıyla analizi için aşağıdaki kurallardan faydalanırız:
  1. **Döngüler (Loops)**
  2. **İç içe Döngüler**
  3. **Ardışık deyimler**
  4. **If-then-else deyimleri**
  5. **Logaritmik karmaşıklık**

Kural 1: Döngüler (Loops)

Bir döngünün çalışma zamanı, en çok döngü içindeki deyimlerin çalışma zamanının **iterasyon sayısıyla çarpılması** kadardır.

n defa çalışır

```
for (i=1; i<=n; i++)
{
    m = m + 2;
}
```

← Sabit zaman

Toplam zaman = sabit c * n = cn = **O(*N*)**

DİKKAT: Eğer bir döngünün n değeri sabit verilmişse.
Örneğin: n = 100 ise değeri O(1)'dir.

Kural 2: İç İçe Döngüler

İçteki analiz yapılır. Toplam zaman bütün döngülerin çalışma sayılarının çarpımına eşittir.

Dış döngü *n* defa çalışır

```
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        k = k+1;
    }
}
```

iç döngü *n* defa çalışır

Sabit zaman

Toplam zaman = c * n * n * = cn^2 = $\mathbf{O(\boldsymbol{N}^2)}$

Kural 3: Ardışık Deyimler

Her deyimin zamanı birbirine eklenir.

Sabit zaman → `x = x +1;`

Sabit zaman → `m = m + 2;`

```
x = x +1;
for (i=1; i<=n; i++) {
     m = m + 2;
}
for (i=1; i<=n; i++)  {
    for (j=1; j<=n; j++) {
        k = k+1;
    }
}
```

n defa çalışır

Dış döngü *n* defa çalışır

iç döngü *n* defa çalışır

Sabit zaman

Toplam zaman = $c_0 + c_1 n + c_2 n^2 = \mathbf{O(}\boldsymbol{N}^{\mathbf{2}}\mathbf{)}$

Kural 4: If Then Else Deyimleri

En kötü çalışma zamanı: **test zamanına** *then veya else kısmındaki çalışma zamanının* **hangisi büyükse** o kısım eklenir.

test:
sabit

```
if (depth( ) != otherStack.depth( ) ) {
   return false;
}
else {
   for (int n = 0; n < depth( ); n++) {
   if (!list[n].equals(otherStack.list[n]))
      return false;
   }
}
```

then:
sabit

Diğer if :
sabit+sabit
(else yok)

else:
(sabit +sabit) * n

Toplam zaman = $c_0 + c_1 + (c_2 + c_3) * n$ = **O(*N*)**

Kural 5: Logaritmik Karmaşıklık

Problemin **büyüklüğünü belli oranda(genelde ½) azaltmak** için **sabit bir zaman harcanıyorsa** bu algoritma O(log *N*)'dir.

for(i=1; i<=n;)
*i = i*2;*

- kod parçasında **n döngü sayısı** i = i*2 den dolayı her seferinde yarıya düşer.
- Loop'un k kadar döndüğünü varsayarsak;
  - k adımında $2^i = n$ olur.
  - Her iki tarafın logaritmasını alırsak;
    - ilog2 = logn ve i = log n olur.
  - i'ye bağlı olarak (problemi ikiye bölen değişken!)

Kural 5: Logaritmik Karmaşıklık (devam…)

- Örneğin: **Binary Search (İkili arama)** algoritması kullanılarak bir sözlükte arama:
  - Sözlüğün orta kısmına bakılır.
  - Sözcük ortaya göre sağda mı solda mı kaldığı bulunur?
  - Bu işlem sağ veya solda sözcük bulunana kadar tekrarlanır.

- bu tarz bir algoritmadır. Bu algoritmalar genel olarak "divide and conquer (böl ve yönet)" yaklaşımı ile tasarlanmışlardır. Bu yaklaşımla tasarlanan olan örnek sıralama algoritmaları:
  - **Merge Sort** and **Quick Sort.**

Big-O Avantajları

- Sabitler göz ardı edilirler çünkü
  - Donanım, derleyici, kod optimizasyonu vb. nedenlerden dolayı bir komutun **çalışma süresi** her zaman *farklılık* gösterebilir. Amacımız bu etkenlerden bağımsız olarak algoritmanın ne kadar etkin olduğunu ölçmektir.
  - Sabitlerin atılması analizi **basitleştirir**. $3.2n^2$ veya $3.9n^2$ yerine sadece n^2'ye odaklanırız.
- Algoritmalar arasında kıyaslamayı basit tek bir değere **indirger**.
- Küçük n değerleri göz ardı edilerek sadece büyük n değerlerine *odaklanılır*.

Big-O Avantajları (devam…)

- ***Özetle:*** Donanım, işletim sistemi, derleyici ve algoritma detaylarından bağımsız, sadece büyük n değerlerine odaklanıp, sabitleri göz ardı ederek daha basit bir şekilde algoritmaları analiz etmemize ve karşılaştırmamızı sağlar.

- RAM'den O(n)' dönüşüm:
  - $4n^2 - 3n \log n + 17.5\, n - 43\, n^{2/3} + 75 \rightarrow$
  - $n^2 - n \log n + n - n^{2/3} + 1 \rightarrow$ sabitleri atalım
  - $n^2 \rightarrow$ sadece büyük n değerlerini alalım
  - **$O(n^2)$**

Çalışma

- Aşağıdaki fonksiyonların karmaşıklıklarını Big O notasyonunda gösteriniz.
  - $f1(n) = 10\ n + 25\ n^2$
  - $f2(n) = 20\ n \log n + 5\ n$
  - $f3(n) = 12\ n \log n + 0.05\ n^2$
  - $f4(n) = n^{1/2} + 3\ n \log n$