

# Algoritmaların Özellikleri

- Bir algoritmanın taşınması gereken beş tane temel özelliği vardır.

- **1. Giriş (Input)**

- Bir algoritmanın sıfır veya daha fazla giriş değişkeni vardır. Giriş değişkenleri algoritma işlemeye başlamadan önce, algoritmaya verilen değerler kümesidir veya değer kaydetmesi için verilen hafıza bölgesidir.

- **2. Belirlilik (Definiteness)**

- Bir algoritmanın her adımı için kesin olarak ne iş yapacağı belirlenmelidir ve belirsizlik olmamalıdır. Her durum için hangi işlem gerçekleştirilecekse, o açık olarak tanımlanmalıdır.

- **3. Çıkış (Output)**

- Her algoritmanın bir veya daha fazla çıkış değeri vardır. Çıkış değerleri ile giriş değerleri arasında bağıntılar vardır.

- **4. Etkililik (Efficiency)**

- Olabildiğince hızlı çalışmalıdır, olabildiğince az hafıza kullanılmalıdır. Bunun anlamı yapılan işlemler yeterince temel işlemler olacak ve sınırlı zaman süresince işleyip bitmelidir.

- **5. Sınırlılık (Boundedness)**

- Her algoritma sınırlı sayıda çalışma adımı sonunda bitmelidir. Bir algoritma için sınırlılık çok önemlidir. Aynı işlemi yapan iki algoritmadan biri bir milyar adımda bitiyor olsun ve diğeri de yüz adımda bitiyor olsun. Bu durumda yüz adımda biten algoritma her zaman daha iyidir. Bunun anlamı sınırlılık kavramı ile anlatılmak istenen mümkün olan en az sayıda adım ile işlemin bitirilmesidir.

# Algoritmaların Özellikleri

- Diğer bazı kriterler ise **algoritmanın bilgisayar ortamına aktarılabilme özelliği, basitliği**, vb. gibi özelliklerdir.
- Bir problem için birden fazla algoritma verilmişse, bu algoritmalardan en iyisinin seçilmesi gerekir.
- İyi algoritmayı belirlemek için uygulanan testler veya yapılan işlemler **Algoritma Analizi'** nin konusudur.

# Algoritmaların Yönleri

## ○ 1- Algoritmaları Tasarlama

- Bulmacaların (puzzel) parçalarını birleştirme,
- Veri yapılarını seçme,
- Problemin çözümü için temel yaklaşımlar seçme.
- En popüler tasarım stratejileri böl ve fethet (divide&conquer), açgözlü(greedy), dinamik programlama, özyineleme (recursive).

## ○ 2- Algoritma ifadesi ve uygulanması

- Algoritmayı tasarladıktan sonra sözde kod (pseudocode) ifadesinin belirlenmesi ve problem için uygulanması
- Bu konudaki endişeler, netlik, özlülük, etkinlik vb.

# Algoritmaların Yönleri

## ○ 3-Algoritma Analizi (Çözümlemesi)

- Algoritma analizi, algoritmayı gerçekte uygulamadan, bir algoritmayı çalıştırabilmek için gereken kaynakların (zaman, yer gibi) araştırılması demektir.

## ○ 4- Çözümünüzün yeterince iyi olup olmadığını görmek için alt ve üst sınırları karşılaştırma

- Algoritma analizi problemi çözmek için bize alt ve üst sınırları verir.

# Algoritmaların Yönleri

## ○ 5- Algoritma veya programı doğrulama

- Algoritmanın verilen tüm olası girişler için hesaplama yaptığını ve doğru çıkış ürettiğini göstermektir.

## ○ 6- Algoritmaların test edilmesi

- Test için iki aşama vardır;
- **Hata ayıklama (Debugging):** Programın örnek veriler üzerinde çalıştırılması sırasında oluşan hataları tespit etme ve onları düzeltme işlemi.
- **Profil oluşturma (Profiling):** Çeşitli veriler üzerinde programın çalıştırılması ve sonuçların hesaplaması için gerekli zamanın (ve alan) ölçülmesi işlemi.

# Algoritma Tasarımı

- Algoritmaları tasarlamada kullanılacak yöntemler
  - Özyineleme
  - Böl ve fethet
  - Bilinen probleme indirgeme
  - Dinamik programlama
  - Kaba Seçim veya Haris (Greedy) algoritması
  - Bir veri yapısı icat etme
  - İhtimali (olasılıksal) çözümler
  - Yaklaşım çözümleri

## Algoritmaları tasarlamada kullanılacak yöntemler:

### ○ Özyineleme

- Problemin çözümünün tekrarlı olması durumunda bilinen bir veya birkaç çözümünden faydalanarak bir sonraki çözümü elde etme ve elde edilen çözüm ile önceki çözümlerin birkaçının kullanılması ile bir sonraki çözümün elde edilmesi ile problemi çözme işlemine **özyineleme yöntemi** denir.

### ○ Böl ve fethet

- Kompleks problemlerin bir bütün olarak çözümleri çok zor olacağından dolayı, bu problemler alt problemlere bölünürler. Bu bölünme işleminin yapılabilmesi için alt problemlerin bir üst seviyedeki problem ile aynı özelliği sağlamalıdır. Bu yöntem ile algoritma tasarımı yapmaya **böl ve fethet yöntemi** denir.

### ○ Dinamik programlama

- Böl ve yönet yöntemine benzer olarak alt problemlerin çözümlerini birleştirerek çözüme gitme mantığına sahip olup alt problem tekrarı varsa, bunlardan bir tanesi çözülür ve bu çözüm diğer tekrarlarda kullanılır. Bu yöntem ile yapılan algoritma tasarım yöntemine **dinamik programlama yöntemi** denir.

### ○ Kaba Seçim veya Haris (Greedy) algoritması

- Optimizasyon problemlerinin çözümü için yerel optimumların seçilmesi ilkesinden yola çıkar ve veriyi belli bir kritere göre düzenledikten sonra ilk veri her zaman optimum çözüme götürür mantığına sahiptir. Temel amaç en iyi sonucu elde etmek için en iyi ara adım çözümlerini seçmeye yönelik bir yöntem olduğundan bu yöneme **haris algoritması yöntemi** denir.

## Algoritmaları tasarlamada kullanılacak yöntemler:

### ○ Bir veri yapısı icat etme

- O ana kadar var olmayan bir veri yapısının icat edilmesi ile problemin çözülmesine **veri yapısı icat etme yöntemi** denir.

### ○ Bilinen probleme indirgeme

- Kompleks olan bir problemin çözümünü yapmak için çözümü bilinen bir veya birden fazla başka probleme dönüştürüp bu şekilde problemi çözme işlemine **bilinen probleme indirgeme yöntemi** denir.

### ○ İhtimali (olasılıksal) çözümler

- Bazı durumlarda gelişigüzellik ilkesi ile etkili bir şekilde problem çözümü yapılabilmektedir. Bunlara örnek olarak Las Vegas polinom-zamanlı ve Monte Carlo polinom-zamanlı algoritmalar verilebilir. Gelişigüzellik kullanılarak yapılan problem çözümlerine **ihtimali çözümler yöntemi** denir.

### ○ Yaklaşım çözümleri

- Çözümü deterministik Turing makinası ile yapılamayan yani karmaşık hesaplamaların belirli bir yöntem ile çözülemediği bu problemlerin bir kısmına bazı kriterler uygulayarak yaklaşım mantığı ile çözüm üretilebilmektedir. Bundan dolayı bu mantık ile yapılan algoritma tasarımına **yaklaşım çözümler yöntemi** adı verilir.



# Algoritma Analizi

- Algoritma analizi, bilgisayar programının performansı (başarım) ve kaynak kullanımı konusunda teorik çalışmalardır.
- Bir başka ifadeyle, algoritmanın icra edilmesi sırasında duyacağı kaynak miktarının tahmin edilmesine **Algoritma Analizi** denir.
  - Kaynak denildiğinde, bellek, iletişim bant genişliği, mantık kapıları akla gelebilir, fakat en önemli kaynak algoritmanın icra edilebilmesi için gerekli olan zamanın tahmin edilmesidir.
- Algoritma analizi, farklı çözüm yöntemlerinin verimliliğini analiz eder.
- Biz bu derste performans yani başarım üzerine yoğunlaşacağız.

# Algoritma Analizi

- Performanstan daha önemli ne vardır ?
  - modülerlik
  - doğruluk
  - bakım kolaylığı
  - işlevsellik
  - sağlamlık
  - kullanıcı dostluğu
  - programcı zamanı (fiyat)
  - basitlik
  - genişletilebilirlik
  - güvenilirlik

# Neden algoritmalar ve başarımla uğraşırız?

- Başarım (performans) genelde yapılabilir olanla imkansızın arasındaki çizgiyi tanımlar.
- Algoritmik matematik program davranışlarını açıklamak için ortak dil oluşturur.
- Başarım **bilgi işleme**'nin para birimidir.
- Program başarımından alınan dersler diğer bilgi işleme kaynaklarına genellenebilir.
- Hız eğlencelidir!

# Algoritmik Performans

- Algoritmik performansın iki yönü vardır:
  - **Zaman (Time)**
    - Yönergeler veya talimatlar zaman alabilir.
    - Algoritma ne kadar hızlı bir performans gösteriyor?
    - Algoritmanın çalışma zamanını (runtime) ne etkiler?
    - Bir algoritma için gerekli olan zaman nasıl tahmin edilir?
    - Gerekli olan zaman nasıl azaltılabilir?
  - **Alan (Space)**
    - Veri yapıları yer kaplar.
    - Ne tür veri yapıları kullanılabilir?
    - Veri yapılarının seçimi çalışma zamanını nasıl etkiler?

# Algoritma Analizi

- Bir algoritmanın analizinin yapılabilmesi için matematiksel bilgilere (temel olasılık, kümeler, cebir, v.b.) ihtiyaç duyulduğu gibi bazı terimlerin formül olarak ifade edilmesi gereklidir. Çünkü her giriş için algoritmanın davranışı farklı olabilir.
- Benzer problemi çözmek için iki algoritmanın zaman verimliliğini nasıl karşılaştırabiliriz?
  - **Naif( Basit ) yaklaşım:** bir programlama dilinde bu algoritmaların uygulanması ve çalışma zamanlarının karşılaştırılması.

# Algoritma Analizi

- Algoritmalar yerine programların karşılaştırılmasında aşağıda belirtilen zorluklar vardır.
  - **Programın kullanabileceği veri nedir?**
    - Analiz yöntemi veriye bağımlı olmamalıdır. Çalışma zamanı verinin büyüklüğü ile değişebilir.
  - **Hangi bilgisayarı kullanmak gerekir?**
    - Algoritmaların verimliliği belirli bir bilgisayara bağımlı olmadan karşılaştırılmalıdır. Çünkü, aynı algoritmanın işlemci hızları farklı iki bilgisayarda çalışma zamanı aynı olmaz.
  - **Algoritma nasıl kodlanmalıdır?**
    - Çalışma zamanını karşılaştırmak, uygulamaları karşılaştırmak anlamına gelir. Uygulamalar, programlama tarzına duyarlı olduğundan karşılaştıramayız. Programlama tarzı çok verimli bir algoritmanın çalışma zamanını bile etkileyebilir.
  - Programları karşılaştırmak, bir algoritmanın kesin ölçümü için uygun değildir.

# Algoritmaların Analizi

- Algoritma analizi, özel uygulamalardan, bilgisayarlardan veya veriden bağımsızdır.
- Algoritma analizi, tasarlanan program veya fonksiyonun belirli bir işleme göre matematiksel ifadesini bulmaya dayanır.
- Algoritmaları analiz etmek;
  - İlk olarak, algoritmanın etkinliğini değerlendirmek için belirli bir çözümde anlamalı olan işlemlerin kaç adet olduğu sayılır.
  - Daha sonra büyüme fonksiyonları kullanılarak algoritmanın verimliliği ifade edilir.

# Algoritma Analizi

Problem	$n$ elemanlı giriş	Temel işlem
Bir listede arama	liste $n$ elemanlı	karşılaştırma
Bir listede sıralama	liste $n$ elemanlı	karşılaştırma
İki matrisi çarpma	$n \times n$ boyutlu iki matris	çarpma
Bir ağaçta dolaşma	$n$ düğümlü ağaç	Bir düğüme erişme
Hanoi kulesi	$n$ disk	Bir diski taşıma

- Not: Temel işlem tanımlayarak bir algoritmanın karmaşıklığını ölçebiliriz ve giriş büyüklüğü  $n$  için bu temel işlemi algoritmanın kaç kez gerçekleştirdiğini sayabiliriz.



# Algoritmaların Analizi

- Anlamalı olan işlemler hakkında önemli not:
  - Eğer problemin boyutu çok küçük ise algoritmanın verimliliğini muhtemelen ihmal edebiliriz.
  - Algoritmanın zaman ve bellek gereksinimleri arasındaki ağırlığı dengelemek zorundayız.
  - Dizi tabanlı listelerde geri alma işlemleri  **$O(1)$** 'dir. Bağlı listelerde geri alma işlemi ise  **$O(n)$** 'dir. Fakat eleman ekleme ve silme işlemleri bağlı liste uygulamalarında çok daha kolaydır.

## Algoritmaların Analizi:

### Çalışma Zamanı fonksiyonu : $T(n)$

- Çalışma zamanı veya koşma süresi (running time) fonksiyonu:
- ' $n$ ' boyutlu bir problemin algoritmasını çalıştırmak için gerekli zamandır ve  $T(n)$  ile gösterilir.
- Başka bir ifadeyle  $T(n)$ : bir programın veya algoritmanın işlevini yerine getirebilmesi için, döngü sayısı, toplama işlemi sayısı, atama sayısı gibi işlevlerden kaç adet yürütülmesini veren bir bağıntıdır.

## Algoritmaların Çalışma Zamanı

- Örnek: Basit *if* bildirimi

	<u>Cost</u>	<u>Times</u>
<code>if (n &lt; 0)</code>	<code>c1</code>	1
<code>absval = -n;</code>	<code>c2</code>	1
<code>else</code>		
<code>absval = n;</code>	<code>c3</code>	1

- Toplam maliyet  $\leq c1 + \max(c2, c3)$

# Tahmin için Genel Kurallar

- **Döngüler (Loops)**

- Bir döngünün çalışma zamanı en çok döngü içindeki deyimlerin çalışma zamanının iterasyon sayısı ile çarpılması kadardır.

- **İç içe döngüler (Nested Loops)**

- İç içe döngülerde grubunun içindeki deyim toplam çalışma zamanı, deyimlerin çalışma sürelerinin bütün döngülerin boyutlarının çarpımı kadardır. Bu durumda analiz içten dışa doğru yapılır.

- **Ardışık deyimler**

- Her deyim zamanı birbirine eklenir.

- **if/else**

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

## Algoritmaların Çalışma Zamanı

- **Örnek:** Basit bir döngü

- `i = 1;`
- `sum = 0;`
- `while (i <= n) {`
- `i = i + 1;`
- `sum = sum + i;`
- `}`

### Maliyet

c1  
c2  
c3  
c4  
c5

### Tekrar

1  
1  
n+1  
n  
n

- Toplam maliyet =  $c1 + c2 + (n+1)*c3 + n*c4 + n*c5 = 3n+3$
- $T(n) = 3n+3$
- Bu algoritma için gerekli zaman ***n*** ile doğru orantılıdır.

## Algoritmaların Çalışma Zamanı

- | Örnek: İç içe döngü   | Maliyet | Tekrar    |
|---|---------|-----------|
| ○ $i=1;$  | $c1$    | 1         |
| ○ $sum = 0;$  | $c2$    | 1         |
| ○ $while (i \leq n) \{$   | $c3$    | $n+1$     |
| ○ $j=1;$  | $c4$    | $n$       |
| ○ $while (j \leq n) \{$   | $c5$    | $n*(n+1)$ |
| ○ $sum = sum + i;$  | $c6$    | $n*n$     |
| ○ $j = j + 1;$  | $c7$    | $n*n$     |
| ○ $\}$  |         |           |
| ○ $i = i + 1;$  | $c8$    | $n$       |
| ○ $\}$  |         |           |
| ○ Toplam maliyet= $c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5 + n*n*c6 + n*n*c7 + n*c8$ |         |           |
| ○ Bu algoritma için gerekli zaman $n^2$ ile doğru orantılıdır.                      |         |           |

## Asimptotik Analiz (Notasyonlar)

$O$ -,  $\Omega$ -, ve  $\Theta$ -simgelemi

## Algoritmanın Büyüme Oranları

- Bir algoritmanın orantılı zaman gereksinimi **büyüme oranı (veya büyüme hızı)** olarak bilinir.
  - $T(n)$  nin büyüme oranı, algoritmanın **hesaplama karmaşıklığı**dır.
- **Hesaplama karmaşıklığı** belirli bir uygulamadan bağımsız olarak  $n$  ile değişen  $T(n)$ ' nin çalışma zamanını daha doğru bir şekilde bulmayı sağlar.
  - Genel olarak, az sayıda parametreler için karmaşıklıkla ilgilenilmez; eleman sayısı  $n$ 'nin sonsuza gitmesi durumunda  $T(n)$  büyümesine bakılır.
  - Karmaşıklığı belirtmek için asimtotik notasyon (simgelem) ifadeleri kullanılmaktadır.

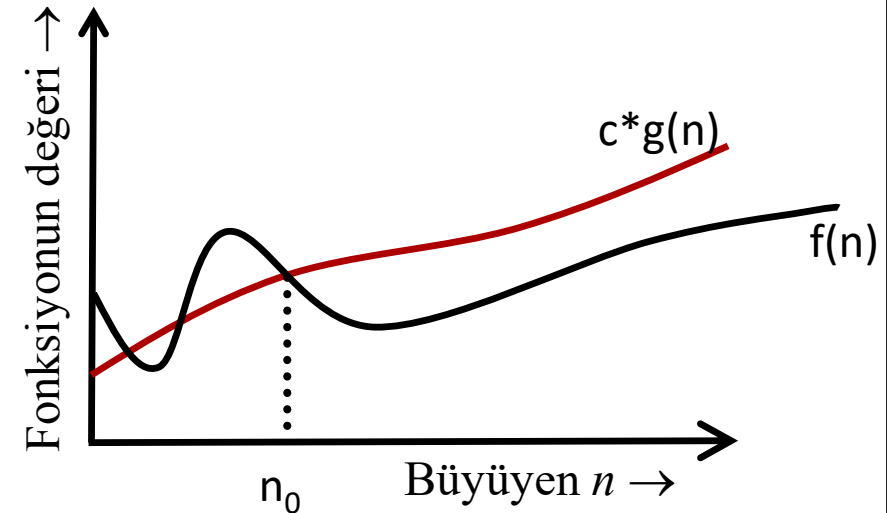


## Asimptotik simgelem (notasyon)

### *O-simgelemi (üst sınırlar)*

Tüm  $n \geq n_0$  değerleri için sabitler  $c > 0$ ,  $n_0 > 0$  ise  $0 \leq f(n) \leq cg(n)$  durumunda  $f(n) = O(g(n))$  yazabiliriz.

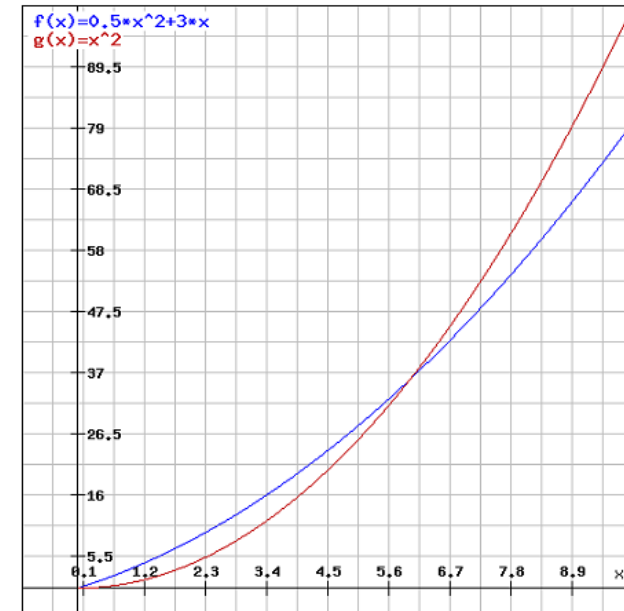
- $f(n)$  ve  $g(n)$  verilen iki çalışma zamanı fonksiyonudur.
- $f(n) \leq c \cdot g(n)$  ve  $n \geq n_0$  koşullarını sağlayan  $c$  ve  $n_0$  değerleri varsa  $f(n)$  zaman karmaşıklığı  $O(g(n))$  dir.
- Başka bir deyişle,  $n$  sayısı yeteri kadar büyük olduğunda,  $f(n)$ ,  $g(n)$  ile aynı büyüklüktedir.
- $O$ -notasyonu sabit bir katsayı içinde bir fonksiyon için üst sınırı verir



## ***O-simgelemi (üst sınırlar)***

- **Örnek:**  $2n^2 = O(n^3)$  için  $c, n_0$  değerlerini bulunuz?
- $0 \leq f(n) \leq c \cdot g(n)$  ,  $0 \leq 2n^2 \leq cn^3$
- $c=1$  için  $n_0=2$ , şartı sağlar.
- **Örnek:**  $(1/2)n^2 + 3n$  için üst sınırın  $O(n^2)$  olduğunu gösteriniz.
- $c=1$  için
- $(1/2)n^2 + 3n \leq n^2$
- $3n \leq 1/2n^2$
- $6 \leq n$ ,
- $n_0=6$

Çözüm kümesini sağlayan kaç tane  $n_0$  ve  $c$  çifti olduğu önemli değildir. Tek bir çift olması notasyonun doğruluğu için yeterlidir.



## Algoritmanın Büyüme Oranları

- Büyüme oranlarına bakarak iki algoritmanın verimliliğini karşılaştırabiliriz.
  - $n'$  nin yeterince büyük değerleri için düşük büyüme oranına sahip algoritma her zaman daha hızlıdır.
  - Örneğin;  $f(n)=n^2+3n+5$  ifadesinin büyüme oranı  $O(n^2)$  dir.
- Algoritma tasarımcılarının amacı, çalışma zaman fonksiyonu olan  $f(n)$  nin mümkün olduğu kadar düşük büyüme oranı sahip bir algoritma olmasıdır.

## O-notasyonu

- Örnek
- $3n^2 + 2n + 5 = O(n^2)$  olduğunu gösteriniz
- $10n^2 = 3n^2 + 2n^2 + 5n^2$
- $\geq 3n^2 + 2n + 5, n \geq 1$
- $c = 10, n_0 = 1$

## Ortak Büyüme Oranları

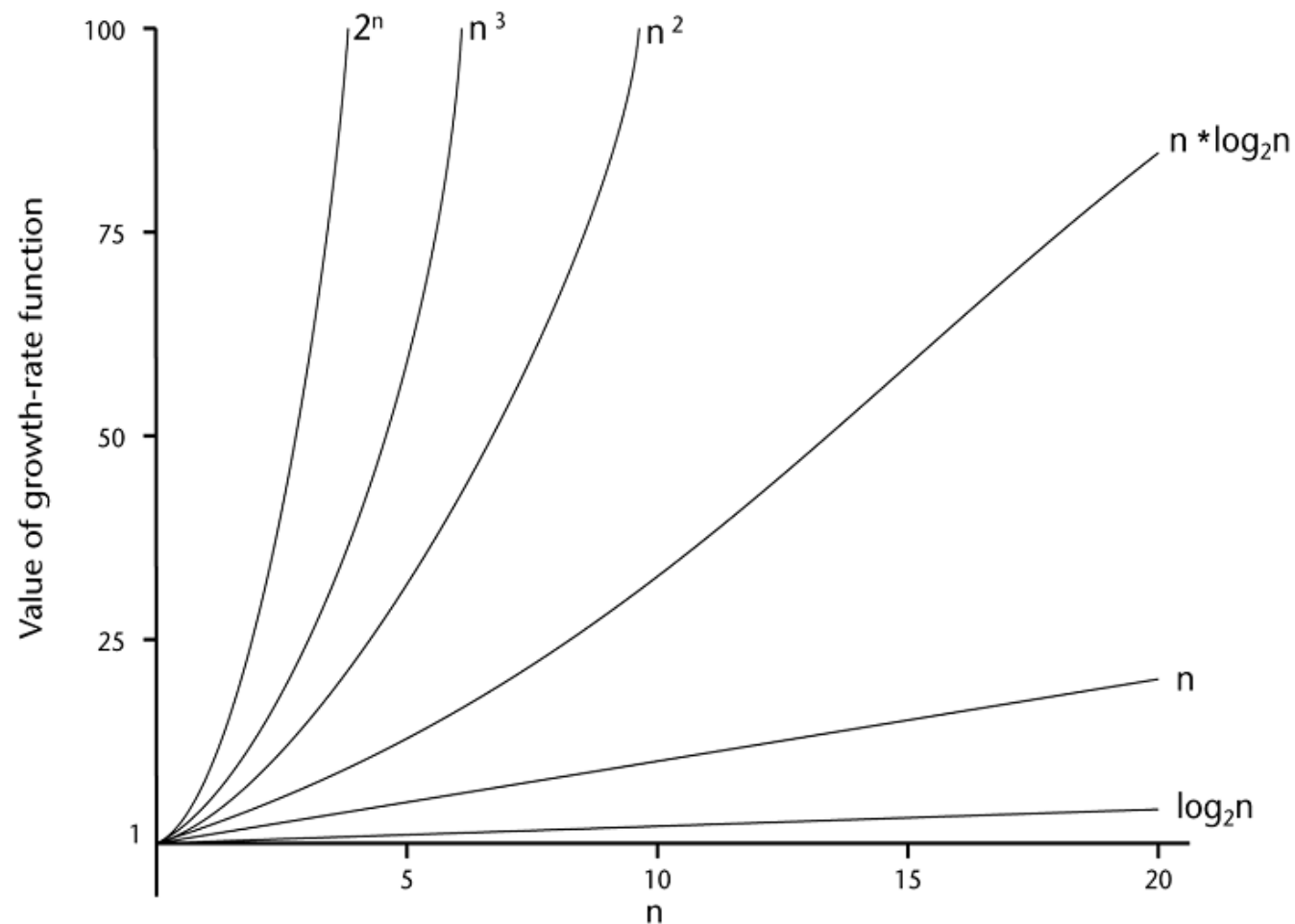
Maliyet artar

Fonksiyon	Büyüme oranı ismi
$c$	Sabit, komut bir veya birkaç kez çalıştırılır. Yenilmez!
$\log n$	Logaritmik, Büyük bir problem, her bir adımda sabit kesirler tarafından orijinal problemin daha küçük parçalara ayrılması ile çözülür. İyi hazırlanmış arama algoritmalarının tipik zamanı
$\log^2 n$	Karasel logaritmik
$n$	Doğrusal, Küçük problemlerde her bir eleman için yapılır. Hızlı bir algoritmadır. N tane veriyi girmek için gereken zaman.
$n \log n$	Doğrusal çarpanlı logaritmik. Çoğu sıralama algoritması
$n^2$	Karasel. Veri miktarı az olduğu zamanlarda uygun ( $n < 1000$ )
$n^3$	Kübik. Veri miktarı az olduğu zamanlarda uygun ( $n < 1000$ )
$2^n$	İki tabanında üssel. Veri miktarı çok az olduğunda uygun ( $n \leq 20$ )
$10^n$	On tabanında üssel
$n!$	Faktöriyel
$n^n$	$n$ tabanında üstel ( çoğu ilginç problem bu kategoride)

## Büyüme oranı fonksiyonlarının karşılaştırılması

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
$n$	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
$n * \log_2 n$	30	664	9,965	$10^5$	$10^6$	$10^7$
$n^2$	$10^2$	$10^4$	$10^6$	$10^8$	$10^{10}$	$10^{12}$
$n^3$	$10^3$	$10^6$	$10^9$	$10^{12}$	$10^{15}$	$10^{18}$
$2^n$	$10^3$	$10^{30}$	$10^{301}$	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

# Büyüme oranı fonksiyonlarının karşılaştırılması



# Büyüme oranı fonksiyonlarının karşılaştırılması

- Verimli algoritmaları geliştirmenin önemi:

Dizi boyutu	Sıralı arama	İkili (Binary) arama
128	128	8
1,048,576	1,048,576	21

- Verilen zaman karmaşıklığı ile algoritmalar için yürütme zamanı

n	$f(n)=n$	$f(n)=n \log n$	$f(n)=n^2$	$f(n)=2^n$
20	0.02 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	1 ms
$10^6$	1 $\mu$ s	19.93 ms	16.7 dk	31.7 yıl
$10^9$	1s	29.9 s	31.7 yıl	!!! yüzyıllar



# Büyüme oranı fonksiyonlarının özellikleri

- **1-** Bir algoritmanın büyüme oranı fonksiyonunda düşük dereceli terimler, sabitler ve katsayılar ihmal edilebilir.
  - $O(n^3+4n^2+3n) \rightarrow O(n^3)$
  - $O(8n^4) \rightarrow O(n^4)$
- **2-** Algoritmanın büyüme fonksiyonlarını birleştirebiliriz.
  - $O(f(n))+O(g(n)) = O(f(n)+g(n))$
  - $O(n^3)+O(4n^2) \rightarrow O(n^3+4n^2) \rightarrow O(n^3)$
  - Çarpma içinde benzer kurallara sahiptir.

## Büyüme oranı analizi ile ilgili problemler

- **1-** Daha küçük büyüme oranına sahip bir algoritma yeterince büyük olmayan belirli  $n$  değerleri için daha hızlı büyüme oranına sahip algoritmadan hızlı çalışmaz.
- **2-** Aynı büyüme oranına sahip algoritmalar çalışma zamanı fonksiyonlarındaki sabitlerden dolayı farklı çalışma zamanlarına sahip olabilirler. Ama iki algoritmanın da kırılma noktası aynı  $n$  değerine sahiptir.

## Notasyonlarda eşitlik"=" gösterimi

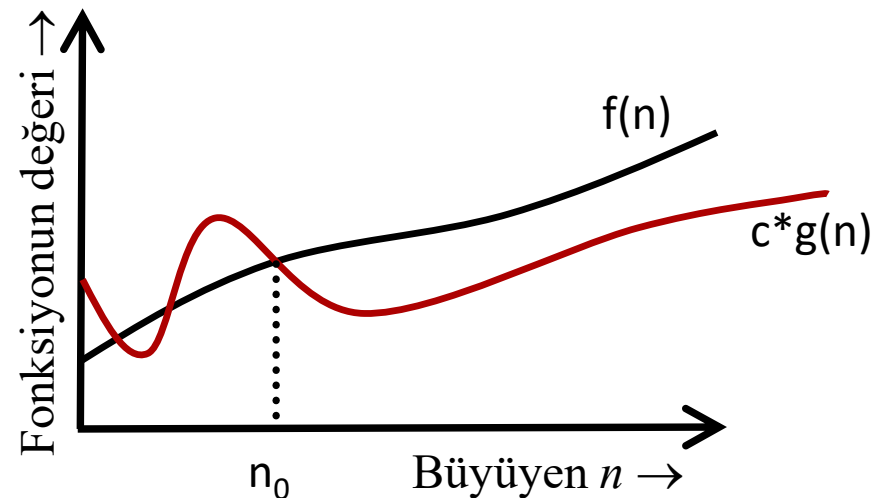
- $A=B$  ise  $B = A$  anlamında değil mi?
- Fakat,  $f(n) = O(g(n))$  ,  $O(g(n)) = f(n)$  anlamına gelmez. Burada tek eşitlik söz konusudur.
- Burada "=", üyelik işlemi ( $\in$ ) olarak tercih edilmiştir.
  - $f(n) = O(g(n)) \rightarrow f(n) \in O(g(n))$  dir
  - $O(g(n))$  bir küme anlamına gelir.
  - $f(n) = O(g(n)) \rightarrow O(g(n)) = \{ f(n) \}$  gösterimi doğrudur.

## Diğer Asimptotik Notasyonlar

### $\Omega$ -simgelemi (alt sınırlar)

$$\Omega(g(n)) = \{ f(n) : \text{tüm } n \geq n_0 \text{ değerlerinde} \\ c > 0, n_0 > 0 \text{ ise,} \\ 0 \leq cg(n) \leq f(n) \}$$

- Her durumda  $f(n) \geq c g(n)$  ve  $n \geq n_0$  koşullarını sağlayan pozitif, sabit  $c$  ve  $n_0$  değerleri bulunabiliyorsa  $f(n) = \Omega(g(n))$  dir.



## $\Omega$ notasyonu-Örnek

- $2n + 5 \in \Omega(n)$  olduğunu gösteriniz
  - $n_0 \geq 0$ ,  $2n+5 \geq n$ , olduğundan sonuç elde etmek için  $c=1$  ve  $n_0 = 0$  alabiliriz.
- $5*n^2 - 3*n = \Omega(n^2)$  olduğunu gösteriniz.
  - $5*n^2 - 3*n \geq n^2$ ,  $c=1$ ,  $n_0 = 0$  değerleri için sağlar

## Examples

- $5n^2 = \Omega(n)$

$\exists c, n_0$  such that:  $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$  and  $n_0 = 1$

- $100n + 5 \neq \Omega(n^2)$

$\exists c, n_0$  such that:  $0 \leq cn^2 \leq 100n + 5$

$100n + 5 \leq 100n + 5n \ (\forall n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

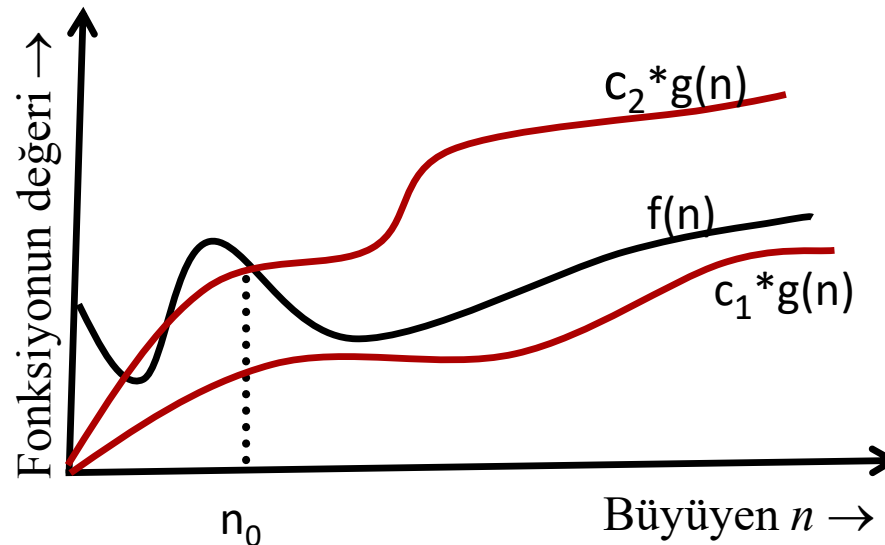
Since  $n$  is positive  $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

$\Rightarrow$  contradiction:  $n$  cannot be smaller than a constant

- $n = \Omega(2n), n^3 = \Omega(n^2), n = \Omega(\log n)$

## ⊖ Notasyonu – Sıkı Sınırlar

- Her durumda  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  ve  $n \geq n_0$  koşullarını sağlayan pozitif, sabit  $c_1, c_2$  ve  $n_0$  değerleri bulunabiliyorsa  $f(n) = \Theta(g(n))$  ifadesi doğrudur.



## $\Theta$ notasyonu- Örnek

- $f(n) = 2n + 5 \in \Theta(n)$ .
  - $2n \leq 2n+5 \leq 3n$ , tüm  $n \geq 5$  için
- $f(n) = 5*n^2 - 3*n \in \Theta(n^2)$ .
  - $4*n^2 \leq 5*n^2 - 3*n \leq 5*n^2$ , tüm  $n \geq \frac{4}{3}$  için



## Examples

- $n^2/2 - n/2 = \Theta(n^2)$

- $\frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \quad \forall n \geq 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$

- $\frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \quad (\forall n \geq 2) = \frac{1}{4} n^2 \Rightarrow \quad c_1 = \frac{1}{4}$

- $n \neq \Theta(n^2): c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$  only holds for:  $n \leq 1/c_1$

- $6n^3 \neq \Theta(n^2): c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$  only holds for:

$$n \leq c_2 / 6$$

n değerini keyfi olarak belirlemek imkansızdır. Çünkü  $c_2$  sabittir.

## Another example

- Prove that  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

- **Determine  $c_1$ ,  $c_2$  and  $n_0$  such that**

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\frac{1}{2} - \frac{3}{n} \leq c_2 \rightarrow n \geq 1, c_2 \geq \frac{1}{2}$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \rightarrow n \geq 7, c_1 \leq \frac{1}{14}$$

- $c_1 = 1/14, c_2 = 1/2, n_0 = 7$

For any polynomial  $p(n) = \sum_{i=0}^d a_i n^i$   
 $p(n) = \Theta(n^d)$

## O, $\Omega$ ve $\Theta$ notasyonları arasındaki ilişkiler

- Eğer  $g(n) = \Omega(f(n))$  ise  $\rightarrow f(n) = O(g(n))$
- Eğer  $f(n) = O(g(n))$  ve  $f(n) = \Omega(g(n))$  ise  $\rightarrow f(n) = \Theta(g(n))$
- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

## $O$ , $\Omega$ ve $\Theta$ notasyonları arasındaki ilişkiler

- $n$ 'nin büyük olduğu ve sabitlerin elendiği durumlarda.
- $O(g(n))$  düşünürsek  $f(n)$  ile “eşit veya büyük”
  - Üstten sınır:  $f(n)$  ile “aynı hızda veya hızlı büyür”
- $\Omega(g(n))$  düşünürsek  $f(n)$  ile “eşit veya küçük”
  - Alttan sınır:  $f(n)$  ile “aynı hızda veya yavaş büyür”
- $\Theta(g(n))$  düşünürsek  $f(n)$  ile “eşit”
  - Alttan ve Üsten sınır : büyüme oranları eşit
- Her bağıntı için alt ve üst sınırlar aynı zamanda belirlenemiyorsa veya bu sınırlardan sadece biri belirlenebiliyorsa, o sınıra göre notasyon gösterimi yapılır.

## $\Theta$ Notasyonunun özellikleri

- $f(n) = \Theta(f(n))$ , yansıtma (reflexivity) özelliği
- $g(n) = \Theta(f(n))$  olduğu durumda  $f(n) = \Theta(g(n))$  dir. simetri (symmetry) özelliği
- Eğer  $f(n) = \Theta(g(n))$  ve  $g(n) = \Theta(h(n))$  ise  $f(n) = \Theta(h(n))$  geçişme (transitivity) özeliği
- $c > 0$  olduğu her hangi bir durum için, bu fonksiyon  $c \cdot f(n) = \Theta(f(n))$  dir.
- Eğer  $f_1 = \Theta(g_1(n))$  ve  $f_2(n) = \Theta(g_2(n))$  ise  $(f_1 + f_2)(n) = \Theta(\max\{g_1(n), g_2(n)\})$
- Eğer  $f_1 = \Theta(g_1(n))$  ve  $f_2(n) = \Theta(g_2(n))$  ise  $(f_1 \cdot f_2)(n) = \Theta((g_1 \cdot g_2)(n))$
- Simetri özelliği dışındaki diğer özellikler  $O$  ve  $\Omega$  notasyonlarında da vardır.

## o-notasyonu ve $\omega$ -notasyonu

- $O$ -notasyonu ve  $\Omega$ -notasyonu  $\leq$  ve  $\geq$  gibidir.
- $o$ -notasyonu ve  $\omega$ -notasyonu  $<$  ve  $>$  gibidir.
- $o$ -notasyonunda üst sınıra,  $\omega$  notasyonunda ise alt sınıra eşitlik yoktur. Bundan dolayı üst ve alt sınırları sıkı bir asimptotik notasyon değildir.
- Öncekinden tek farklılığı,  $c$  katsayısı ve bir  $n_0$  değeri var demek yerine, her  $c$  katsayısı için başka bir  $n_0$  olacağını kabul etmek.

## o-notasyonu

- $o(g(n)) = \{ f(n) : \text{tüm } n \geq n_0 \text{ değerlerinde}$   
 $c > 0$  sabiti için  $n_0$  sabiti varsa  
 $0 \leq f(n) < cg(n). \}$
- Çalışma sürelerinin karşılaştırılması için kullanılır. Eğer  $f(n) = o(g(n))$ , ise  $g(n)$ ,  $f(n)$  fonksiyonundan daha ağırlıklıdır. Sıkı bir üst sınır vermez.
- $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$
- o-notasyonunda  $n$  sonsuza gittiğinde  $f(n)$  fonksiyonu  $g(n)$  fonksiyonu karşısında önemini kaybeder.

## o-notasyonu

- Örnek:  $n^2/2 \in o(n^3)$ ,
  - $\lim_{n \rightarrow \infty} (n^2/2)/n^3 = \lim_{n \rightarrow \infty} 1/2n = 0$
- Örnek:  $2n = o(n^2)$ , fakat  $2n^2 \neq o(n^2)$
- Örnek:  $2n^2 = o(n^3)$  ( $n_0 = 2/c$ )
- **Önerme:**  $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$



## $\omega$ -notasyonu

- $\omega(g(n)) = \{ f(n) : \text{tüm } n \geq n_0 \text{ değerlerinde}$   
 $c > 0$  sabiti için  $n_0$  sabiti varsa  
 $0 \leq c \cdot g(n) < f(n) \}$
- Çalışma sürelerinin karşılaştırılması için kullanılır. Eğer  $f(n) = \omega(g(n))$ , ise  $g(n)$ ,  $f(n)$  fonksiyonundan daha ağırlıklıdır. Sıkı bir alt sınır vermez.
  - $\lim_{n \rightarrow \infty} f(n)/g(n) = \infty$ , eğer limit varsa
- Örnek:  $n^2/2 \in \omega(n)$ ,
  - $\lim_{n \rightarrow \infty} (n^2/2)/n = \infty$ , fakat  $n^2/2 \neq \omega(n^2)$
- Örnek:  $n^{1/2} = \omega(\log n)$ , ( $n_0 = 1 + 1/c$ )

# Asimptotik Notasyonlar

- $f(n) = O(g(n)) \quad \cong \quad f \leq g$
- $f(n) = \Omega(g(n)) \quad \cong \quad f \geq g$
- $f(n) = \Theta(g(n)) \quad \cong \quad f = g$
- $f(n) = o(g(n)) \quad \cong \quad f < g$
- $f(n) = \omega(g(n)) \quad \cong \quad f > g$

## Asimptotik Notasyonların Karşılaştırılması

- Geçişlilik(Transitivity):
  - $f(n) = \Theta(g(n))$  ve  $g(n) = \Theta(h(n))$  ise  $f(n) = \Theta(h(n))$ ,
  - $f(n) = O(g(n))$  ve  $g(n) = O(h(n))$  ise  $f(n) = O(h(n))$ ,
  - $f(n) = \Omega(g(n))$  ve  $g(n) = \Omega(h(n))$  ise  $f(n) = \Omega(h(n))$ ,
  - $f(n) = o(g(n))$  ve  $g(n) = o(h(n))$  ise  $f(n) = o(h(n))$ ,
  - $f(n) = \omega(g(n))$  ve  $g(n) = \omega(h(n))$  ise  $f(n) = \omega(h(n))$ .
- Dönüştürülebilirlik veya yansıma(Reflexivity):
  - $f(n) = \Theta(f(n))$ ,
  - $f(n) = O(f(n))$ ,
  - $f(n) = \Omega(f(n))$ .
- Simetri(Symmetry):
  - $g(n) = \Theta(f(n))$  olduğu durumda,  $f(n) = \Theta(g(n))$
- Transpose symmetry:
  - $g(n) = \Omega(f(n))$  olduğu durumda,  $f(n) = O(g(n))$ ,
  - $g(n) = \omega(f(n))$  olduğu durumda,  $f(n) = o(g(n))$ .

## En iyi (Best), En kötü (Worst), Ortalama(Average) Durum Analizi

- **En iyi durum (best case):** Bir algoritma için, çalışma zamanı, maliyet veya karmaşıklık hesaplamalarında en iyi sonucun elde edildiği duruma “en iyi durum” denir. Bir giriş yapısında hızlı çalışan yavaş bir algoritma ile hile yapmak. (gerçek dışı) Ör: Bütün elemanların sıralı olduğu durum.
- **En kötü durum (worst case):** Tüm olumsuz koşulların oluşması durumunda algoritmanın çözüm üretmesi için gerekli maksimum çalışma zamanıdır. (genellikle). Ör: Bütün elemanlar ters sıralı.
- **Ortalama durum (avarage case):** Giriş parametrelerin en iyi ve en kötü durum arasında gelmesi ile ortaya çıkan durumda harcanan zamandır. Fakat bu her zaman ortalama durumu vermeyebilir. (bazen) Ör: Elemanların yaklaşık yarısı kendi sırasındadır.

## En iyi (Best), En kötü (Worst), Ortalama(Average) Durum Analizi

- **Dizi arama**

- Worst case =  $O(n)$

- Average case =  $O(n)$

- **Quick sort**

- Worst case =  $O(n^2)$

- Average case =  $O(n \log n)$

- **Merge Sort, Heap Sort**

- Worst case =  $O(n \log n)$

- Average case =  $O(n \log n)$

- **Bubble sort**

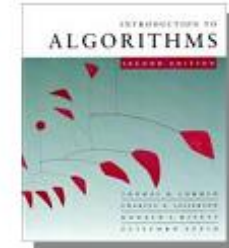
- Worst case =  $O(n^2)$

- Average case =  $O(n^2)$

- **Binary Search Tree:** Bir elaman için arama

- Worst case =  $O(\log n)$

- Average case =  $O(\log n)$



## Diziler (Arrays)

- Diziler, tipleri homojen olan birden fazla elemandan oluşan veri grubudur. Diziler tek boyutlu, iki boyutlu, üç boyutlu, vb. şekilde tanımlanabilirler.
- Diziler genelde diğer veri yapılarında kullanılan bir veri yapısıdır. Diziler **statik veriler** olarak da isimlendirilebilirler.
- Örnek: Tek boyutlu A dizisi

0	1	2	3	4	5	6	7	8	9
6	7	12	23	46	78	12	5	8	2



## Diziler-Doğrusal Arama (Search)

- Statik ve sıralanmamış veri dizisi üzerinde arama algoritması (Linear arama).

### Linear Arama( $n, x$ )

- $i \leftarrow 1, \text{Veri\_Var} \leftarrow 0$
- $(i \leq n)$  ve  $\sim (\text{Veri\_Var})$  olduğu sürece devam et
- eğer  $\text{Dizi}[i] = x$  ise
- $\text{Veri\_Var} \leftarrow 1$
- $i \leftarrow i + 1$

- $x$ , dizi içinde aranacak veri ve  $n$  dizinin boyutudur.
- $\sim$  simgesinin anlamı önüne geldiği mantıksal ifadenin değilini almaktır.

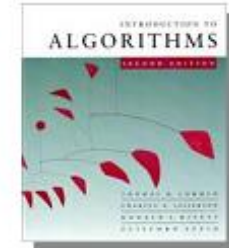
# Diziler-Arama (Serach) Analizi

- Başarısız Arama :  $O(n)$
- Başarılı Arama:
  - **Best-Case:** x dizinin ilk elemanı ise:  $O(1)$
  - **Worst-Case:** x dizinin son elemanı ise:  $O(n)$
  - **Average-Case:** Listedeki her bir sayıyı bir kez aradığımızı düşünelim. Anahtar karşılaştırmalarının sayısı  $1,2,\dots,n$  ise :  $O(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{en iyi durum} \\ \Theta(n) & \text{en kötü durum} \\ \Theta(n) & \text{ortalama durumu} \end{cases}$$

$$\frac{\sum_{i=1}^n i}{n} = \frac{(n^2 + n)/2}{n}$$





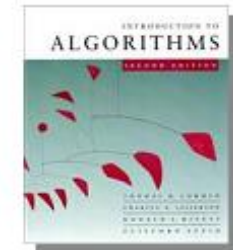
## Diziler-Arama (Serach) Analizi

- Lineer arama algoritması, herhangi bir dizi üzerinde yapılan bir aramadır.
- Dizinin elemanları sıralı olması veya olmaması herhangi bir anlam ifade etmez. Bu aramada aranacak eleman dizinin bütün elemanları ile karşılaştırılır ve bu işleme, aranan eleman dizide bulununcaya kadar veya dizide olmadığı kesinlik kazanıncaya kadar devam edilir.
- Eğer dizinin elemanları sıralı ise, bu durumda mertebesi daha iyi olan bir algoritma kullanılabilmektedir.
  - **İkili Arama (Binary Search)**

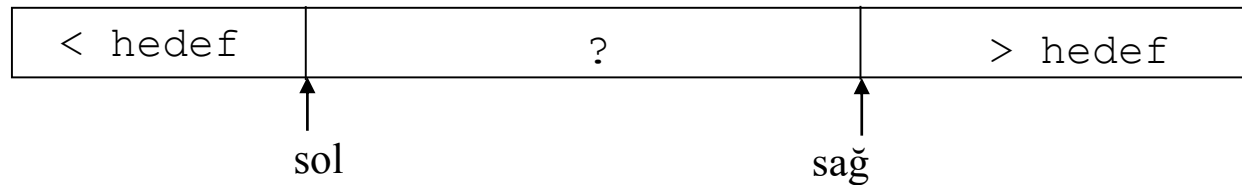


# İkili Arama (Binary Search)

- İkili aramada dizi sıralanmış olduğundan, dizinin ortasında bulunan sayı ile aranan sayıyı karşılaştırarak arama boyutunu yarıya düşürülür ve bu şekilde devam edilir.
- Bu algoritmanın temel mantığı aranacak elemanı dizinin ortasındaki eleman ile karşılaştırıp, aranacak eleman eğer bu elemana eşitse, amaca ulaşılmıştır. Eğer eşit değilse, bu durumda aranacak eleman dizinin hangi parçası içinde olabileceği kararı verilir. Bu sayede arama boyutu yarıya düşürülür ve bu şekilde devam edilir.



# İkili Arama (Binary Search)

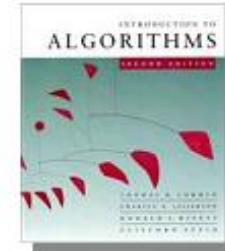


- Statik ve sıralı veri dizisi üzerinde ikili arama algoritması.

**İkili Arama(Dizi,n,x,bulundu, yeri)**

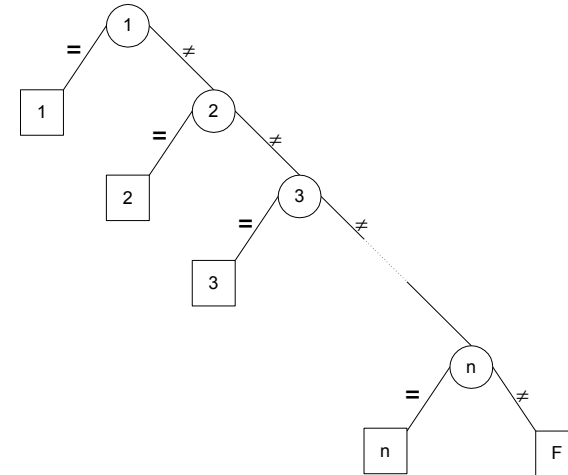
1. Yerel değişkenler  
alt, ust, orta : integer;
2.  $Ust \leftarrow n$ ,  $alt \leftarrow 1$ ,  $bulundu \leftarrow 0$
3.  $(bulundu = 0)$  ve  $(ust \geq alt)$  olduğu sürece devam et
4.  $orta \leftarrow \lfloor (alt + ust) / 2 \rfloor$
5. eğer  $x = Dizi[orta]$  ise
6.  $bulundu \leftarrow 1$
7. değil ve eğer  $x < Dizi_2[orta]$  ise
8.  $ust \leftarrow orta - 1$
9. değilse
10.  $alt \leftarrow orta + 1$
11.  $yeri \leftarrow orta$

# İkili Arama (Binary Search)

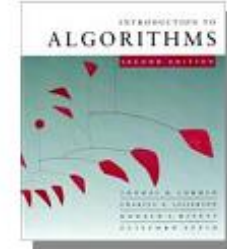


- İkili aramanın mantığı veya işleyiş şekli ağaç şeklinde gösterilebilir ve bu ağaca **karşılaştırma ağacı** veya **karar ağacı** denir. **Karşılaştırma ağacı**, bir algoritmanın yapmış olduğu karşılaştırmaların hepsinin temsil edildiği bir ağaçtır.
- Örneğin, statik veri yapıları üzerinde Lineer arama algoritmasının karşılaştırma ağacı;

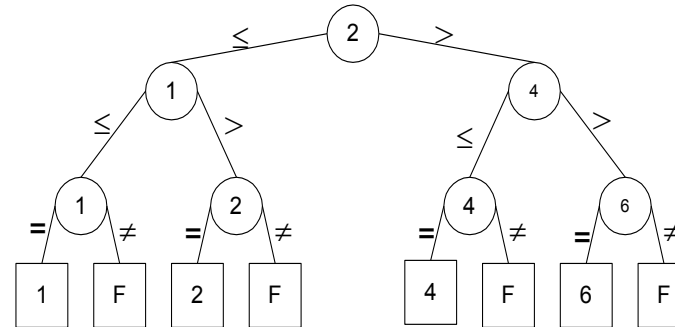
Aranacak bir elemanın dizinin içinde olup olmadığını test etmek amacıyla dizinin her elemanı ile karşılaştırma yapıldığından ağaç tek dal üzerine büyümektedir ve ağacın bir tarafı hiç yok iken, diğer tarafı çok büyüyebilir. Buradan da rahatlıkla görülebileceği gibi Lineer arama algoritması iyi bir algoritma değildir.



# İkili Arama (Binary Search)

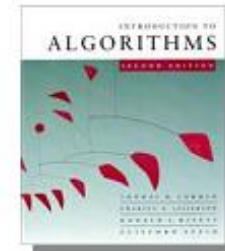


- Statik veri yapıları üzerinde yapılan ikili arama algoritması için karşılaştırma ağacı;



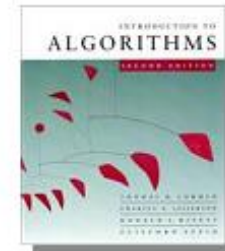
- Görülen karşılaştırma ağacı dengeli bir ağaçtır. Bazı durumlarda karşılaştırma ağacının bütün yaprakları aynı seviyede olmayabilir.

# İkili Arama (Binary Search)

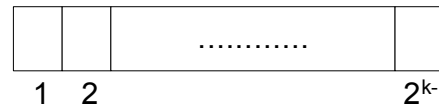


- Bu algoritmanın analizini yapmak için ilk olarak  $n=2^k$  kabulü yapılsın.
- $\frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \dots, \frac{n}{2^k} = 1, n=2^k$
- Bu kabulün yapılması genelliliği bozmayacaktır. Algoritma çalıştığında ilk bakılacak eleman  $2^{k-1}$  endeksli eleman olacaktır. Eğer eşitlik varsa, amaca ulaşılmıştır. Eşitlik yoksa geriye kalan ve her birinin boyutu  $2^{k-1}$  olan dizi parçalarının hangisinde aranan elemanın olacağına karar verilir.

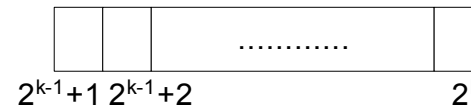
# İkili Arama (Binary Search)



- Eğer  $x$  değişkeninin değeri  $Dizi[2^{k-1}]$  elemanın değerinden küçükse,

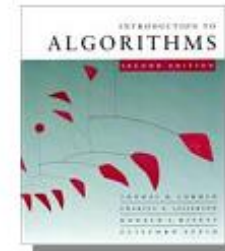


- $x$  elemanı bu parçanın içinde olabilir, diğer parçanın içinde olması mümkün değildir. Eğer  $x$  değişkeninin değeri  $Dizi[2^{k-1}]$  elemanın değerinden büyükse,



- $x$  elemanı bu parçanın içinde olabilir, diğer parçanın içinde olması olasılığı sıfırdır. Bu şekilde geriye kalan parçanın içindeki eleman sayısı  $2'$  nin kuvveti kadar eleman kalacaktır. Bundan dolayı ortadaki elemanı bulmak için taban veya tavan fonksiyonuna ihtiyaç duyulmayacaktır. Dizi üzerinde yapılacak bölme sayısı  $\log n$  olacaktır.

# İkili Arama (Binary Search)

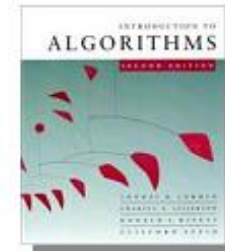


- Bir dizi için başarılı arama sayısı  $n$  ve 1 tanede başarısız arama sayısı yapılabilir. Toplam  $n+1$  tane arama yapılabilir.
- Bu algoritmada en iyi durum aramasında döngü kısmı bir sefer çalışır ve en kötü durumda döngü kısmı  $\lg n + 1$  sefer çalışır.
- Ortalama arama zamanına bakılacak olursa, toplam arama sayısı ile arama adım sayısı çarpılır ve  $n$  değerine bölünür.
- Asimptotik notasyonda sabit katsayılar ihmal edildiğinden dolayı, ikili arama algoritmasının mertebesi  $T(n)$ ,

$$T(n) = \begin{cases} \Theta(1) & \text{en iyi durum} \\ \Theta(\lg n) & \text{en kötü durum} \\ \Theta(\lg n) & \text{ortalama durum} \end{cases} \quad \frac{\sum_{i=1}^n \log n}{n} = \frac{n \log n}{n} = \log n$$



# İkili Arama (Binary Search)



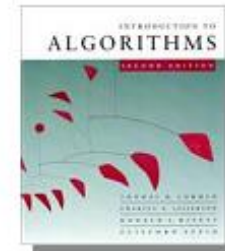
- Bu algoritmanın da en kötü durumu ile ortalama durumun asimptotik davranışları aynıdır. Fakat en kötü durum ve ortalama durum mertebeleri (çalışma zamanı) logaritmik olduğundan, lineer aramaya göre çok iyi olan bir algoritmadır.
- Eğer bir dizi içindeki veriler sıralı ise, her zaman ikili arama algoritmasını kullanmak sistemin performansını artırır.

# Binary Arama Analizi

- Başarısız Arama :  $O(n)$
- Başarılı Arama:
- Best-Case:  $O(1)$
- Worst-Case:  $O(\log n)$
- Average-Case:  $O(\log n)$

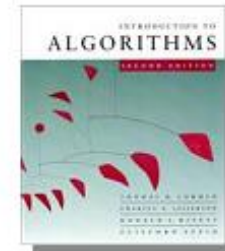
```
int ikiliArama(int A[], int N, int sayi)
{
    sol = 0;
    sag = N-1;
    while (sol <= sag){
        int orta = (sol+sag)/2;
        if (A[orta] == sayi) return orta;
        else if (sayi < A[orta]) sag = orta - 1;
        else sol = orta+1;
    }
    return -1;
}
```

# İkili Arama (Binary Search)



- İkili arama algoritmasının performansını ölçmek için, bu algoritmanın  $T(n)$  zaman bağıntısının tekrarlı bağıntısı (özyinelemeli bağıntı) elde edilebilir. İkili arama algoritmasının  $T(n)$  bağıntısı
- $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$   
şeklinde olur. Bu tekrarlı bağıntının çözümü iteratif (iterasyon) yapılırsa (taban fonksiyonunu ihmal ederek),
  - $T(n) = T(n/2) + \Theta(1)$
  - $= \Theta(1) + (\Theta(1) + T(n/2^2))$
  - $= \Theta(1) + (\Theta(1) + (\Theta(1) + T(n/2^3)))$
  - $= (\lg n - 1) \Theta(1) + T(n/2^{\lg n})$
  - $= (\lg n - 1) \Theta(1) + T(1)$
- elde edilir.

# İkili Arama (Binary Search)



- İkili arama için  $T(1)=\Theta(1)$  olur, bundan dolayı
- $T(n) = (\lg n - 1)\Theta(1) + \Theta(1)$
- $= (\lg n)\Theta(1)$
- $= \Theta(\lg n)$  olur.
- Master yöntemi kullanılırsa,  $f(n) = \Theta(1)$  ve  $a=1$ ,  $b=2$ ,
- $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$  olduğundan olur. Bundan dolayı  $f(n) = \Theta(n^{\log_b a})$  olur. Master yönteminde bu şart sağlandığında
- $T(n) = \Theta(n^{\log_b a} \lg n)$
- $= \Theta(n^0 \lg n)$
- $= \Theta(\lg n)$
- elde edilir.
- Bu yöntemler ilerleyen bölümde detaylı olarak ele alınacaktır.