VERİ YAPILARI VE ALGORİTMALAR

BLM2512 Gr.2

2020-2021 Bahar Yarıyılı (Uzaktan Eğitim)

Dr.Öğr.Üyesi Göksel Biricik

ALGORİTMA ANALİZİ TEMELLERİ

Algoritma Analizi

- Algoritma Analizi, farklı çözüm yöntemlerinin başarımını analizi etmeyi sağlayan araçlar sunan Bilgisayar Bilimlerinde bir alandır.
- Neden algoritmayı analiz ederiz?
  - Algoritmanın performansını ölçmek için
  - Farklı algoritmalarla karşılaştırmak için
  - Daha iyisi mümkün mü? Olabileceklerin en iyisi mi?
- Aynı problemi çözen iki algoritmanın performansı nasıl karşılaştırılır?
  - Teorik analiz
  - Empirik analiz

Empirik Yöntemle Algoritma Analizi

- Basit yöntem, algoritmaların bir programlama dili ile gerçekleştirilerek çalıştırılması ve işletim sürelerinin karşılaştırılmasıdır. Algoritma yerine programın başarımın karşılaştırılmasının da zorluk ve sakıncaları bulunmaktadır.
  - Algoritma nasıl kodlandı?
  - İşletim sürelerinin karşılaştırılması gerçekleştirimlerin karşılaştırılmasıdır.
  - Kodlar, programcının stiline bağlıdır ve algoritma başarımının düzgün ölçülmesine engel olabilir.
  - Kullanılan bilgisayar sonucu etkileyebilir.
- Algoritma başarımının aşağıdaki unsurlardan bağımsız hesaplanabilmesi gerekir:
  - Belirli bir bilgisayarın özelliklerinden
  - Kullanılan verinin özelliklerinden
  - Programlama dili avantaj/dezavantajlarından

Teorik Algoritma Analizi

- Algoritma analizi sırasında, algoritmanın başarımını gerçekleştirim, bilgisayar ya da veriden bağımsız değerlendirmeye olanak sağlayan matematiksel yöntemler kullanılabilmesi gereklidir.
- Algoritma Analizinde;
  - Belirli bir çözümün etkinliğini belirlemek için öncelikle önemli işlemlerin sayısı belirlenir.
  - Ardından büyüme fonksiyonları (growth function) kullanılarak algoritma etkinliği ifade edilir.

Algoritmaların İşletim Süresi

- Bir algoritmadaki her bir işlemin bir maliyeti vardır.
  - Her işlem belirli bir sürede tamamlanır.
  - `count = count + 1;` // Belirli bir süre gerekir. Bu süre sabittir.

- Birden çok işlem
  - `count = count + 1;` Maliyet: c1
  - `sum = sum + count;` Maliyet : c2
- ➔ Toplam Maliyet = c1 + c2

Algoritmaların İşletim Süresi

- If komutu Maliyeti:

	Maliyet	Tekrar
`if (n < 0)`	c1	1
`  absV = -n`	c2	1
`else`		
`  absV = n;`	c3	1

  - Toplam Maliyet = c1 + max(c2,c3)

Algoritmaların İşletim Süresi

- Basit Döngü Maliyeti:

	Maliyet	Tekrar
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  i = i + 1;`	c4	n
`  sum = sum + i;`	c5	n
`}`		

Toplam Maliyet = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ Bu algoritma için gereken süre ***n*** ile doğru orantılıdır.

Algoritmaların İşletim Süresi

- İç içe Döngü Maliyeti:

	Maliyet	Tekrar
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  j=1;`	c4	n
`  while (j <= n) {`	c5	n*(n+1)
`  sum = sum + i;`	c6	n*n
`  j = j + 1;`	c7	n*n
`}`		
`i = i + 1;`	c8	n
`}`		

```
Total Cost  =  c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5 +
 n*n*c6 + n*n*c7 + n*c8
```

➔ Bu algoritma için gereken süre n^2 ile doğru orantılıdır.

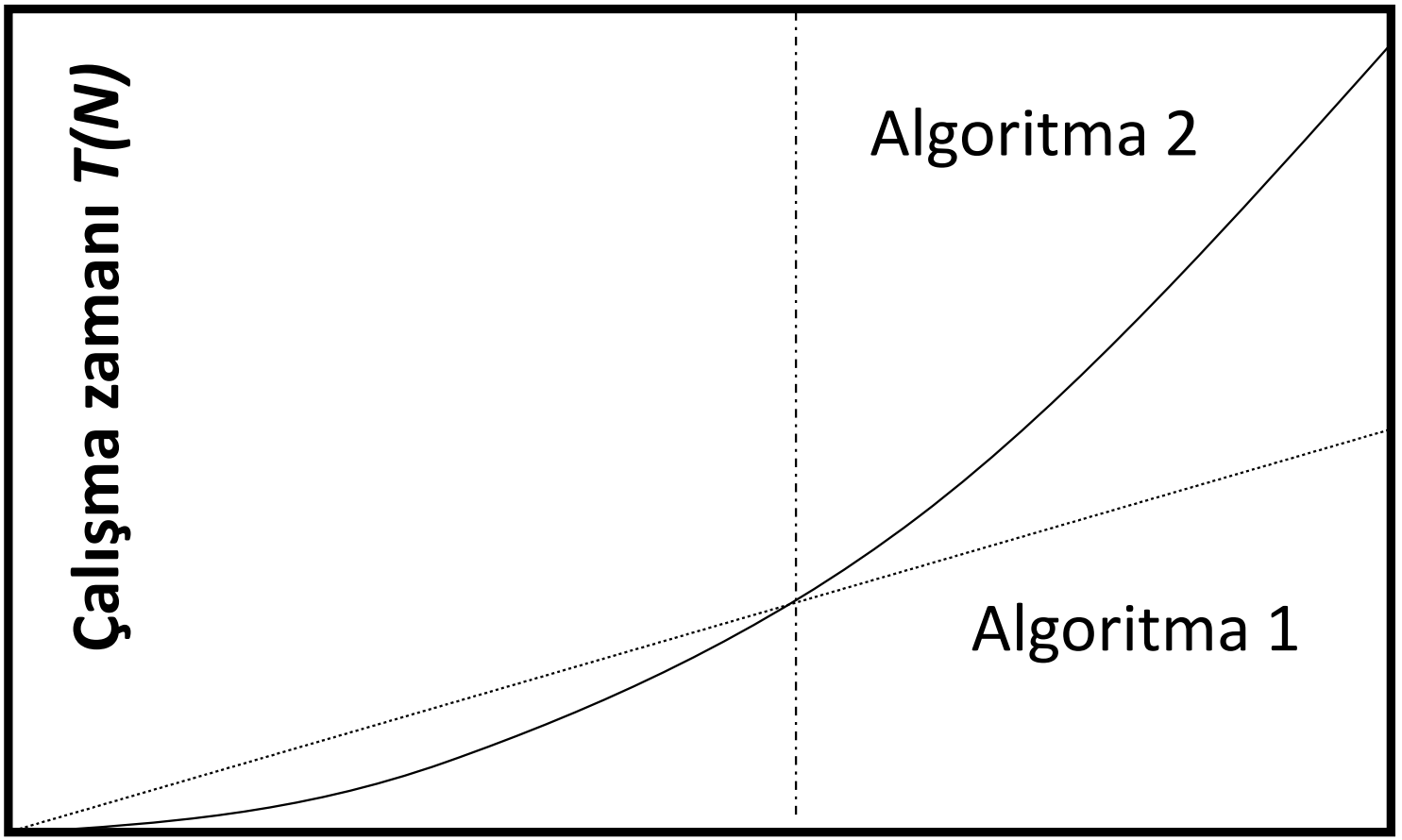
Genel Öngörü Kuralları

- **Döngü:** Döngü işletim süresi, azami olarak, iterasyon sayısı kere döngü içerisindeki komutların toplam işletim süresidir.
- **İç içe döngü:** İç içe döngü işletim süresi, en içte kalan döngü bloğundaki komutların toplam işletim süresinin tüm döngü iterasyon sayısı çarpımı ile bulunur.
- **Art arda Komutlar:** Basitçe her bir komutun işletim süreleri toplamı ile elde edilir.
- **If/Else: if** bloğundaki ve **else** bloğundaki işlemlerin daha çok zaman gerektirene, karşılaştırma için gereken işlem süresinin eklenmesiyle elde edilen toplamdan fazla olamaz.

Algoritmada Büyüme Oranları

- Algoritma zaman gereksinimi, problemin boyutunun bir fonksiyonu olarak ölçülür.
  - Problem boyutu uygulamaya bağımlıdır: Sıralanacak eleman sayısı, arama yapılacak listedeki kullanıcı sayısı, vb..
- Problem boyutu n ise, örneğin;
  - A Algoritması $n^2/5$ birim zaman gerektirir.
  - B Algoritması 5*n birim zaman gerektirir.
- Anlaşılması gereken en önemli husus, problem boyutuna göre algoritmanın zaman gereksiniminin ne kadar hızlı arttığıdır.
  - A Algoritması n^2 ile orantılı zaman gerektirir.
  - B Algoritması n ile orantılı zaman gerektirir.
- Algoritmanın oransal zaman gereksinimi **büyüme oranı** olarak adlandırılır.
- Algoritmaların başarımları büyüme oranlarına bakılarak karşılaştırılabilir.

Algoritmada Büyüme Oranları



Algoritma Büyüme Oranları

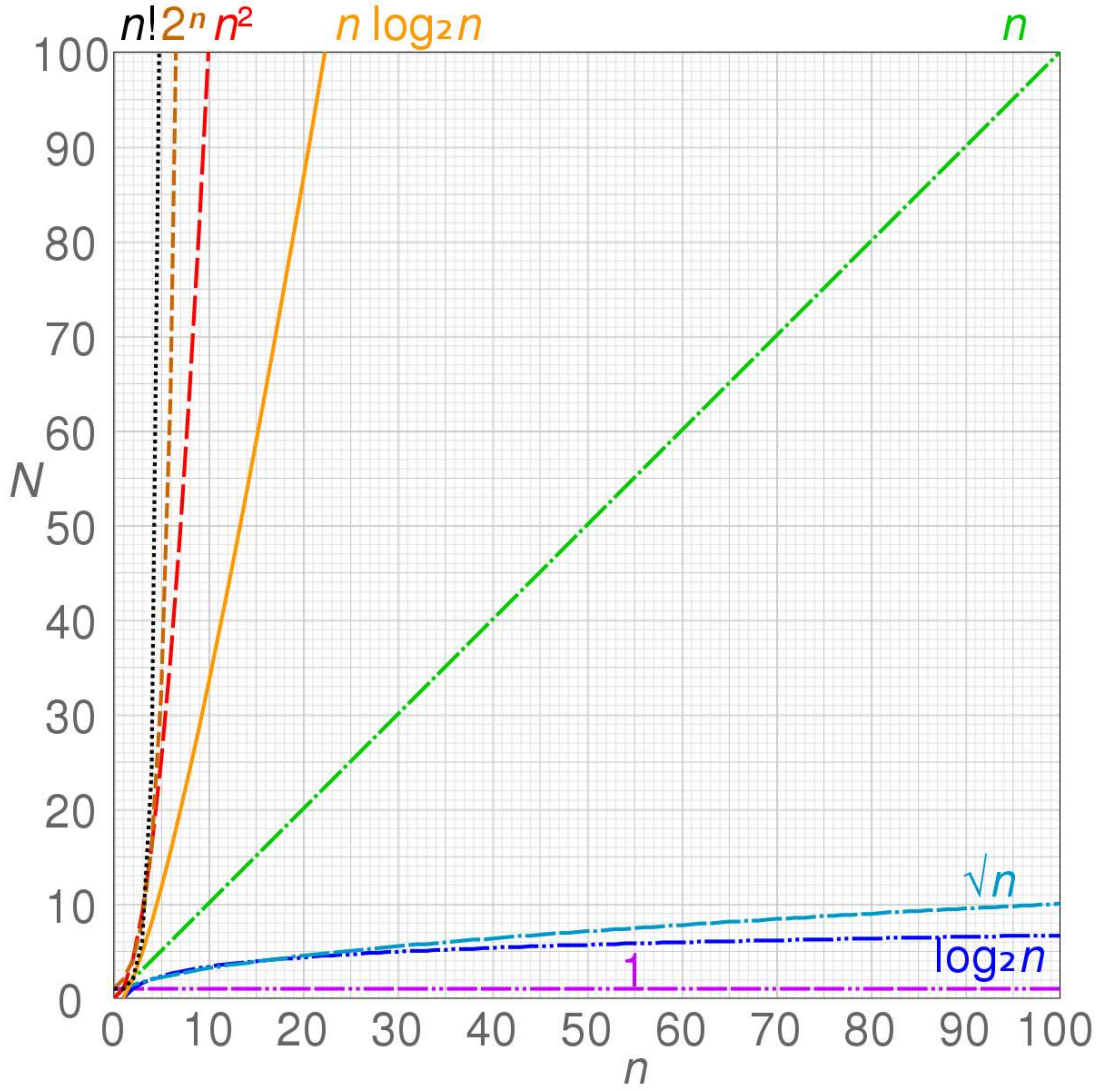
- N < 1000 için iki algoritma arasındaki yürütme zamanı ihmal edilebilir.

N	T1	T2
10	10^{-2} sn.	10^{-4} sn.
100	10^{-1} sn.	10^{-2} sn.
1000	1 sn.	1 sn.
10000	10 sn.	100 sn.
100000	100 sn.	10000 sn.

Yaygın Büyüme Oranları

Fonksiyon	Büyüme oranı İsmi
C	Sabit
log N	Logaritmik
$\log^2 N$	Logaritmik kare
N	Doğrusal
N log N	Doğrusal Logaritmik
N2	İkinci derece (Kare)
N3	Kübik
2N	Üstel
N!	Faktöryel

Yaygın Büyüme Oranları



Döngülerin Büyüme Oranları

- `for(i=0; i<1000; i++)`
  - f(n)=n (Doğrusal)

- `for(i=0; i<1000; i+=2)`
  - f(n)=n/2 (Doğrusal)

- `for(i=1; i<=1000; i*=2)`
  - f(n)=log n (Logaritmik)

Döngülerin Büyüme Oranları

- `for(i=0; i<1000; i++)`
- `    for(j=0; j<1000; j++)`
  - f(n)=n^2 (Quadratik)

- `for(i=0; i<1000; i++)`
- `    for(j=0; j<i; j++)`
  - f(n)=n((n+1)/2) (Bağımlı Quadratik)

- `for(i=0; i<1000; i++)`
- `    for(j=1; j<=1000; j*=2)`
  - f(n)=n logN (Doğrusal Logaritmik)

Fonksiyon Büyüme Analizi ve ‘Büyük O’ Gösterimi

- Bir algoritma g(n) ile doğru orantılı zaman gerektiriyorsa, g(n) seviyesinde olduğu söylenir ve **O(g(n))** şeklinde gösterilir.
- g(n) fonksiyonu algoritmanın büyüme oranı fonksiyonu olarak adlandırılır.
- Gösterim için **O** (büyük harf) kullanıldığından **Büyük O** (**Big O**) gösterimi olarak adlandırılır.
  - A Algoritması n^2 ile doğru orantılı zaman gerektiriyor ise $O(n^2)$.
  - A Algoritması n ile doğru orantılı zaman gerektiriyor ise O(n).

Büyüme Oranı Fonksiyonları

- **O(1)** Zaman gereksinimi sabittir ve problem boyutundan bağımsızdır.
- **O($\log_2 n$)** Logaritmik algoritmaların zaman gereksinimi, boyut artışından daha yavaş artış.
- **O(n)** Doğrusal algoritma zaman gereksinimi, boyut artışına paralel artış.
- **O($n*\log_2 n$)** **$n*\log_2 n$** algoritmanın zaman gereksinimi, doğrusal algoritmadan çok daha hızlı artar.
- **O(n^2)** İkinci dereceden bir algoritmanın zaman gereksinimi, problem boyutuna göre süratle artar.
- **O(n^3)** Üçüncü dereceden bir algoritmanın zaman gereksinimi, problem boyutuna göre ikinci dereceden bir algoritmanın zaman gereksiniminden çok daha yüksek süratle artar.
- **O(2^n)** Üssel bir algoritmanın zaman gereksinimi, algoritmanın kullanışlı olmasının önüne geçecek şekilde aşırı yüksek süratle artar.

Büyüme Oranlarının Karşılaştırılması (N=10.000)

Büyüme Oranı	Big-O	Iterasyon Sayısı	Tahmini süre
Logaritmik	O(logN)	13	mikrosaniyeler
Doğrusal	O(N)	10.000	saniyeler
Doğrusal Logaritmik	O(NlogN)	140.000	saniyeler
Quadratik	$O(N^2)$	10.000^2	dakikalar
Polinomiyal	$O(N^k)$	10.000^k	saatler
Üssel	$O(c^N)$	$2^{10.000}$	Felaket :)
Faktoryel	O(N!)	10.000!	Ölümcül :)

Büyüme Oranlarının Karşılaştırılması (N=10.000)

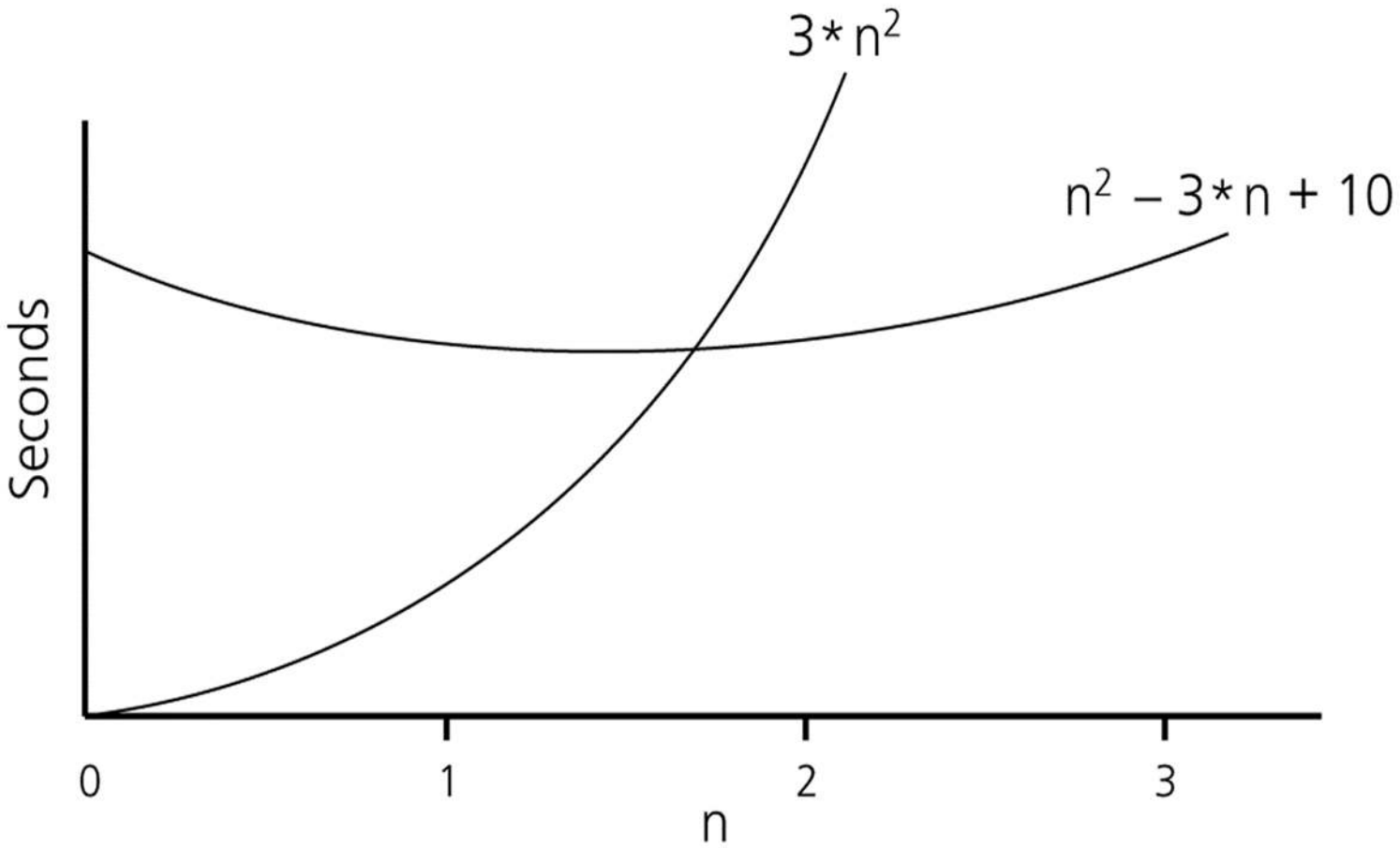
(a)

	n					
Function	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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Algoritma Seviyesi

- Bir algoritma **n** boyutundaki bir problemi çözmek için $f(n)=n^2–3*n+10$ saniye gerektiriyor.
- $k*n^2 > n^2–3*n+10$ (her $n \geq n0$ için) olacak şekilde
- **k** ve $\mathbf{n_0}$ sabitleri bulunabiliyorsa, algoritma $\mathbf{n^2}$ seviyesindedir.
- $k = 3$ ve $n_0 = 2$ ile;
- $3*n^2 > n^2–3*n+10$ ($n \geq 2$ için) .
- Dolayısıyla, algoritma $n \geq n_0$ olduğunda $k*n^2$ birim zamandan fazlasına gerek duymaz ve seviyesi $\mathbf{O(n^2)}$ olur.

Algoritma Seviyesi



Algoritma Seviyesi Tanımı

- A algoritmasının,
- **n** ≥ **n_0** boyutundaki problemi çözmek için
- asgari **k*g(n)** birim zamana ihtiyaç duyacağı şekilde
- **k** ve **n_0** sabitleri bulunabiliyorsa,

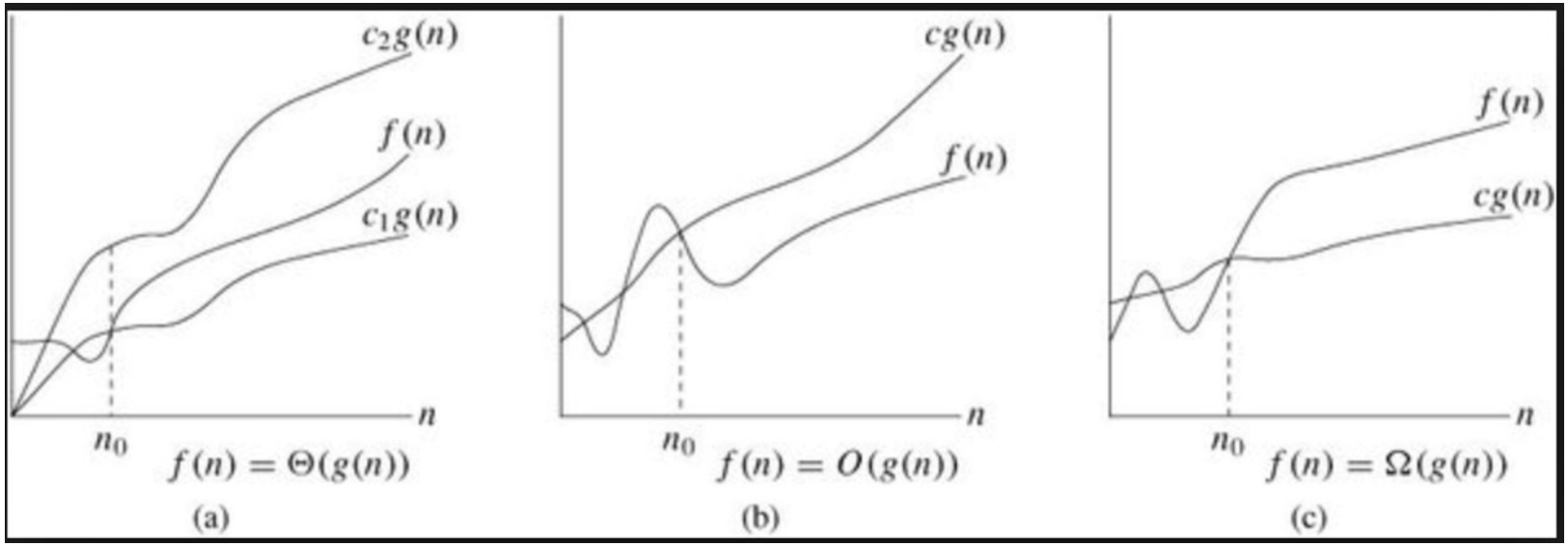
- A'nın omega **g(n)** seviyesinde olduğu söylenir ve
- **Ω (g(n))** olarak gösterilir.
  - f(n) ≥ k*g(n); her n ≥ n0 için

Algoritma Seviyesi Tanımı

- A algoritmasının,
- **$n \geq n_0$** boyutundaki problemi çözmek için
- asgari **$k_1 * g(n)$** ve
- azami **$k_2 * g(n)$** birim zamana ihtiyaç duyacağı şekilde
- **k_1, k_2** ve **n_0** sabitleri bulunabiliyorsa,

- A'nın teta **g(n)** seviyesinde olduğu söylenir ve
- **Θ (g(n))** olarak gösterilir.
  - $k_2 * g(n) \geq f(n) \geq k_1 * g(n)$; her $n \geq n_0$ için

Algoritma Seviyesi Tanımı



- O fonksiyonu üstten, Ω alttan bağlarken, **θ** hem alttan hem üstten bağlamaktadır. Başka açıdan:
- **θ**(n): O(n) **≥** f(n) **≥** Ω(n) olur.
- ≥ yerine > konursa sırasıyla o, ve ω gösterimleri elde edilir.

Büyük-O Özellikleri

1. Düşük seviyeli terimler göz ardı edilebilir.
   - Bir algoritma $O(n^3+4n^2+3n)$ ise aynı zamanda $O(n^3)$'dür.
   - Büyüme oranı fonksiyonu olarak **sadece en yüksek seviyeli terim** kullanılır.
2. Sabit bir çarpan göz ardı edilebilir.
   - Bir algoritma $O(5n^3)$ ise aynı zamanda $O(n^3)$'dür.
3. Büyüme oranı fonksiyonları birleştirilebilir.
   - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
   - Bir algoritma $O(n^3) + O(4n^2)$ ise aynı zamanda $O(n^3 +4n^2)$ olur
   - ➔ $O(n^3)$.
   - Çarpma için de aynı kural geçerlidir.

Büyük-O Örnek-1

	Maliyet	Tekrar
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  i = i + 1;`	c4	n
`  sum = sum + i;`	c5	n
`}`		

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

= (c3+c4+c5)*n + (c1+c2+c3)

= a*n + b

➔ Algoritmanın büyüme oranı fonksiyonu (karmaşıklığı) **O(n)** olur.

Büyük-O Örnek-2

```
i=1;                                  c1      1
sum = 0;                              c2      1
while (i <= n) {                      c3      n+1
          j=1;                        c4      n
          while (j <= n) {            c5      n*(n+1)
               sum = sum + i;         c6      n*n
               j = j + 1;             c7      n*n
     }
     i = i + 1;                       c8      n
}
```

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5 + n*n*c6 + n*n*c7 + n*c8

= $(c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$

= $a*n^2 + b*n + c$

➔ Algoritmanın büyüme oranı fonksiyonu (karmaşıklığı) **$O(n^2)$** olur.

Büyük-O Örnek-3

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

T(n) = $\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1$

Algoritmanın büyüme oranı fonksiyonu **O(n^3)** olur.

Özyinelemeli Algoritmalarda Büyük-O

- Özyineli fonksiyonların zaman karmaşıklık fonksiyonu T(n), kendisi cinsinden ifade edilir ve T(n) yineleme denklemi (recurrence equation) olarak adlandırılır.
- Özyineli bir fonksiyonun büyüme oranı fonksiyonunu çözebilmek için yineleme ilişkisini çözmek gerekir.

- Örnek: Fibonacci Sayıları

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

Özyinelemeli Algoritmalarda Büyük-O

- n=0 T(0) = c1
- n=1 T(1) = c2
- n>1 ise T(n)=c1+c2+T(n-1)+T(n-2) ➔ **T(n-1)+T(n-2)+c**
- Yineleme ilişkisi denklemini çözmek gereklidir.

- T(n-1) ~ T(n-2) kabul edebiliriz
  - aslında T(n-1) ≥ T(n-2) ama üstten sınırlayabiliriz (upper bound)

Özyinelemeli Algoritmalarda Büyük-O

$T(n) = T(n-1) + T(n-2) + c$

$= 2T(n-1) + c$ // $T(n-1) \sim T(n-2)$ yaklaşımı ile

$= 2*(2T(n-2) + c) + c$

$= 4T(n-2) + 3c$

$= 8T(n-3) + 7c$

$= 2^k * T(n-k) + (2^k-1)*c$

$n-k=0$, **$k=n$**

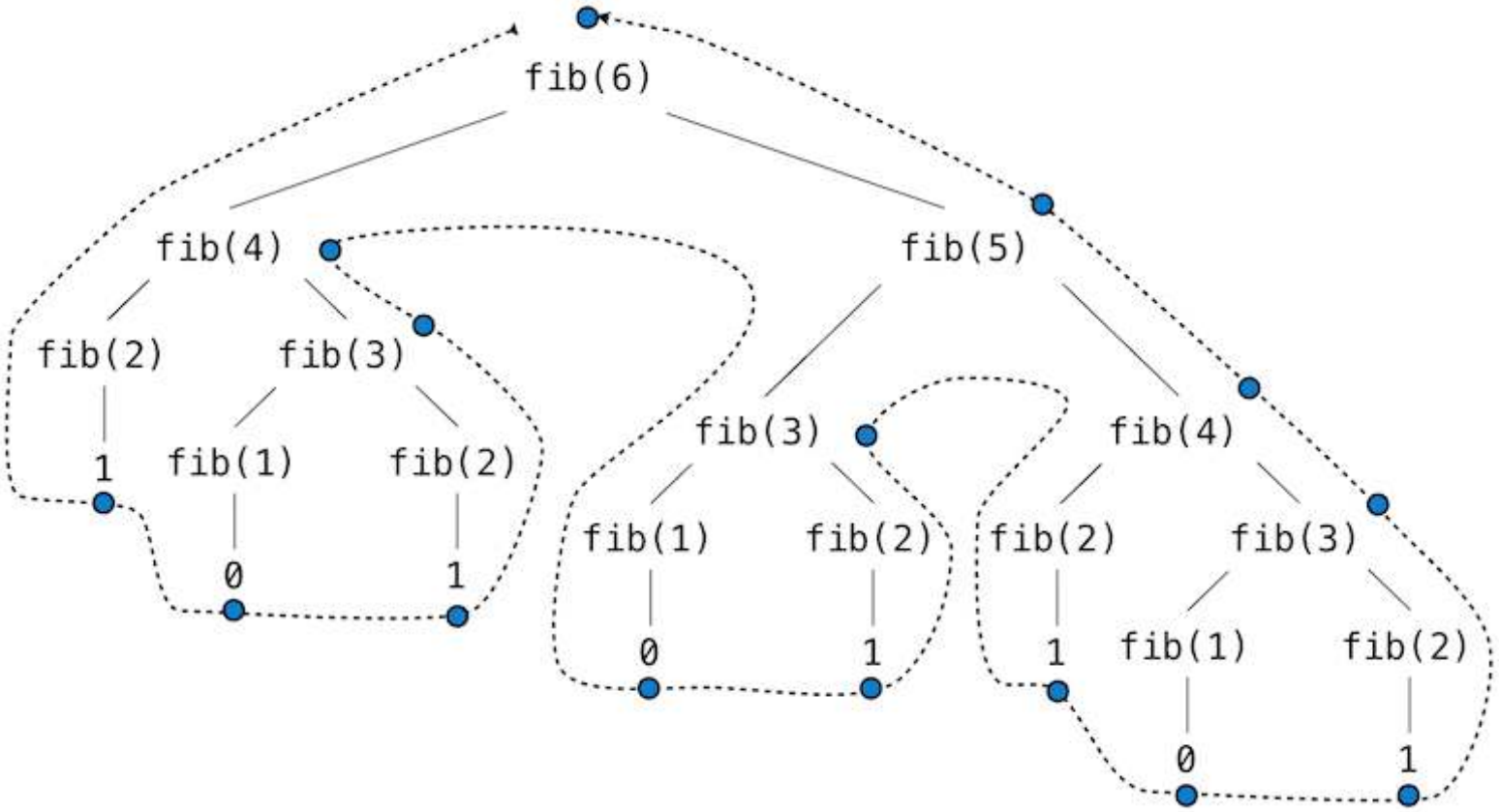
$T(n) = 2^n * T(0) + (2^n - 1) * c$

$= 2^n * (1 + c) - c$

$T(n) \sim 2^n$

- ➔ Algoritmanın büyüme oranı fonksiyonu **$O(2^n)$** olur.

Özyinelemeli Algoritmalarda Büyük-O



Değerlendirme

- Bir algoritma aynı boyuttaki farklı sorunları çözmek için farklı sürelere gereksinim duyabilir.
  - n elemanlı bir dizide, sıradan arama yapmanın maliyeti. ➔ 1, 2, ..., n
- ***En kötü durum (Worst-Case) Analizi*** – Algoritmanın ***n*** boyutundaki problemi çözmesi için gereken azami süre.
  - Algoritma karmaşıklığı için üst sınır verir.
  - Genel olarak algoritmaların en kötü durum davranışı incelenir.
- ***En iyi durum (Best-Case) Analizi*** – Algoritmanın ***n*** boyutundaki problemi çözmesi için gereken asgari süre.
  - Çok kullanışlı değildir.
- ***Ortalama durum (Average-Case) Analizi*** – Algoritmanın ***n*** boyutundaki problemi çözmesi için gereken ortalama süre.
  - Çoğunlukla belirlenmesi zordur.
  - Olası veri durumu değerlendirilip, dağılımına bakılır.
  - ***En kötü durum analizi, ortalama durum analizinden daha çok kullanılır.***

Sıralı Arama

```
int sequentialSearch(int a[], int size, int x) {
   int i;

   for (i=0; i<size && a[i]!=x; i++);

   if (i==size)
      return -1;

   return i;
```

- Bulunamazsa **O(n)**
- Bulunursa,
  - En iyi durum: aranan değer ilk eleman, **O(1)**
  - En kötü durum: aranan değer son elemandır **O(n)**
  - Ortalama durum: karşılaştırma sayısı 1, 2, …, n $(\sum_{i=1}^{n} n)/n$ **O(n)**

İkili Arama

- Dizi sıralı ise, ikili arama kullanılabilir.
- Aranılan eleman dizide yoksa, döngü içerisindeki yineleme sayısı $\lfloor \log_2 n \rfloor + 1$

  ➔ $O(\log_2 n)$
- Başarılı arama :

  En iyi durum: *aranan değer dizinin bakılan ilk elemanıdır* ➔$O(1)$

  En kötü durum: *aranan değer son bakılacak konumdadır* $\lfloor \log_2 n \rfloor + 1$

  ➔ $O(\log_2 n)$

  Ortalama durum : karşılaştırma sayısı $< \log_2 n$ ➔ $O(\log_2 n)$

0	1	2	3	4	5	6	← 7 elemanlı dizi
3	2	3	1	3	2	3	← yineleme sayısı

Ortalama yineleme sayısı : = 17/7 = 2.4285 < $\log_2 7$

O(logN) Performansı

n	***O($log_2 n$)***
16	4
64	6
256	8
1024	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576	20
1,073,741,824	30

Binary Search

```
int binarySearch(int a[], int size, int x) {
    int low =0;
    int high = size -1;
    int mid; // aranan değer bulunursa mid konumu tutar.

    while (low <= high) {
      mid = (low + high)/2;

      if (a[mid] < x)
         low = mid + 1;
      else if (a[mid] > x)
         high = mid - 1;
      else
          return mid;
    }
    return -1;
}
```