VERİ YAPILARI DERS NOTLARI

BÖLÜM 2 – ALGORİTMA ANALİZİ

Yard. Doç. Dr. Deniz KILINÇ

CELAL BAYAR ÜNİVERSİTESİ, YAZILIM MÜHENDİSLİĞİ

2015-2016

1. ALGORİTMA TANIMI

Verilen herhangi bir sorunun çözümüne ulaşmak için uygulanması gerekli adımların hiç bir yoruma yer vermeksizin; basit, açık, düzenli ve sıralı bir şekilde söz ve yazı ile ifadesidir. Algoritmalar üzerinde makine ve dil bağımsız çalışılabildiği için bilgisayar bilimlerindeki en önemli konulardan bir tanesidir. Algoritmaların sahip olması gereken özellikler aşağıdaki gibidir:

- **Giriş/çıkış bilgisi:** Algoritmalarda giriş ve çıkış bilgileri olmalıdır. Dışarıdan gelen verilere giriş bilgisi denir. Bu veriler algoritmada işlenir ve çıkış bilgisini oluşturur. Çıkış bilgisi her algoritmada mutlaka vardır. Algoritmaların temel amacı giriş bilgisini işleyerek çıkış bilgisi oluşturmaktır.
- **Sonluluk:** Her türlü olasılık için algoritma sonlu adımda bitmelidir. Algoritma sonsuz döngüye girmemelidir.
- **Kesinlik:** Her komut, kişinin kalem ve kağıt ile yürütebileceği kadar basit olmalıdır. Algoritmanın her adımı anlaşılır, basit ve kesin bir biçimde ifade edilmiş olmalıdır. Kesinlikle yorum gerektirmemeli ve belirsiz ifadelere sahip olmamalıdır.
- **Etkinlik:** Yazılan algoritmalar etkin ve dolayısıyla gereksiz tekrarlardan uzak oluşturulmalıdır. Bu algoritmanın temel özelliklerinden birisidir. Ayrıca algoritmalar genel amaçlı yazılıp yapısal bir ana algoritma ve alt algoritmalardan oluşturulmalıdır. Böylece daha önce yazılmış bir algoritma daha sonra başka işlemler için de kullanılabilir.
- **Başarım ve performans:** Amaç donanım gereksinimi (işlemci, bellekvb.), çalışma süresi gibi performans kriterlerini dikkate alarak yüksek başarımlı programlar yazmak olmalıdır. Gereksiz tekrarlar ortadan kaldırılmalıdır. *Algoritmanın başarısını nasıl ölçeriz?*

2. ALGORİTMALARIN ANALİZİ

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.

Çalışma zamanı analizi (***karmaşıklık analizi***) *bir algoritmanın (artan) "(veri) giriş" boyutuna bağlı olarak işleme zamanının 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/karmaşıklık analizi için kullanılabilecek başlıca yöntemlere aşağıdaki gibidir:

2.1. Deneysel Analiz Yöntemi

Deneysel analiz, örnek problemlerde denenmiş bir algoritmada hesaplama deneyimine dayanır. Bu analizin amacı, pratikte algoritmanın nasıl davrandığını tahmin etmektir. Bu analizde, algoritma için bir bilgisayar programı yazılır ve problem örneklerinin bazı sınıfları üzerinde programın performansı test edilir. Bilimsel yaklaşımdan çok, uygulamaya yöneliktir. Deneysel analizin başlıca dezavantajları şunlardır:

i. Bir algoritmanın performansının, programı yazan programcının tekniği kadar kullanılan bilgisayara, derleyiciye ve programlama diline bağlı olması.
ii. Bu analizin çok zaman alması ve düzenlenmesinin pahalıya mal olması.

2.2. RAM (Random Access Machine) Modeli ile Komut Sayarak Çalışma Zamanı Analiz Yöntemi:

Her basit operasyon (+,-, * =, if, call) "bir" zaman biriminde gerçekleşir. Döngüler ve alt rutinler (fonksiyonlar) basit operasyonlar ile aynı şekilde değerlendirilmezler. RAM modelinde her bellek erişimi yine "bir" zaman biriminde gerçekleşir. RAM modeli bilgisayarların algoritmaları gerçekleştirimini ölçmek için kullanılan basit bir yöntemdir. Genel olarak çalışma zamanı veri giriş boyutu n'e bağlı $T(n)$ ile ifade edilir.

RAM'in en büyük dezavantajı algoritmanın çok detaylı bir şekilde implemente edilmesi gerekliliğidir. Ayrıca bu durum programlama diline ve programcıya göre değişkenlik gösterir. Örneğin; "case kullanımı" veya "iç-içe if kullanımı".

Örnek 1: Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N) {
  int toplam = 0;               → 1
  for  (i=0; i < N; i++){       → N
    toplam += A[i];             → N
  } //Bitti-for
  return toplam;                → 1
} //Bitti-Topla
```

Toplam zaman = 1 + N + N + 1 = 2N + 2
T(N) = 2N + 2

Örnek 2: Dizideki bir elemanın aranması

```
int Arama(int A[], int N, int sayi) {
  int i = 0;                    → 1
  while (i < N){                → N
    if (A[i] == sayi) break;    → N
    i++;                        → N
  } //bitti-while

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

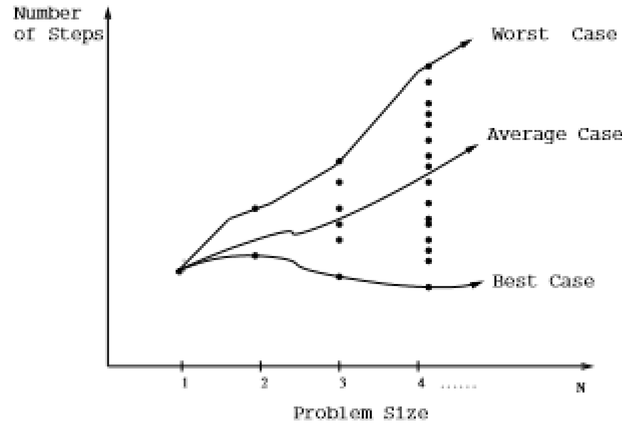
Toplam zaman = 1 + N + N + N + 1 + 1= 3N + 3
T(N) = 3N + 3

Algoritma Analiz Türleri:

Bir algoritmanın analizi için o algoritmanın kabaca <u>bir polinom</u> veya diğer <u>zaman karmaşıklıkları cinsinden</u> 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:

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

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



<u>Örnek 2'den</u> yola çıkarsak en kötü, en iyi ve ortalama algoritma çalışma zamanlarını aşağıdaki gibi bulabiliriz:

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

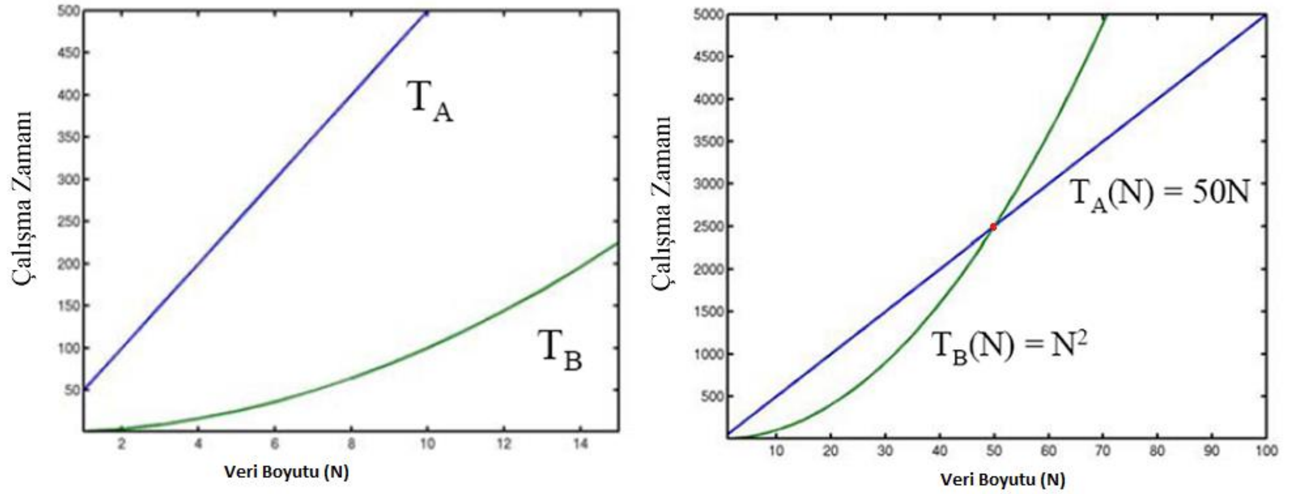
2.3. Asimptotik Analiz ve Notasyon

Genel bir tanım yapacak olursak, asimptotik notasyon, eleman sayısı <u>n'nin sonsuza gitmesi</u> durumunda algoritmanın, benzer işi yapan algoritmalarla karşılaştırmak için kullanılır. Eleman sayısının küçük olduğu durumlar pratikte mümkün olabilir fakat bu birçok uygulama için geçerli değildir. Verilen iki algoritmanın çalışma zamanını T1(N) ve T2(N) fonksiyonları

şeklinde gösterilir. Ancak hangisinin daha iyi olduğunu belirlemek için bir yol belirlememiz gerekiyor.

Örneğin: aşağıdaki 2 diyagramı da değerlendirdiğinizde $T_A(N)$ ve $T_B(N)$ fonksiyonlarına sahip hangi algoritmayı seçeriz?

Veri boyutu azken ikinci algoritma (T_B) daha iyi çalışıyor. N = 50 olduğunda her iki algoritmanın çalışma zamanları eşitleniyor. N>50 için ilk algoritma doğrusal ilerlerken ikinci algoritma karesel ilerliyor.



Hangi algoritmanın daha iyi olduğunu belirlemek sıkça kullanılan asimptotik notasyonlar aşağıdaki gibidir:

- **Big-Oh (Big O):** Asimptotik üst sınır
- **Big Ω :** Asimptotik alt sınır
- **Big Θ:** Asimptotik alt ve üst sınır

2.4. Big-O Notasyonu

Algoritmanın *f(n)* şeklinde ifade edildiğini varsayalım. Algoritma, fonksiyonunun sıkı üst sınırı (tight upper bound) olarak tanımlanan ve performans ölçümünde en çok kullanılan Big-O notasyonunu inceleyelim:

Bir fonksiyonun sıkı üst sınırı genel olarak:

$f(n) = O(g(n))$

şeklinde ifade edilir. Bu ifade n'nin artan değerlerinde *f(n)*'nin *üst sınırı g(n)*'dir, şeklinde yorumlanır.

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**

$g(n) = n^4$

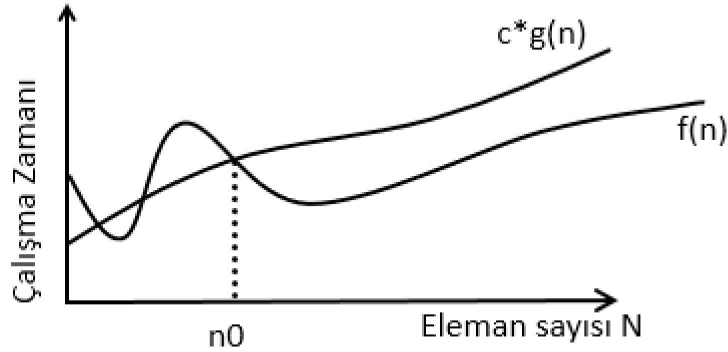
olarak elde edilir.

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

O-notasyonunu matematiksel olarak daha açık yazarsak;

O(g(n)) = { f(n): tüm $n>=n_0$ için, $0 <= f(n) <= cg(n)$ olmak üzere pozitif c ve n_0 sabitleri bulunsun}.

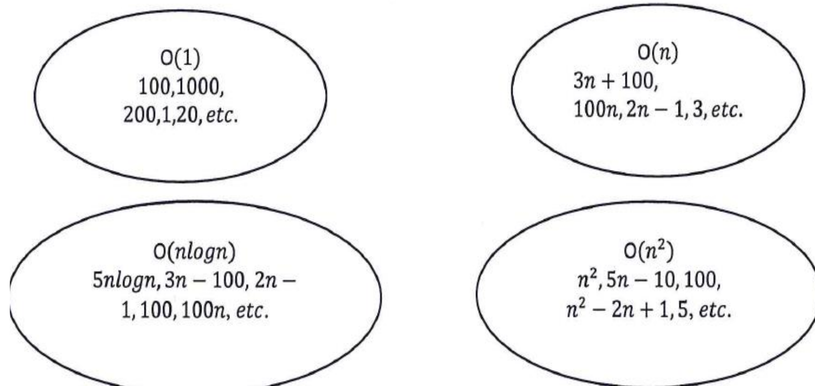
- Bu durumda g(n), f(n)'nin asimptotik (n sonsuza giderken, artarken!) sıkı üst sınırı olur.
- Genel olarak n'nin düşük değerleri ve o değerlerdeki değişim dikkate alınmazken, n_o'dan büyük değerler için algoritmanın büyüme oranı değerlendirilir.
- Bu durum aşağıdaki şekilde gösterilmiştir.



Şekil. Girişteki değişime bağlı olarak cg(n) ve f(n)'in değişimi

Dikkat edilirse, n_o'dan büyük değerler için cg(n), f(n) için üst sınırı (asimptot gibi!) olarak görülürken, n_o öncesinde iki fonksiyonun değişimi farklı olabilir.

Bu yaklaşımla, O(g(n)) aynı veya daha düşük büyüme oranına sahip bir grup fonksiyona işaret eder. Örneğin, $O(n^2)$ fonksiyonu O(1), O(n) veya O(nlogn) fonksiyonlarını da içerir (onların üst sınırı olur!).



Şekil. Big-O fonksiyonları ve örnek alt fonksiyonlar

2.4.1. Algoritmalar İçin "Rate of Growth" (-Zaman- Büyüme Oranı)

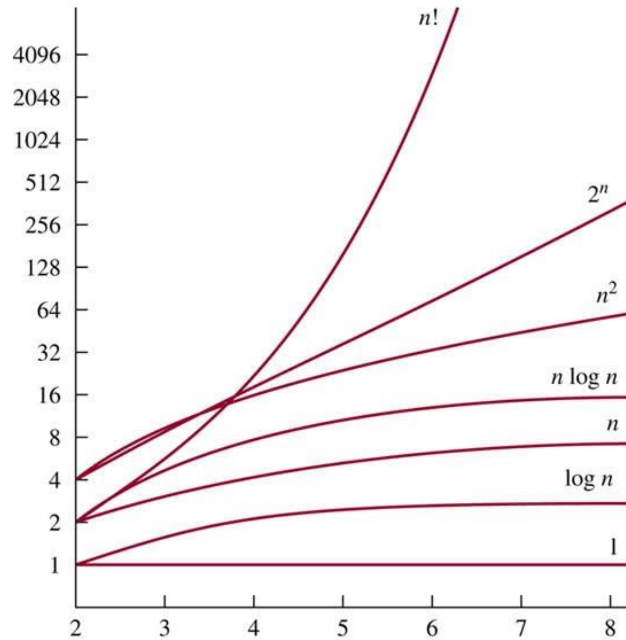
Girişin bir fonksiyonu olarak çalışma zamanının artışı ***büyüme oranı*** *olarak tanımlanır.* Bu tanımda incelenen fonksiyon büyüyen n değerleri için genellikle başka bir değere yakınsanarak ele alınır.

Örneğin: $n^4 + 2n^2 + 100n + 500$ gibi bir fonksiyonun n'e bağlı değişimi, büyük n değerleri için kabaca n^4 olur.

Literatürde sıklıkla karşılaşılan "Büyüme Oranı" fonksiyonları aşağıdaki tabloda verilmiştir.

Zaman karmaşıklığı		**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-yut 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

Tablo. Büyüme Oranı İçin Sık Karşılaşılan Fonksiyonlar



Şekil. Büyüme oranı gösterimi

2.4.2. Big-O Analiz Kuralları

n'in artan değerleri için, n'e bağlı şekilde modellenen algoritmaların analizi-karşılaştırılması için bir grup kural tanımlanmıştır:

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.

2.4.3. Big-O Yardımıyla Çalışma Zamanı Kestirimi Kuralları

Kodların Big-O yaklaşımıyla analizi için aşağıdaki kurallardan faydalanırız:

i) Loop'lar: Bir döngünün çalışma zamanı döngü sayısının bir c sabitiyle çarpımı kadardır.

```
//döngü n defa çalışır
for (i=1;i<=n;i++)
      m = m+2; // c sabiti
```

Toplam zaman = c x n = cn = O(n) olur.

Not: Eğer bir döngünün n değeri sabit verilmişse. Örneğin: n = 100 ise değeri O(1)'dir.

ii) İç-içe loop'lar: *İçten dışa doğru* her loop'un çalışma sayısı dış loop'la çarpılırken, sabit değerler de çarpıma dahil edilir.

```
// dış-loop n defa çalışır
for (i=1;i<=n;i++)
{
      // iç loop n defa çalışır
      for (j=1;j<=n;j++)
            k=k+1; // sabit c zamanı
}
```

Toplam zaman = n.c (iç-döngü) x n(dış döngü) = $c.n^2 = O(n^2)$

iii)Ardışık ifadeler: Her bir ifadenin zaman karmaşıklığı eklenerek tek bir fonksiyon elde edilir.

```
x = x+1; //sabit zaman

//döngü n defa çalışır
for (i=1;i<=n;i++)
```

```
        m=m+2; // c sabiti

// dış-loop n defa çalışır
for (i=1;i<=n;i++)
{
        // iç loop n defa çalışır
        for (j=1;j<=n;j++)
                k=k+1; // sabit c zamanı
}
```

Toplam zaman = $c_0 + c_1n + c_2n^2 = O(n^2)$

iv) if (then) else ifadesi: En-kötü çalışma zamanı mantığıyla, "test bölümü", "then" veya "else" bölümlerinden çalışma zamanı büyük olanla toplanır.

```
//test sabit1
if(lenght()==0)
{
        return false; // then bölümü sabit2
}
else
{
// else bölümü (sabit3+sabit4) x n
        for (int n=0;n<length();n++)
        {
                // BURADA İKİ ATAMA VARSAYILIYOR… SABİT 3 VE 4 İÇİN..
        }
}
```

Toplam zaman = test + (then veya else'ten büyük olan): c0 + (c3+c4) x n = O(n)

v) Logaritmik karmaşıklık: Bir algoritma, problemin çözüm kümesini belli sabit oranında bölüyorsa (örneğin ½ oranında!) O(logn) zamanına ihtiyaç duyar.

```
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!)

Toplam zaman = O(logn)'dir.

Örneğin: Binary search 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 diğer örnek algoritmalar aşağıdaki gibidir:

- Sıralama: Merge Sort and Quick Sort
- Tree gezinme

vi) Diğer kurallar: İlk 5 kuralın altında incelenebilirler örneğin yinelemeli (recursive) fonksiyonlarda temelde bir for döngüsü işletilir.

2.5. Big-Oh Avantajları (Asimptotik yaklaşım sayesinde…)

Big-Oh notasyonunun başlıca avantajları aşağıdaki gibidir:

- 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*.

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:

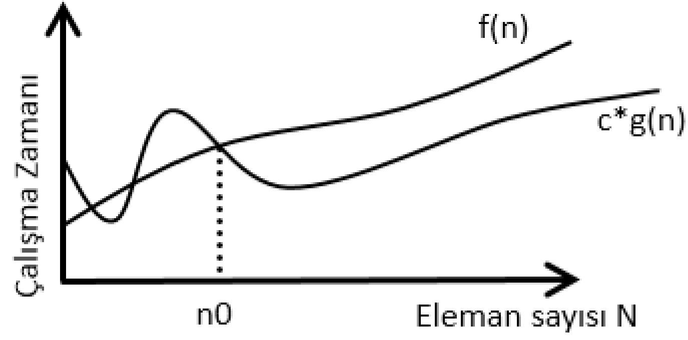
$$4n^2 - 3n \log n + 17.5\,n - 43\,n^{2/3} + 75 \rightarrow n^2 - n \log n + n - n^{2/3} + 1 \rightarrow n^2 \rightarrow O(n2)$$

Çalışma: Aşağıdaki fonksiyonların karmaşıklıklarını Big O notasyonunda gösteriniz.

- $f1(n) = 10\,n + 25\,n^2$
- $f2(n) = 20\,n \log n + 5\,n$
- $f3(n) = 12\,n \log n + 0.05\,n^2$
- $f4(n) = n^{1/2} + 3\,n \log n$

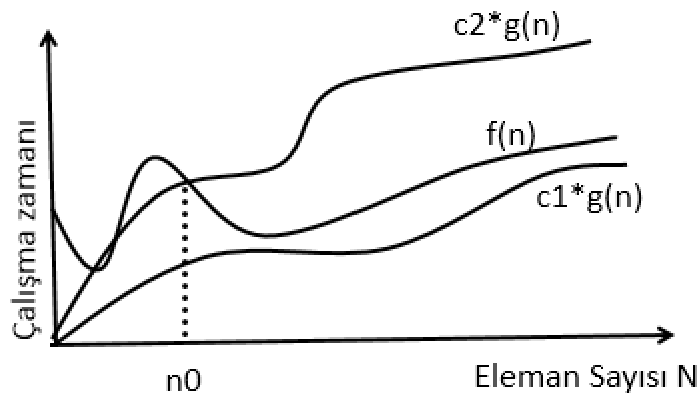
3. Big Ω (Omega) Notasyonu

$O(g(n)) = \{ f(n)$: tüm $n >= n_0$ için, $0 <= cg(n) <= f(n)$ olmak üzere pozitif c ve n_0 sabitleri bulunsun$\}$. g(n), f(n)'nin asimptotik (n sonsuza giderken, artarken!) sıkı alt sınırı olur.



Şekil. Girişteki değişime bağlı olarak cg(n) ve f(n)'in değişimi

4. Big Θ (Teta) Notasyonu



Şekil. Girişteki değişime bağlı olarak cg(n) ve f(n)'in değişimi

O(g(n)) = { f(n): tüm n>=n_0 için, c2*g(n) <= f(n) <= c1*g(n)olmak üzere pozitif c1, c2, ve n_0 sabitleri bulunsun}. g(n), f(n)'nin asimptotik (n sonsuza giderken, artarken!) sıkı alt sınırı ve üst sınırı olur.

g(n), f(n)'nin asimptotik (n sonsuza giderken, artarken!) sıkı alt ve üst sınırı olur.