BM 210 Algoritma Analizi (Analysis of Algorithms)

Hazırlayan: M.Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Algoritmaların çalışma süresi
- Fonksiyonların büyümesi ve Asymptotic notasyonlar
- Algoritmaların doğruluğu

Ödev - Arama (Searching)

Önceki haftanın ödevi

- Aşağıdaki arama algoritmasının analizini yapınız

INPUT: A[1..n] – an array of integers, q – an integer.
OUTPUT: an index j such that A[j] = q. *NIL*, if $\forall j$ ($1 \le j \le n$): A[j] $\neq q$

```
j←1
while j ≤ n and A[j] ≠ q
   do j++
if j ≤ n then return j
else return NIL
```

Algoritmaların Çalışma Süresi

Insertion Sort

	cost	times
for j←2 **to** n	c_1	n
do key←A[j]	c_2	n−1
▶Insert A[j] into the sorted sequence A[1..j-1]	0	n−1
i←j−1	c_4	n−1
while i>0 **and** A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i−−	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_8	n−1

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Algoritmaların Çalışma Süresi

Insertion Sort (Best Case)

	cost	times
for j←2 **to** *n*	c_1	n
do key←A[j]	c_2	n−1
►Insert A[j] into the sorted sequence A[1..j-1]	0	n−1
i←j−1	c_4	n−1
while i>0 **and** A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i−−	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_8	n−1

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = an + b \qquad \textbf{(Linear Function)}$$

Algoritmaların Çalışma Süresi

Insertion Sort (Worst Case)

	cost	times
for j←2 **to** *n*	c_1	n
do key←A[j]	c_2	n−1
►Insert A[j] into the sorted sequence A[1..j-1]	0	n−1
i←j−1	c_4	n−1
while i>0 **and** A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i−−	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_8	n−1

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad ve \qquad \sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = an^2 + bn + c \qquad \textbf{(Quadratic Function)}$$

Computational Complexity

- Algoritmaları karşılaştırabilmek için bir algoritmanın zorluk derecesi ölçümüne "Computational Complexity" denir.
- Juris Hartmanis ve Richard E.Stearns tarafından geliştirilmiştir.
- Computational complexity bir algoritmanın gerçekleştirilmesi için gereken maliyeti veya çabayı ifade eder. Maliyet veya çaba zaman (time) ve kullanılan alan (space) ile ifade edilir.

Asymptotic Complexity

- Algoritmalarda t (süre) ve n (giriş boyutu) arasındaki ilişki çoğu zaman çok karmaşıktır.
- Fonksiyon içerisindeki önemsiz kısımlar ve katsayılar atılarak basitleştirilir ve gerçek fonksiyona göre yaklaşık bir değer bulunur.
- Elde edilen bu yeni etkinlik ölçümüne "Asymptotic Complexity" denir.
- Genellikle girişin büyümesine bağlı olarak fonksiyonun büyümesinde en büyük etkiye sahip olan parametre alınır.

Asymptotic Complexity

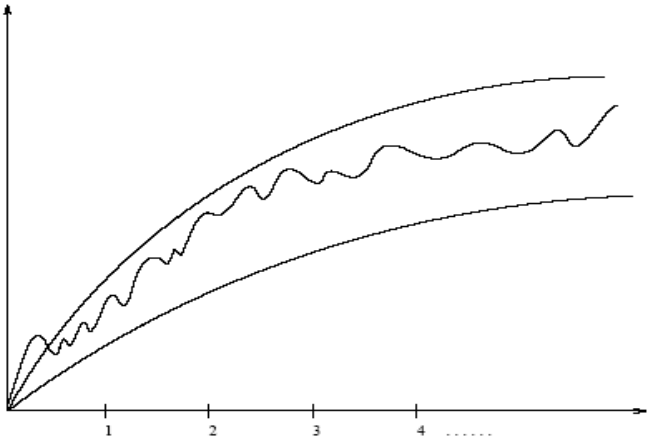
n	$t(n)=60n^2+5n+1$	$60n^2$
10	6.051	6.000
100	600.501	600.000
1.000	60.005.001	60.000.000
10.000	6.000.050.001	6.000.000.000

Asymptotic Analiz

- Amaç: detaylardan kurtularak çalışma süresi analizini basitleştirmek
  - Sayılar için "rounding" işlemi: 1,000,001 ≈ 1,000,000
  - Fonksiyonlar için "rounding" işlemi: $3n^2 \approx n^2$
- Niteliğini belirlemek (Capturing the essence): belirlenen limit içerisinde girişin boyutuna göre algoritmanın çalışma süresinin nasıl arttığının bulunması

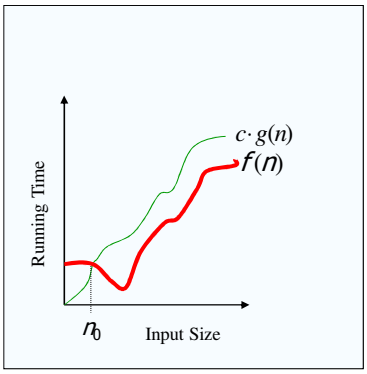
Asymptotic Analiz

- **Algoritmaların best, worst ve average case çalışma süreleri bir fonksiyonla ifade edilebilir.**
- **Ancak kesin sürenin hesaplanması oldukça zor ve karmaşıktır.**

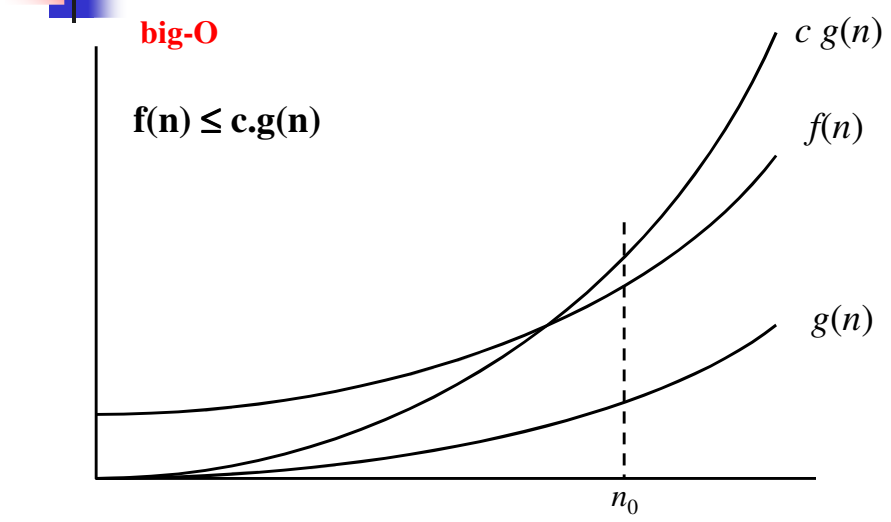


Asymptotic Notasyonlar

- "big-O" *O*-ifadesi
  - asymptotic upper bound
  - *f(n) = O(g(n)),* eğer sabit bir *c* ve n_0 *değeri için* **f(n)** ≤ **c.g(n)** bütün $n \geq n_0$ değerleri için doğruysa
  - *f(n)* ve *g(n)* pozitif değere sahip fonksiyonlardır
- *worst-case analiz için kullanılır*



Asymptotic Notasyonlar



Asymptotic Notasyonlar

Örnek:

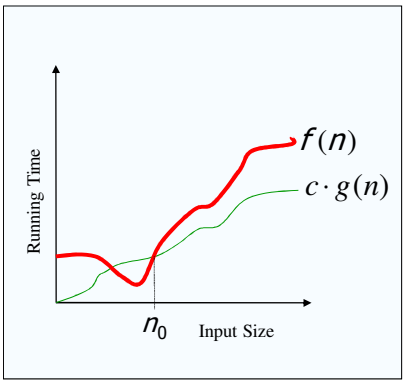
- $3n^2+2n+5 = O(n^2)$ olduğunu gösteriniz
  - $10\ n^2 = 3n^2 + 2n^2 + 5n^2$
    $\geq 3n^2 + 2n + 5\ ,\ n \geq 1$
  - $c = 10,\ n_0 = 1$

Asymptotic Notasyonlar

- O-ifadesi için genellikle en basit formül kullanılır.
  - Örnek
    - $3n^2+2n+5 = O(n^2)$
  - Aşağıdaki örneklerde doğrudur ancak genellikle kullanılmazlar
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

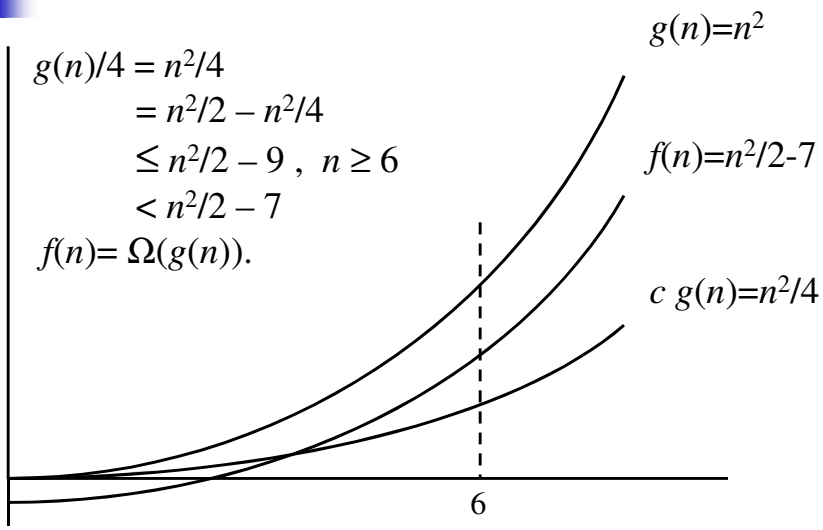
Asymptotic Notasyonlar

- The "big-Omega" Ω–İfadesi
  - asymptotic lower bound
  - *f(n) = Ω(g(n)),* eğer sabit bir c ve n_0 değeri için **c.g(n) ≤ f(n)** bütün $n \geq n_0$ değerleri için doğruysa
- *best-case* çalışma süresi veya lower bound tanımlamasında kullanılı



Asymptotic Notasyonlar

$g(n)/4 = n^2/4$
$= n^2/2 - n^2/4$
$\leq n^2/2 - 9\ ,\ n \geq 6$
$< n^2/2 - 7$
$f(n) = \Omega(g(n))$.

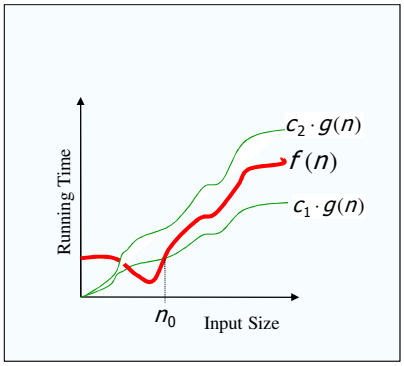


Asymptotic Notasyonlar

- Basit kural: Küçük dereceden terimler ve sabitler atılır.
  - 50 $n \log n$ ifadesi $O(n \log n)$ şeklinde gösterilir
  - $7n - 3$ ifadasi $O(n)$
  - $8n^2 \log n + 5n^2 + n$ ifadesi $O(n^2 \log n)$ şeklinde ifade edilir
- Note: Aslında $(50\ n \log n)$ ifadasi $O(n^5)$ şeklindede gösterilebilir, ancak yaklaşık olarak fonksiyonla aynı dereceden alınması gerekmektedir

Asymptotic Notasyonlar

- "big-Theta" Θ–İfadesi
  - asymptotic tight bound
  - *f(n) = Θ(g(n)),* eğer sabit *c_1, c_2,* ve *n_0, değerleri için* **c_1 g(n) ≤ f(n) ≤ c_2 g(n)** bütün $n \geq n_0$ değerleri için doğruysa



Asymptotic Notasyonlar

- İki tane daha asymptotic ifade
  - "Little-O" ifadesi *f(n)=o(g(n))* *Örn: $2n = o(n^2)$*
    - Her *c>0* için, bir n_0 değeri vardır ve 0 ≤ **f(n) < c.g(n)** ifadesi bütün $n \geq n_0$ değerleri için doğrudur
    - Çalışma sürelerinin karşılaştırılması için kullanılır. Eğer *f(n)=o(g(n)),* ise *g(n), f(n)* fonksiyonundan daha ağırlıklıdır (dominates).

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

  - "Little-omega" ifadesi *$f(n)=\omega(g(n))$* *Örn: $n^2/2 = \omega(n)$*
    - Her *c>0* için, bir n_0 değeri vardır ve **f(n) > c.g(n)** ≥ 0 ifadesi bütün $n \geq n_0$ değerleri için doğrudur
    - Çalışma sürelerinin karşılaştırılması için kullanılır. Eğer *$f(n)=\omega(g(n))$,* ise *f(n), g(n)* fonksiyonundan daha ağırlıklıdır (dominates).

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Asymptotic Notasyonlar

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

- $f(n) = O(g(n))$ gerçekte $f(n) \in O(g(n))$ anlamındadır

Çalışma süreleri

Çalışma süresi (µs)	Maksimum problem boyutu (n)		
	1 saniye	1 dakika	1 saat
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Algoritmaların Doğruluğu

- Bir algoritma geçerli olan herhangi bir giriş için sonlanmakta ve istenen bir sonucu üretmekte ise doğrudur.
- Algoritmaların doğruluğunun kontrolünde pratik teknikler kullanılır.

Loop Değişmezleri (Invariants)

- **Invariants** – Herhangi bir zamanda ulaşıldıklarında veya işlem yapıldıklarında doğru oldukları varsayılır (algoritma çalışma sırasında tekrarlı gerçekleşen işlemler, örnek: döngülerde)
- Döngü değişmezleri için üç şeyi göstermek zorunludur:
  - **Initialization** – ilk iterasyondan önce doğrudur
  - **Maintenance** – bir iterasyondan önce doğruysa bir sonraki iterasyondan öncede doğruluğunu korur
  - **Termination** – döngünün değişmezleri bitirmesi algoritmanın doğruluğunu gösterir

Örnek : Binary Search (1)

- NIL değeri döndüğünde q değerinin A dizisinde olmadığından emin olmak istiyoruz.
- **Invariant**: her **while** döngüsünün başlangıcında, *A[i] < q bütün i ∈ [1..left-1] and A[i] > q bütün i ∈ [right+1..n]*
- **Initialization**: *left = 1, right = n* olarak seçilir ve left'in solunda ve right'ın sağında hiçbir eleman kalmaz

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

Örnek : Binary Search (2)

- **Invariant**: her **while** döngüsünün başlangıcında, *A[i] < q bütün i ∈ [1..left-1] and A[i] > q bütün i ∈ [right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

- **Maintenance**: Eğer *A[j]>q,* ise *A[i] > q* olur her *i ∈ [j..n]*, çünkü dizi sıralıdır. Algoritma *right* değişkenine *j-1* değerini atar.

Örnek : Binary Search (3)

- **Invariant**: her **while** döngüsünün başlangıcında, *A[i] < q bütün i ∈ [1..left-1] and A[i] > q bütün i ∈ [right+1..n]*

```
left←1
right←n
do
   j←⌊(left+right)/2⌋
   if A[j]=q then return j
   else if A[j]>q then right←j-1
   else left=j+1
while left<=right
return NIL
```

- **Termination**: *left > right* olduğunda loop bitirilir. *q* değeri *left*'in solundaki A 'nın tüm değerlerinden büyüktür veya *right*'ın sağındaki A'nın tüm değerlerinden küçüktür. Bu A'nın tüm elemanlarından *q* değerinin küçük yada büyük olduğunu gösterir.

Örnek : Insertion Sort (1)

- **Invariant**: her for döngüsü başlangıcında, A[1...j-1] dizisi sıralı olarak A[1...j-1] aralığındaki elemanlardan ibarettir.

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

Örnek : Insertion Sort (2)

- **Invariant**: her for döngüsü başlangıcında, A[1...j-1] dizisi sıralı olarak A[1...j-1] aralığındaki elemanlardan ibarettir.

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

- **Initialization**: *j = 2,* olarak alınır çünkü A[1] zaten sıralıdır.

Örnek : Insertion Sort (3)

- **Invariant**: her for döngüsü başlangıcında, A[1...j-1] dizisi sıralı olarak A[1...j-1] aralığındaki elemanlardan ibarettir.

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

- **Maintenance**: inner **while** loop elamanlar arasında *A*[*j-1*], *A*[*j-2*], …, *A*[*j-k*] şeklinde sıralarını değiştirmeden bir önceki elemana gider. Daha sonra *A*[*j*] elemanı *k.* pozisyona yerleştirilir, böylece $A[k-1] \leq A[k] < A[k+1]$ olur.

  A[*1…j-1*] sıralı + *A*[*j*] ⟶ *A*[*1…j*] sıralı

Örnek : Insertion Sort (4)

- **Invariant**: her for döngüsü başlangıcında, A[1...j-1] dizisi sıralı olarak A[1...j-1] aralığındaki elemanlardan ibarettir.

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

- **Termination**: *j=n+1* olduğunda döngü bitirilir. Böylece *"A[1…n] orijinal olarak alınmış olan A[1…n] elemanla aynıdır ancak sıralanmış şekildedir.*