BM 210 Algoritma Analizi (Analysis of Algorithms)

Hazırlayan: M.Ali Akcayol
Gazi Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Sıralama Algoritmaları
- Quicksort
  - Popüler algoritma, average case'de çok hızlıdır
- Heapsort
  - Heap veri yapısını kullanır.

Niçin Sıralama ?

- Sıralama birçok algoritmada subroutine olarak kullanılır:
  - Veritabanında arama: sıralı data üzerinde binary search yapabiliriz
  - Eleman tekliğinin sağlanması, çiftlerin elimine edilmesi
  - Bilgisayar grafik ve geometri hesaplama problemleri
    - En yakın çift (closest pair)
    - En kısa yol (shortest path)

Niçin Sıralama ?

- Farklı algoritma teknikleri ile çok sayıda sıralama algoritması geliştirilmiştir.
- Sıralama için lower bound $\Omega(n \log n)$ olarak elde edilebilmektedir.

Sıralama Algoritmaları

- Insertion sort, selection sort, bubble sort
  - Worst-case çalışma süresi $\Theta(n^2)$; in-place
- Merge sort
  - Worst-case çalışma süresi $\Theta(n \log n)$, ancak $\Theta(n)$ boyutunda ek alan gerektirir;

Quick Sort

- Karakteristik
  - Insertion sort gibi diziyi yerinde sıralar ve ek dizi gerektirmez
  - Çok pratik uygulanabilir ve average case sıralama performansı $O(n \log n)$ worst case sıralama performansı $O(n^2)$

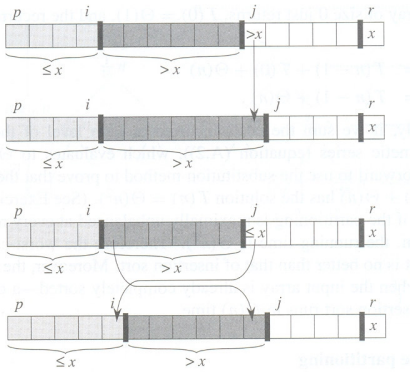
Quick Sort

- Divide-and-conquer algoritması
  - **Divide**: diziyi iki parçaya böler düşük parçadaki elemanlar yüksek parçadaki elemanlardan küçük veya eşittir
  - **Conquer**: özyinelemeli olarak (recursively) 2 altdizi sıralanır
  - **Combine**: yerinde sıraladığı için önemli değildir

Partitioning (1)

- Linear time partitioning procedure

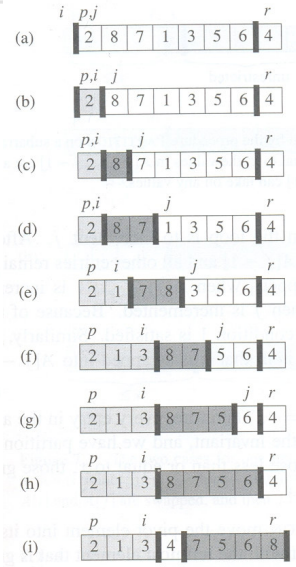
```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



Quick Sort Algoritması (1)

- Başlangıçta **Quicksort(A, 1, length[A])**

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

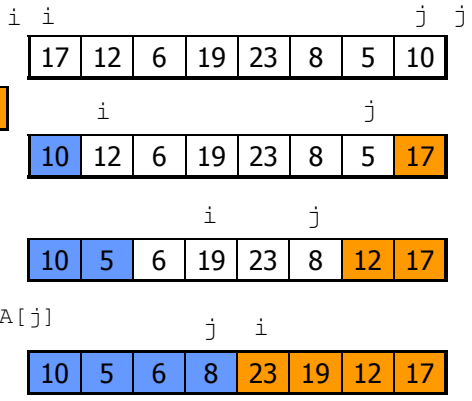


Partitioning (2)

- Linear time partitioning procedure

```
Partition(A,p,r)
01 x←A[r]
02 i←p-1
03 j←r+1
04 while TRUE
05    repeat j←j-1
06       until A[j] ≤x
07    repeat i←i+1
08       until A[i] ≥x
09    if i<j
10       then exchange A[i]↔A[j]
11       else return j
```

≤ X=10 ≤



Quick Sort Algoritması (2)

- Başlangıçta **Quicksort(A, 1, length[A])**

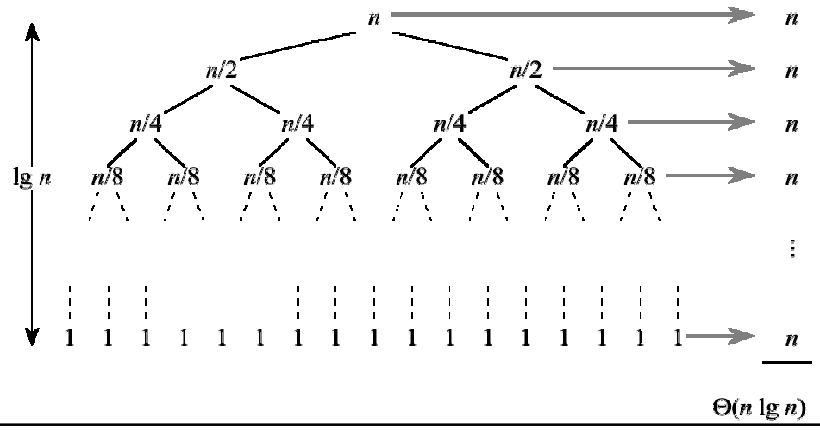
```
Quicksort(A,p,r)
01 if p<r
02    then q←Partition(A,p,r)
03           Quicksort(A,p,q)
04           Quicksort(A,q+1,r)
```

Quicksort Algoritmasının Analizi

- Bütün girişlerin birbirinden farklı olduğu kabul edilirse
- Çalışma süresi parçaların dağılımına bağlıdır

Best Case

- Partition diziyi düzgün bir şekilde ikiye böler $T(n) = 2T(n/2) + \Theta(n)$

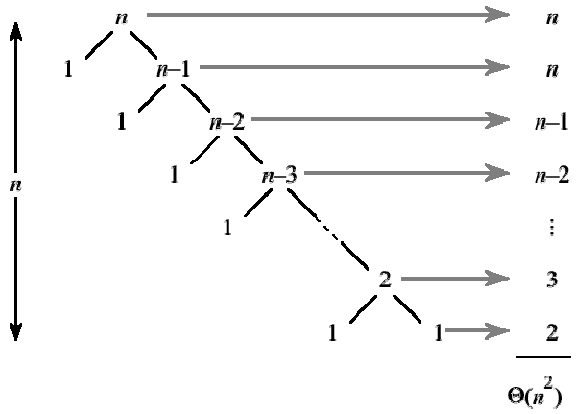


Worst Case

- Parçalardan birisi sadece bir elemana sahiptir

$$
\begin{aligned}
T(n) &= T(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) \\
&= \Theta(n^2)
\end{aligned}
$$

Worst Case (2)



Worst Case (3)

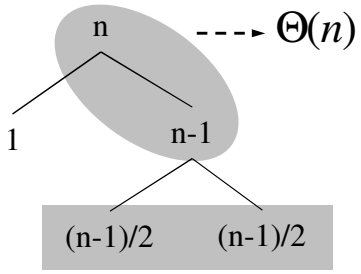
- Aşağıdaki durumlarda worst case oluşur
  - girişler sıralıdır
  - girişler ters sıralıdır
- Worst case durumu insertion sort ile aynı recurrence ifadeye sahiptir
- Ancak sıralı girişler insertion sort için best case olur.

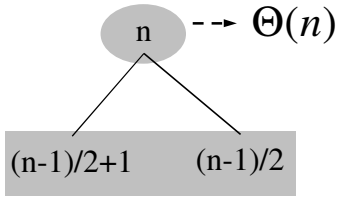
Average Case Durumu

- En iyi (şanslı-lucky) ve en kötü (şansız-unlucky) durumların birleşimi average case olarak alınır

$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

$$L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$$
$$= 2L(n/2-1) + \Theta(n)$$
$$= \Theta(n \log n)$$





Average Case Durumu

- Genellikle şanslı durumda olmak için
  - Ortadaki elemanın yakınından (n/2) bölme yapılır ?
  - Rastgele seçilen bir elemana göre bölme yapılır (pratikte daha iyi çalışır)
- Randomized algorithm
  - Çalışma süresi girişin sırasından bağımsızdır
  - worst-case durumu için belirlenmiş bir giriş yoktur
  - worst-case durumu sadece rastgele sayı üreteci tarafından belirlenir

Randomized Quicksort

- Bütün elemanların farklı olduğunu kabul edelim
- Rastgele seçilen elemanın yakınından bölünür
- Bütün bölme (1:n-1, 2:n-2, ..., n-1:1) durumları 1/n oranında eşit olasılığa sahiptir
- Rastgele seçim algoritmanın average-case durumunu iyileştirir

Randomized Quicksort (2)

```
Randomized-Partition(A,p,r)
01 i←Random(p,r)
02 exchange A[r]↔A[i]
03 return Partition(A,p,r)

Randomized-Quicksort(A,p,r)
01 if p<r then
02    q←Randomized-Partition(A,p,r)
03    Randomized-Quicksort(A,p,q)
04    Randomized-Quicksort(A,q+1,r)
```

Selection Sort

```
Selection-Sort(A[1..n]):
   For i → n downto 2
A:     Find the largest element among A[1..i]
B:     Exchange it with A[i]
```

- A $\Theta(n)$ süresinde ve B $\Theta(1)$ süresinde yapılır: toplam olarak $\Theta(n^2)$ süresinde çalışır

Heap Sort

- *A* bir binary heap veri yapısı
  - Dizi
  - Complete binary tree olarak görülebilir
    - En düşük seviye hariç tüm seviyeler tamamen doludur
  - Root içindeki data kendi child node'larındakinden büyüktür ve sol ve sağ subtree'ler binary heap yapısındadır

Heap Sort

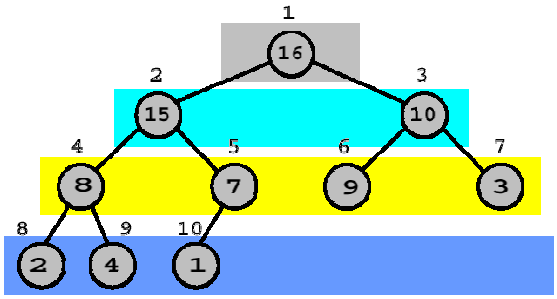
Parent (*i*)
return $\lfloor i/2 \rfloor$

Left (*i*)
return $2i$

Right (*i*)
return $2i+1$

Heap özelliği:

$A[\text{Parent}(i)] \geq A[i]$



1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

Level: 3 2 1 0

Heapify

- *i* , A dizisindeki bir indekstir
- Left(*i*) ve Right(*i*) root'larının subtree'leri heap yapısındadır
- Eğer *A*[*i*] kendi child node'larından küçük olursa heap özelliği ortadan kalkar
- **Heapify** metodu *A*[*i*] elemanını aşağıya doğru taşıyarak heap özelliğini oluşturur

Heapify (2)

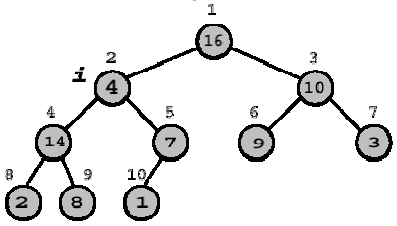
n is total number of elements

HEAPIFY(A, i)

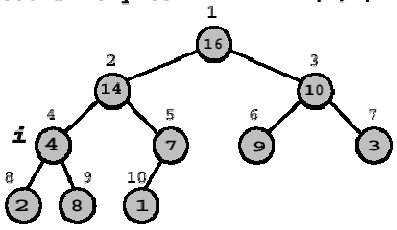
```
1  ▷ Left & Right subtrees of i are heaps.
2  ▷ Makes subtree rooted at i a heap.
3  l ← LEFT(i)          ▷ l = 2i
4  r ← RIGHT(i)         ▷ r = 2i + 1
5  if l ≤ n and A[l] > A[i]
6      then largest ← l
7      else largest ← i
8  if r ≤ n and A[r] > A[largest]
9      then largest ← r
10 if largest ≠ i
11     then exchange A[i] ↔ A[largest]
12          HEAPIFY(A, largest)
```

Heapify Örnek

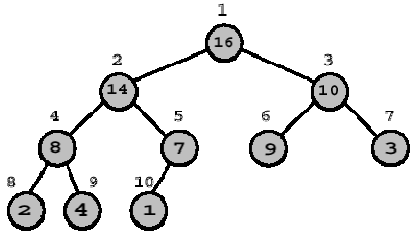
1. Call HEAPIFY(A,2)



2. Exchange A[2] with A[4] and recursively call HEAPIFY(A,4)



3. Exchange A[4] with A[9] and recursively call HEAPIFY(A,9)



4. Node 9 has no children, so we are done.

Heapify: Çalışma Süresi

- Heapify çalışma süresi i root node'una sahip n boyutlu bir subtree için
  - Elemanlar arasındaki ilişkiyi bulma : $\Theta(1)$
  - En kötü durumda bir subtree en fazla 2n/3 node' a sahiptir. Böylece en kötü durum için aşağıdaki recurrence ifade elde edilir.

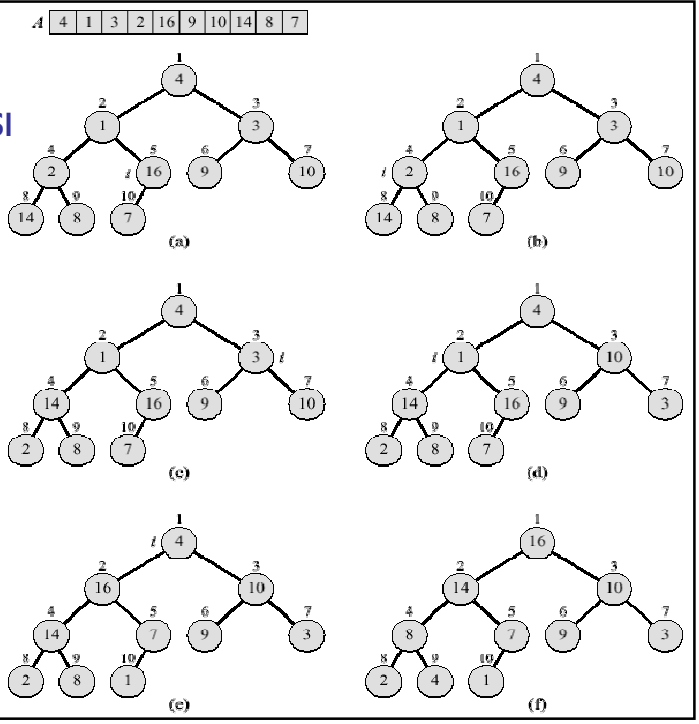
$$T(n) \le T(2n/3) + \Theta(1) \;\Rightarrow\; T(n) = O(\log n)$$

Heap Yapılandırılması

- $A[1...n]$ dizisinin n = length[A] olan bir heap'e dönüştürülmesi
- Altdizideki $A[(\lfloor n/2 \rfloor + 1)...n]$ elemanlar heap durumundadır

```
BUILD-HEAP(A)
1 for i ← ⌊n/2⌋ downto 1
2       do HEAPIFY(A, i)
```

Heap yapılandırması



Heap Yapılandırması Analizi

- Correctness: *m* > *i* olmak kaydıyla m değerlerinin root'u olan tüm i'ler heap özelliğini taşır
- Running time: *n/2* kez Heapify çalışır
- *O(nlg n)*

Heap Yapılandırması Analizi

- Tanımlamalar
  - node yüksekliği: node'dan yaprağa giden en uzun yol
  - tree yüksekliği: root'un yüksekliği

```
BUILD-HEAP(A)
1 for i ← ⌊n/2⌋ downto 1
2       do HEAPIFY(A, i)
```

  - heapify süresi = O(i root node'unun subtree yüksekliği (k))
  - $n = 2^k - 1$ olur (complete binary tree $k = \lfloor lg\ n \rfloor$)

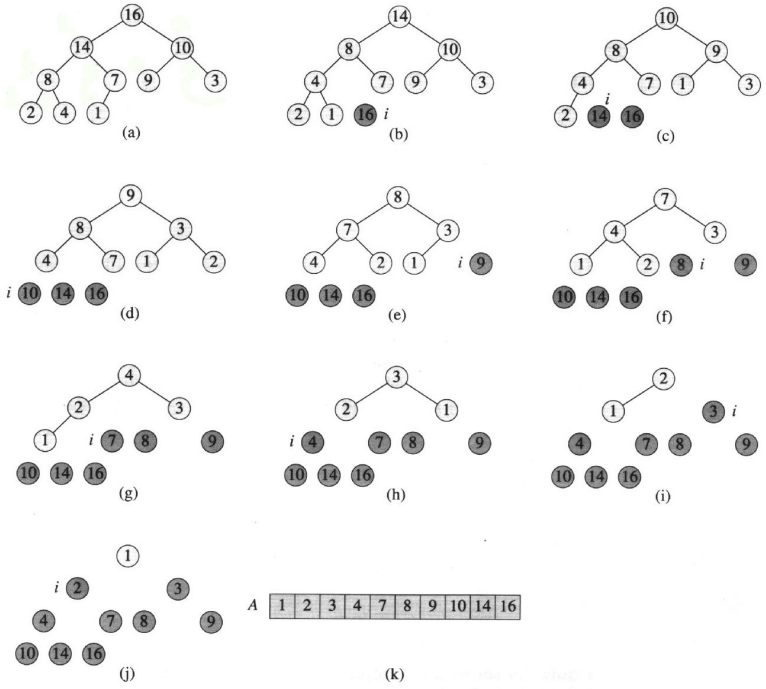
$$T(n) = O(n)$$

Heap Sort

```
HEAPSORT(A)                              Analysis
1 BUILD-HEAP(A)                          O(n)?
2 for i ← n downto 2                     n times
3       do exchange A[1] ↔ A[i]          O(1)
4          n ← n − 1                     O(1)
5          HEAPIFY(A, 1)                 O(lg n)
```

- Heap sort algoritmasının toplam çalışma süresi *O(n lg n)* + Build-Heap(*A*) süresidir. Build-Heap süresi ise *O(n)* 'dir.

Heap Sort



Haftalık Ödev

- Quick sort ve Heap sort algoritmalarının performanslarını karşılaştıran bir program yazınız. Programda dışarıdan girilen sayıda elemandan oluşan bir integer dizi oluşturulacak ve dizi elemanları rastgele alınacak. Elde edilen giriş dizisi her iki algoritmayla sıralanacak ve geçen toplam süreler bulunacaktır. Her iki algoritma best case, worst case ve average case için test edilecektir. Program C# ile yazılacak ve görsel arabirime sahip olacaktır.