Algorithm Analysis And Design

With Infinity Teams

Eng: Abdulrahman Hamdi

INFINITY TEAM
Since 2020

Table of contents

Eng: Abdulrahman Hamdi

Table of contents

Eng: Abdulrahman Hamdi

08

Divide-and-Conquer

Eng: Abdulrahman Hamdi

Divide-and-Conquer Çerçevesi

- Divide and Conquer introduction
- Karatsuba
- Mergesort
- Quicksort

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

Algoritma dizaynında bir diğer önemli teknik Divide and Conquer (Böl ve Yönet) tekniği, Böl ve Yönet tipindeki algoritmalar verilen orijinal problemi, orijinal problemden daha küçük boyutta fakat aynı tipte problemlere bölerler, bu şekilde ortaya çıkan her problemi çözer, daha sonra bu çözümleri birleştirerek orijinal problemin çözümünü elde etmiş olurlar.
Böl problemi, aynı problemin daha küçük örnekleri olan alt problemlere ayır.

Yönet (Çöz) alt problemleri özyineli (recursive) olarak çözerek. Ancak alt problemler yeterince küçükse, bu alt problemleri doğrudan ve basit bir şekilde çöz.

Birleştir alt problemlerin çözümlerini, orijinal problemin çözümünü elde etmek için bir araya getir.

Böl ve yönet algoritmaları aşağıdaki genel plana göre çalışır:
1- Bir problem, ideal olarak eşit boyutta, aynı türden birkaç alt probleme bölünür.
2- Alt problemler çözülür (tipik olarak yinelemeli olarak, ancak alt problemler yeterince küçük hale geldiğinde bazen farklı bir algoritma kullanılır).
3- Gerekirse, alt problemlerin çözümleri orijinal problemin çözümünü elde etmek için birleştirilir.

Eng: Abdulrahman Hamdi

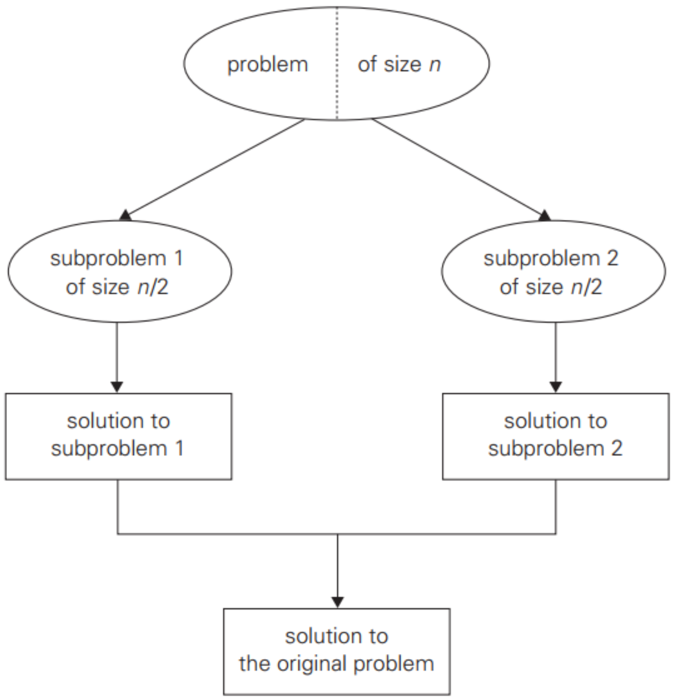
Divide-and-Conquer Algoritmaları

- En sık kullanılan algoritma tasarım tekniklerinden biri

1. Problem alt problemlere bölünür

- Genellikle eşit büyüklükte

2. Alt problemler çözülür
3. Gerekliyse alt problem çözümleri birleştirilir.



Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

– Her böl ve yönet algoritması verimli olmak zorunda değildir. Basit bir örnek olarak dört sayıyı bölerek toplama algoritması verimsiz olabilir.
– Ancak, böl ve yönet yaklaşımı genellikle çok önemli ve verimli algoritmalar üretir. Bilgisayar biliminde birçok klasik ve etkili algoritma bu prensibe dayanır.
– Böl ve yönet, paralel hesaplamalar için idealdir. Alt problemler eş zamanlı olarak farklı işlemciler tarafından çözülebilir.
– Tipik bir durumda, n büyüklüğündeki bir problem iki adet n/2 büyüklüğünde alt probleme bölünür. Daha genel olarak, n büyüklüğündeki bir problem b adet n/b büyüklüğünde alt probleme bölünebilir ve bunlardan a tanesi çözülür.

– Algoritmanın çalışma zamanı (T(n)) için genel bir yineleme bağıntısı (recurrence relation) verilir:

$$T(n) = aT(n/b) + f(n),$$

– Burada f(n), bölme ve birleştirme adımları için harcanan zamandır.
– Çalışma zamanının büyüme hızı (order of growth), a, b sabitlerine ve f(n) fonksiyonunun büyüme hızına bağlıdır.
– Bu tür yineleme bağıntılarının çözümü için bir teoremin (Ek B'de belirtilen) algoritma verimliliği analizini büyük ölçüde kolaylaştırdığı ifade ediliyor.

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Master Teoremi**

– Bir özyineleme ilişkisi için

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence $T(n) = aT(n/b) + f(n)$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

O ve Ω notasyonları için de benzer sonuçlar geçerlidir.

Örneğin, $n = 2^k$ boyutundaki girdiler üzerinde böl ve fethet toplama hesaplama algoritması (yukarıya bakın) tarafından yapılan toplama sayısı A(n)'nin reküransı şöyledir:

A(n) = 2A(n/2) + 1

Bu örnek için a = 2, b = 2 ve d = 0'dır; bu nedenle,

$$a > b^d, A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Örnek :

$T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

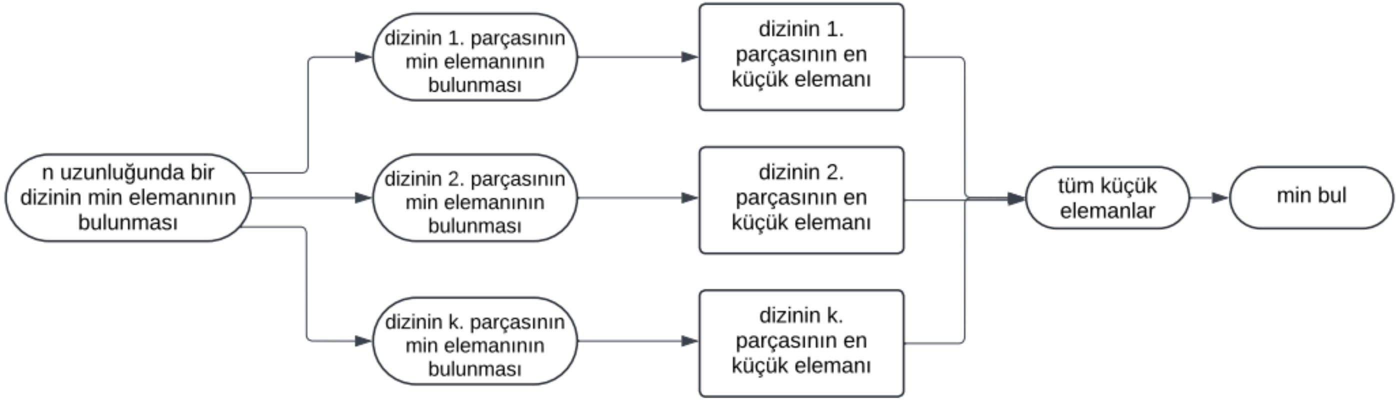
$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

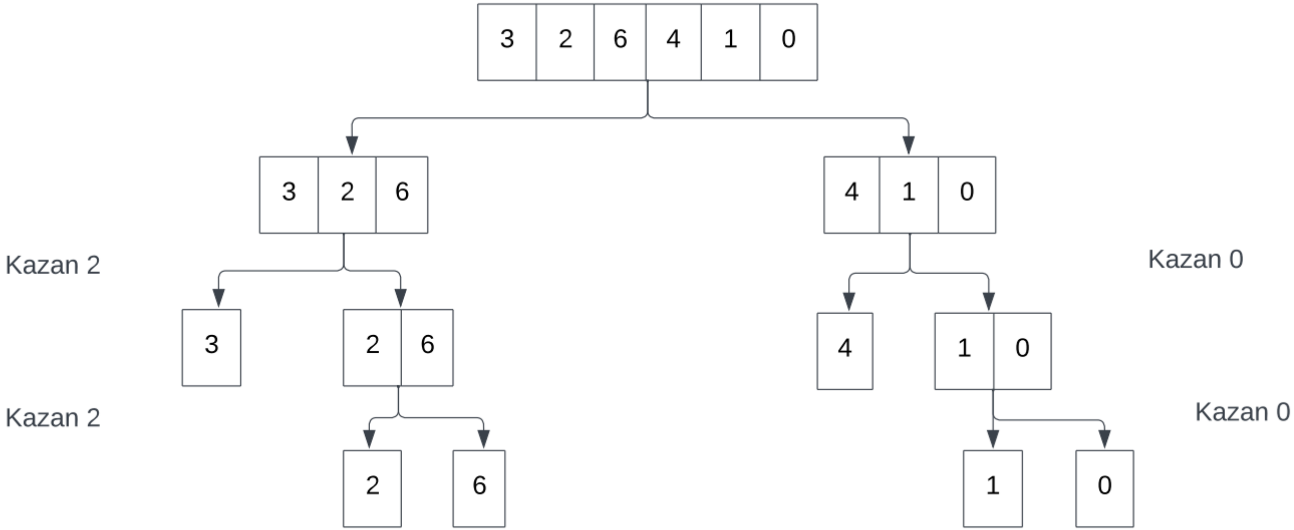
Örnek olarak bir dizinin en küçük elemanının bulunması problemini ele alalım.



Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

Örnek olarak bir dizinin en küçük elemanının bulunması problemini ele alalım.



Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

```
minBl_Böl_Yönet(A[0], ..., A[n-1])
// birden uzunluğunda bir A dizisi
// çıktı: A'nın en küçük elemanı
n ← A'nın uzunluğu
if n == 1 // A eğer 1 elemandan oluşuyorsa
    return A[0]
birinci_yarı ← A[0], ..., A[⌊n/2⌋ - 1]
ikinci_yarı ← A[⌊n/2⌋], ..., A[n-1]
ilk_min ← minBl_Böl_Ysret(birinci_yarı)
ikinci_min ← minBl_Böl_Ysret(ikinci_yarı)
if ilk_min < ikinci_min
    return ilk_min
else
    return ikinci_min
```

Rekürsif tüm algoritmaların → temel durumu (base case) olur.
(Algoritmanın daha ileriye gitmemesi için)

Algoritmanın kendini → çağırması, (Rekürsiyon)

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

Şimdi bu algoritmanın zaman verimliliğine bakalım. Bu algoritma n uzunluğunda bulunan min elemanını bulması için gereken zaman T(n) olsun.
T(n) → n uzunluğunda bir dizinin minBl_Böl_Ysret ile minimum elemanını bulunması için geçen süre. Bu algoritma aldığı n uzunluğundaki bir diziyi yarı boyüt 2 parçaya ayırıyor ve bir parçaya kendini uyguluyor. O halde her bir parçanın yine bu algoritmayla çözülmesi yani min elemanın bulunması için geçen süre T(n/2) dir. Bu sürelerden 2 tane geçmesi lazım (birinci ve ikinci parça için). Son olarak iki parçadan gelen iki minimumun kıyaslanması için 1 adet işleme daha ihtiyacımız var. O halde T(n): **T(n): 2xT(n/2) + 1**

Aynı mantıkla n/2 uzunluğundaki her bir dizi (parçası) nın minimumunun bulunması için herbiri iki parçaya bölünerek (n/4 uzunluğa düşmüş olarak 4 parçalar) daha sonra bu küçük parçalara da algoritma uygulanarak, bunun sonucunda gelen minimumlar kıyaslanıp dizinin n/2 büyüklüğündeki parçası için min bulunmuş olacak. O halde **T(n/2) = 2 x T(n/4) + 1**

n/4 büyüklüğündeki dizilerin algoritma ile min'lerinin bulunması için gereken süre.
+1 kısmını n/4 büyüklüğündeki dizilerden gelen min'lerin kıyaslanması için gelen süre. n/4 büyüklüğündeki

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

T(n/2) = 2 x T(n/4) + 1 Bunu bir önceki denklemde yerine koyarak

T(n) = 2 x (2 x T(n/4) + 1) + 1

= 4 x T(n/4) + 3

Aynı şekilde T(n/4) = 2 x T(n/8) + 1 yazabiliriz.

Böylece

T(n) = 4 x (2 x T(n/8) + 1) + 3

= 8 x T(n/8) + 7

Bulduklarımızı toplarsak

T(n) = 2 x T(n/2) + 1 → 1 kere bölersek

T(n) = 4 x T(n/4) + 3 → 2 " "

T(n) = 8 x T(n/8) + 7 → 3 " "

:

:

T(n) = $2^i \, x \, T(n/2^i) \; + \; 2^i \; - \; 1$ // i kere bölersek

Genel formül

Divide-and-Conquer Algoritmaları

n uzunluğundaki bir diziyi en fazla tek eleman kalana kadar bölebiliriz.

Yani $\frac{n}{2^i}$ en küçük 1 olabilir. Buna kaç bölmede ulaşabileceğimize bakalım.

$\frac{n}{2^i} = 1 \rightarrow i = \log_2 n$ olur. Yani n uzunluğundaki bir diziyi en fazla $\log_2 n$ defa ikiye bölebiliriz. i yerine $\log_2$n koyarsak genel formülde

$T(n) = 2^{\log_2 n} * T(1) + 2^{\log_2 n} - 1$ ($2^{\log_2 n}$ n'e eşit)

($T(1)$ n uzunluğundaki bir dizinin min elemanı budur olduğu için (yani base case temel durum soluyor) T(1) = 1 olur. (1 adet işlem yapılır))

$$T(n) = n + n - 1 \rightarrow T(n) = 2n - 1 \in O(n)$$

Bildiğimiz gibi, uzunluğu n olan bir dizinin en küçük elemanını bulmak için genellikle doğrudan (brute-force) bir yöntem kullanılır ve bu yöntemde bir for döngüsüyle dizinin tüm elemanları gezilir. Bu algoritmanın zaman karmaşıklığı T(n) = O(n) olur. Bu durumda, daha verimli (daha düşük zaman karmaşıklığına sahip) bir yöntem kullanmadıkça bu işlem için daha iyi bir performans elde edemeyiz.

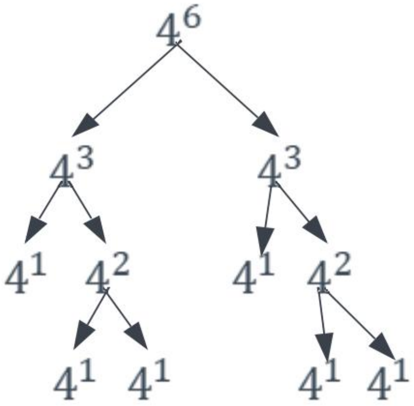
Divide-and-Conquer Algoritmaları

Ör. Şimdi a^n değerini böl ve fethet ile bulmaya çalışalım. (a bir tamsayı)

$4^6 = 4^3 \, x \, 4^3$ şeklinde yazabiliriz.

$$4^3 = 4^2 \, x \, 4^1$$

$$4^2 = 4^1 \, x \, 4^1$$




```
UstBul-Bol-Yonet(a,n)
//Girdi: a ve n tam sayılar
//Çıktı: a^n’in n. kuvveti
if n == 1
    return a
else
    return UstBul-Bol-Yonet(a, ⌊n/2⌋) *
    UstBul-Bol-Yonet(a, ⌈n/2⌉)
```

Divide-and-Conquer Algoritmaları

Soru: n!'i bil ve böl ve yönet ile bulmaya çalışalım.

```
          6!
         /  \
     1·2·3    4·5·6
     /  \     /  \
    1   2·3  4   5·6
        / \      / \
       2   3    5   6
```

6! = (1·2·3) × (4·5·6)

→ bir çarpıma ihtiyacımız var

Sonuç olarak: Bizim herhangi bir başlangıç değerinden herhangi bir bitiş değerine kadar olan sayıların çarpımını hesaplayan bir fonksiyona ihtiyacımız var.Bu fonksiyon kendisi böl ve yönet tipinde olacak.

```
Carp-Bol-Yonet(bas, son)
if bas == son
    return bas
orta = (bas + son) / 2
return Carp-Bol-Yonet(bas, ⌊orta⌋) * Carp-Bol-
    Yonet(⌊orta⌋ + 1, son)
```

Şimdi tek yapmamız gereken bu fonksiyonu baş=1 için çağırmak:

```
faktor_hesapla(n)
// Girdi: n tam sayısı
// Çıktı: n faktöriyel
return Carp-Bol-Yonet(1, n)
```

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

Karatsuba Algoritması (Büyük sayıların çarpımı)

- 1. n basamaklı bir sayı
- 2. n basamaklı sayı

🔁 2. basamaktaki sayı, 1. n basamaklı sayının her basamağı ile çarpılır.

2. n basamaklı sayının her bir basamağı, 1. n basamaklı sayının her basamağı ile çarpılır.sayıdaki 1. basamak için n adet çarpım" "
2. basamak için n adet çarpım:" " n. basamak için n adet çarpım
Toplam çarpım sayısı = n + n + ... + n = n^2 (n adet)

Dolayısıyla 2 adet n basamaklı sayı klasik yöntemle çarpılırkenn^2 adet işlem yapılır. Dolayısıyla bu çarpım $O(n^2)$'ye aittir.Bu yüzden çok büyük basamaklı sayıların birbiriyle çarpımı çok fazla işlem gerektirir, işlemciye yüklüdür.Böyle sayıların çarpımını kolaylaştırmak adına Andrey Karatsuba, böl ve yönet mantığına sahip bir algoritma geliştirmiştir.
n basamaklı x ve y tam sayıları şu şekilde yazılabilir:(devamı sanırım bir sonraki sayfada olacak)

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

$X = a \cdot 10\text{^}(\lfloor n/2 \rfloor) + b$ → a, X'in ilk ⌊n/2⌋ basamağı
→ b, son ⌈n/2⌉ basamağı

Örneğin X = 123456 iken
→ a = 123, b = 456'dır. Bu yüzden X,
→ $X = 123 \cdot 10^3 + 456$ şeklinde yazılabilir.

$y = c \cdot 10\text{^}(\lfloor n/2 \rfloor) + d$ → c, y'nin ilk ⌊n/2⌋ basamağı
→ d, son ⌈n/2⌉ basamağı

Şu halde:
$X \cdot y = (a \cdot c) \cdot 10^n + (a \cdot d + b \cdot c) \cdot 10\text{^}(n/2) + b \cdot d$ olur.

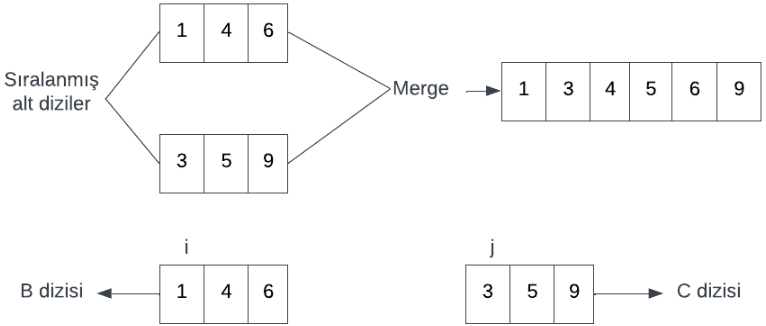
```
karatsuba(x, y)
// Girdi: n basamaklı x ve y tam sayıları
// Çıktı: x ve y'nin çarpımı
    if x < 10 veya y < 10 // Sayılardan biri tek
    basamaklı ise
    return x * y
    a ← x'in ilk ⌊n/2⌋ basamağı
b ← x'in son ⌈n/2⌉ basamağı
c ← y'nin ilk ⌊n/2⌋ basamağı
d ← y'nin son ⌈n/2⌉ basamağı
    ac = karatsuba(a, c)
ad = karatsuba(a, d)   ← rekurisyon!
bc = karatsuba(b, c)
bd = karatsuba(b, d)
    return ac · 10^n + (ad + bc) · 10^(n/2) + bd
```

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

• **Birleştirmeli Sıralama (Merge Sort)**
Böl ve yönet mantığını kullanan sıralama algoritmaları MergeSort ve QuickSort'tur.Bunlardan MergeSort basitçe, aldığı bir diziyi ikiye böler, sıralar, daha sonra bu sıralanmış dizileri sıra korunacak şekilde birleştirir (merge eder).
– Bir sırasız diziyi sıralı hale getirme
– Diziyi özyinelemeli olarak ikiye ayırır
– En küçük hale getirdikten sonra karşılaştırarak birleştirir

Burada asıl olay bu birleştir B'dir

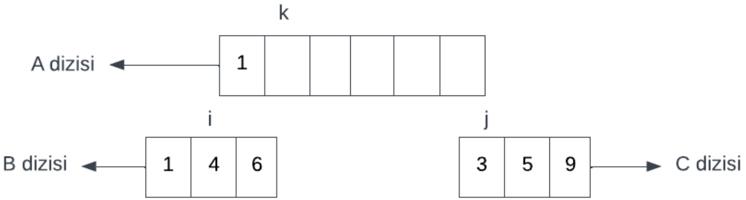


Eng: Abdulrahman Hamdi

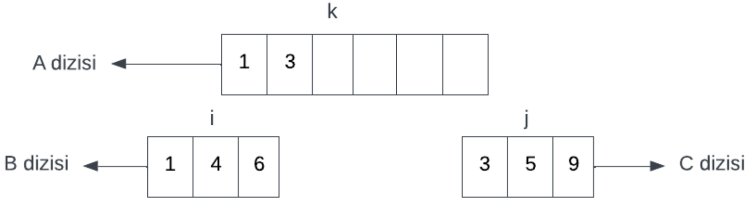
Divide-and-Conquer Algoritmaları

- **Birleştirmeli Sıralama (Merge Sort)**

B[i] ≤ C[j] olduğundan B[i]'yi A'ya (birleşim sırası olacak dizi) ekle. A'nın parametresini k'yi 1 artır, i = i + 1 olur.

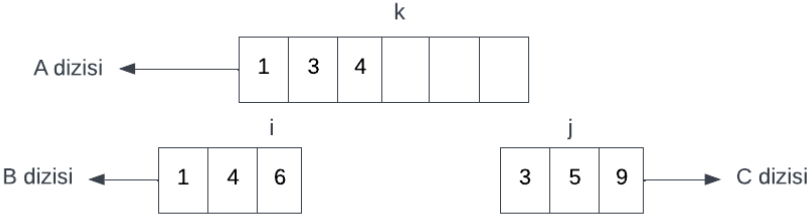


C[j] < B[i] olduğundan C[j]'yi A'ya ekle, k'yi ve j'yi 1 artır.

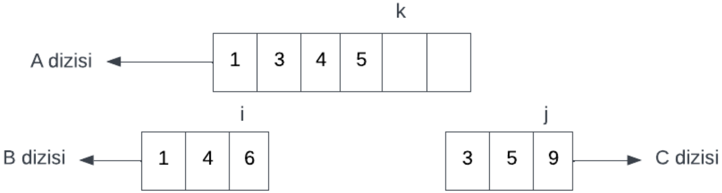


Eng: Abdulrahman Hamdi

B[i] < C[j] olduğundan B[i]'yi A'ya ekle, k'yi ve i'yi 1 artır.



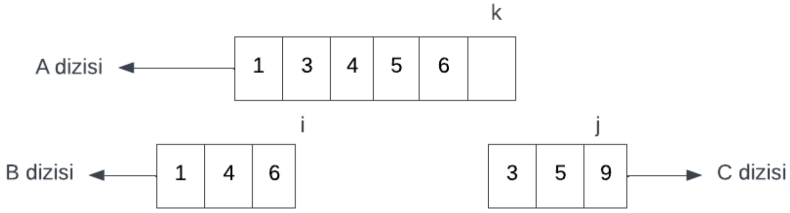
C[j] < B[i] olduğundan C[j]'yi A'ya ekle, k'yi ve j'yi 1 artır.



Divide-and-Conquer Algoritmaları

- **Birleştirmeli Sıralama (Merge Sort)**

B[i] < C[j] olduğundan B[i]'yi A'ya ekle, k'yi ve i'yi 1 artır.



Bu durumda i, B'nin dışına çıkmıştır ve B'nin tamamı A'ya eklenmiştir. B bitmiş, C kalmıştır. Geriye kalan son değer C'nin kalanı A'ya eklenecektir. Şimdi bu mantığı kullanan Merge algoritmasını yazalım.

```
MergeSort( B[0,…,p-1], C[0,…,q-1] )
// Girdi: p ve q uzunluklarında sıralanmış B ve C dizileri
// Çıktı: B'nin ve C'nin elemanlarından oluşan sıralanmış dizi
p ← B'nin uzunluğu
q ← C'nin uzunluğu
i ← 0
j ← 0
k ← 0
p+q uzunluğunda bir A dizisi oluştur.
while i < p ve j < q:
    if B[i] ≤ C[j]:
        A[k] ← B[i]
        i ← i + 1
    else:
        A[k] ← C[j]
        j ← j + 1
    k ← k + 1

if i == p:  // Bu durumda B dizisi bitmiştir
    A[k,…,p+q-1] ← C[j,…,q-1]  // C'nin kalanını A'ya ekle
else:
    A[k,…,p+q-1] ← B[i,…,p-1]  // B'nin kalanını A'ya ekle
return A
```

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Birleştirmeli Sıralama (Merge Sort)**

Bu kısımda herhangi bir Böl ve Yönet yok. Böl ve Yönet MergeSort'un kendisinde vardır. MergeSort her adımında aldığı diziyi iki parçaya böler, her bir parçaya kendini uygular. Böylece parçaları sıralamış hale getirir. En sonunda sıralanmış parçalar Merge algoritmasıyla birleştirilir.

```
MergeSort(A[0,…,n-1])
// Girdi: n uzunluğunda bir A dizisi
// Çıktı: A'nın sıralanmış hali
n ← A'nın uzunluğu
if n == 1:
    return A
B ← A[0,…,⌊n/2⌋]          // A'nın ilk yarısı
C ← A[⌈n/2⌉,…,n-1]        // A'nın ikinci yarısı
B ← MergeSort(B)           // algoritmanın kendisi
    çağrılıp sıralama yapılır
C ← MergeSort(C)           // aynı şekilde
return Merge(B, C)
```

Divide-and-Conquer Algoritmaları

- **Birleştirmeli Sıralama (Merge Sort)**

Orijinal algoritması:

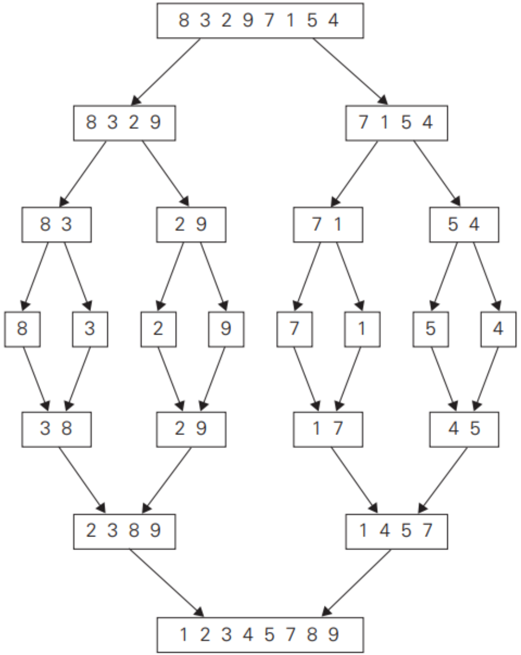
ALGORITHM $Mergesort(A[0..n-1])$
  //Sorts array $A[0..n-1]$ by recursive mergesort
  //Input: An array $A[0..n-1]$ of orderable elements
  //Output: Array $A[0..n-1]$ sorted in nondecreasing order
  if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
    $Mergesort(C[0..\lceil n/2 \rceil - 1])$
    $Merge(B, C, A)$ //see below

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
  //Merges two sorted arrays into one sorted array
  //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
  //Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
  $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
  while $i < p$ **and** $j < q$ **do**
    if $B[i] \le C[j]$
      $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
    else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
    $k \leftarrow k + 1$
  if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
  else copy $B[i..p-1]$ to $A[k..p+q-1]$

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Birleştirmeli Sıralama (Merge Sort)**



Şimdi MergeSort'un zaman verimliliğine bakalım. n uzunluğundaki bir dizinin merge sort ile sıralanması için gereken zamana T(n) diyelim.Bu zamanın içinde n/2 uzunluğundaki 2 dizinin yine MergeSort ile sıralanması için gereken zamanlarT(n/2) + T(n/2) ve bir de n uzunluğundaki bir dizinin merge ile birleştirilmesi için gereken zaman vardır.Toplarsak:

$$T(n) = 2 \times T(n/2) + T_{\text{merge}}(n/2)$$

Bu, n/2 uzunluğundaki sıralanmış dizilerin merge algoritmasıyla sıralanması için gereken süre.Bunu en kötü durum için inceleyelim.Merge için en kötü durum i'nin ve j'nin sona kadar gittiği durumdur. Bu şöyle gerçekleşebilir.

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

• **Birleştirmeli Sıralama (Merge Sort)**

Böyle bir durumda i ve j indexleri n/2 adım atar, yani toplamda while döngüsü n defa çalışır.Bu yüzden $T_{merge}(n/2) = n'dir$.O hâlde: $T(n) = 2 \cdot T(n/2) + n$

Aynı şekilde:

$$T(n/2) = 2 \cdot T(n/4) + n/2$$

$$T(n) = 2 \cdot (2 \cdot T(n/4) + n/2) + n = 4 \cdot T(n/4) + 2n$$

$$T(n/4) = 2 \cdot T(n/8) + n/4 \text{ olduğundan}$$

$$T(n) = 4 \cdot (2 \cdot T(n/8) + n/4) + 2n = 8 \cdot T(n/8) + 3n$$

Genelleştirirsek:

$$T(n) = 2^i \cdot T(n/2^i) + i \cdot n$$

i burada bölünme sayısıdır. Daha önce gördük ki n uzunluğunda bir dizi en fazla $\log_2 n$ kez bölünebilir. O hâlde yerine $\log_2 n$ yazarsak:

$$T(n) = 2^{\log_2 n} \cdot T(1) + n \cdot \log_2 n = n + n \cdot \log_2 n$$

Sonuç: $\boldsymbol{T(n) \in O(n\log n)}$

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

• **Hızlı Sıralama (QuickSort)**

Quicksort, böl ve yönet (divide-and-conquer) yaklaşımına dayanan bir diğer önemli sıralama algoritmasıdır. Quicksort, mergesort’un aksine, giriş elemanlarını dizideki konumlarına göre değil, değerlerine göre ayırır. Bu dizi bölme (partition) fikrine daha önce 4.5. bölümde, selection problemi konusunu tartışırken değinmiştik. Bir "partition" (bölme), dizinin elemanlarının öyle bir şekilde düzenlenmesidir ki, A[s] elemanının solundaki tüm elemanlar A[s]’ten küçük ya da eşit, sağındaki tüm elemanlar ise A[s]’ten büyük ya da eşittir.

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \ge A[s]}$$

Eng: Abdulrahman Hamdi

Açıkça görülüyor ki, bir bölme (partition) işlemi gerçekleştirildikten sonra, A[s] sıralanmış dizideki nihai (son) konumuna yerleşmiş olur ve A[s]'in solundaki ve sağındaki iki alt diziyi bağımsız olarak sıralamaya devam edebiliriz (örneğin, aynı yöntemle). Burada mergesort ile farkı not edin: mergesort’ta problem iki alt probleme doğrudan bölünür ve tüm iş, bu alt problemlerin çözümlerinin birleştirilmesinde gerçekleşir; burada ise tüm iş, bölme aşamasında gerçekleşir ve alt problemlerin çözümlerini birleştirmek için hiçbir iş yapılması gerekmez.

Divide-and-Conquer Algoritmaları

• **Hızlı Sıralama (QuickSort)**
Her zamanki gibi, önce bir pivot seçerek başlıyoruz — bu eleman, alt diziyi değerine göre ikiye ayırmamıza yardımcı olacak. Pivot seçimi için çeşitli stratejiler vardır; algoritmanın verimliliğini analiz ederken bu konuya döneceğiz. Şimdilik, en basit strateji olan alt dizinin ilk elemanını seçiyoruz: **p = A[l].**
Lomuto algoritmasının aksine, alt diziyi uçlardan tarayacağız; pivot ile karşılaştırma yapacağız. Soldan sağa tarama (i ile gösterilen indeks), ikinci elemandan başlar. Pivot'tan küçük elemanların alt dizinin sol kısmında kalmasını istediğimizden, bu tarama pivot'tan küçük olanları atlar ve pivot'a eşit ya da büyük ilk elemanda durur.

Sağdan sola tarama (j ile gösterilen indeks) ise alt dizinin sonundan başlar. Pivot'tan büyük elemanların sağda kalmasını istediğimiz için, bu tarama da pivot'tan büyük olanları atlar ve pivot'a eşit ya da küçük ilk elemanda durur.(Taramaların pivot'a eşit bir elemanda durmasının nedeni nedir? Çünkü bu durum, çok sayıda aynı elemanın bulunduğu dizilerde daha dengeli bir bölünme sağlar. Örneğin, tüm elemanları eşit olan n uzunluğunda bir dizide, aksi halde bölünme n−1 ve 0 olurdu; bu da algoritmanın her adımda yalnızca 1 elemanlık azalma ile çalışmasına neden olurdu.)Her iki tarama da durduktan sonra, üç durum olabilir:Tarama indeksleri i ve j kesişmemişse (i < j), A[i] ile A[j] yer değiştirir.Ardından i artırılır, j azaltılır ve tarama devam eder.

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

• **Hızlı Sıralama (QuickSort)**

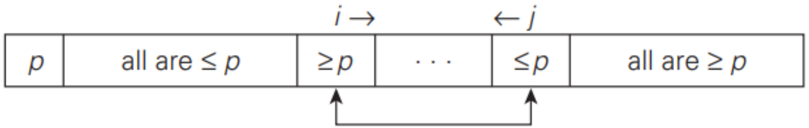
Daha Özet:

Pivot olarak alt dizinin ilk elemanı seçilir.

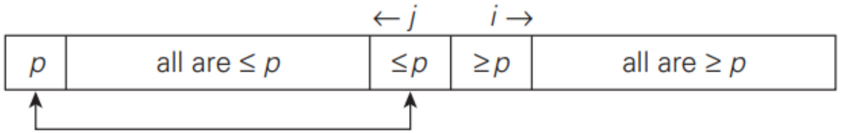
Lomuto yerine Hoare tarzı tarama yapılır:
Soldan sağa tarama (i): pivot'tan küçükleri atlar, büyük veya eşitte durur.
Sağdan sola tarama (j): pivot'tan büyükleri atlar, küçük veya eşitte durur.

i < j ise, A[i] ve A[j] yer değiştirir, sonra i++, j-- yapılır.

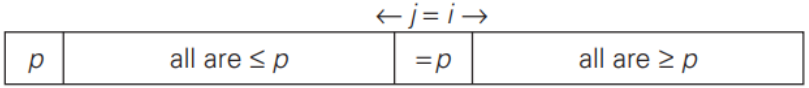
Amaç: daha dengeli bölünmelerle daha verimli sıralama sağlamak, özellikle çok sayıda aynı değerdeki eleman varsa.



Eğer tarama indeksleri birbirini geçmişse, yani i > j, pivot ile A[j] yer değiştirdikten sonra alt dizi bölünmüş olur:



Son olarak, tarama indeksleri aynı elemanı gösterirken durursa, yani i = j, bu durumda işaret ettikleri değer p'ye eşit olmalıdır (neden?). Böylece alt dizi şu şekilde bölünmüş olur ve bölme pozisyonu s = i = j olur:

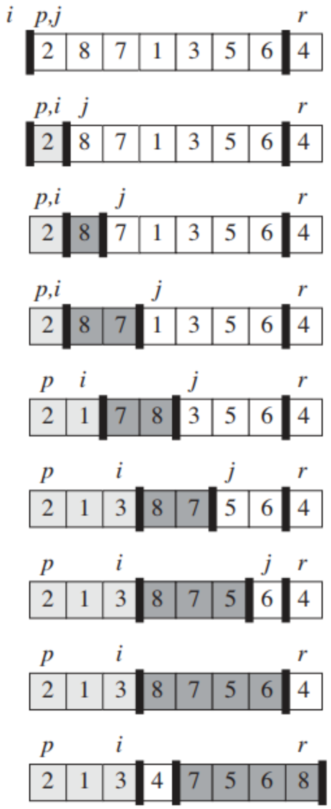


Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Hızlı Sıralama (QuickSort)**

Son durumu (i = j) ile birbirini geçen indeksler durumu (i > j) birleştirilebilir: i ≥ j olduğunda pivot A[j] ile değiştirilerek işlem tamamlanır.
Aşağıda, bu bölme prosedürünü gerçekleştiren sözde kod (pseudocode) verilmiştir.



Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Hızlı Sıralama (QuickSort)**

```
HoarePartition(A[l..r])
// Girdi: A dizisinin l ve r indeksleri arasındaki alt dizisi
(l < r)
// Çıktı: Pivot etrafında bölünmüş dizi, pivotun yeni konumu
döndürülür
p ← A[l]          // İlk elemanı pivot olarak al
i ← l - 1
j ← r + 1
repeat:
// Sağdan sola doğru pivot'tan küçük veya eşit bir değer bul
    repeat:
        j ← j - 1
    until A[j] ≤ p

 // Soldan sağa doğru pivot'tan büyük veya eşit bir değer bul
    repeat:
        i ← i + 1
    until A[i] ≥ p
    if i < j:
        // A[i] ile A[j] yer değiştir (swap)
        temp ← A[i]
        A[i] ← A[j]
        A[j] ← temp
    else:
        return j     // Partition noktası
```

```
Quicksort(A, l, r)
// Girdi: A dizisi ve l..r arasındaki alt dizi
// Çıktı: A[l..r] alt dizisi küçükten büyüğe
    sıralanır

    if l < r then
    s ← HoarePartition(A, l, r)   // Pivot
    etrafında böl
    Quicksort(A, l, s)       // Sol tarafı sırala
    Quicksort(A, s + 1, r)   // Sağ tarafı sırala
```

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Hızlı Sıralama (QuickSort)**

Orijinal Algoritması:

ALGORITHM $Quicksort(A[l..r])$
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow Partition(A[l..r])$ //s is a split position
$Quicksort(A[l..s-1])$
$Quicksort(A[s+1..r])$

ALGORITHM $HoarePartition(A[l..r])$
//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Divide-and-Conquer Algoritmaları

• **Hızlı Sıralama (QuickSort)**

Şimdi QuickSort'un zaman verimliliğine bakalım.

En İyi Durum (Best Case)

Pivot her zaman diziyi tam ortadan ikiye böler.

Yani her adımda dizi eşit büyüklükte iki parçaya ayrılır.

Zaman denklemi:

$$T_{\text{best}}(n) = 2 \cdot T(n/2) + n$$

Burada:

2 · T(n/2) → iki yarıyı sıralamak için gereken süre

+ n → pivotlama sırasında yapılan karşılaştırmalar (partition işlemi)

Çözüm: Bu denklem, Merge Sort ile aynıdır.

Master Theorem'e göre:

$$T_{\text{best}}(n) \in \Theta(n \log n)$$

En Kötü Durum (Worst Case)

Pivot hep en küçük veya en büyük eleman seçilirse.

Yani dizi her zaman tek tarafa bölünür → örneğin sıralı bir dizi.

$$T_{\text{worst}}(n) = T(n-1) + T(0) + n \Rightarrow T(n) = T(n-1) + n$$

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

Açarsak:

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

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

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

$$\vdots$$

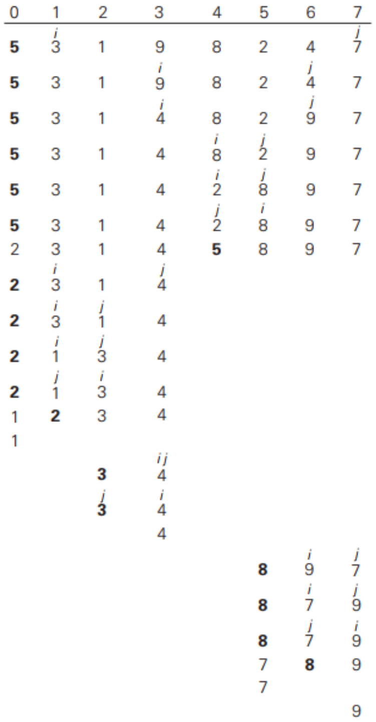
$$T(n) = T(1) + 2 + 3 + \cdots + (n-1) + n$$

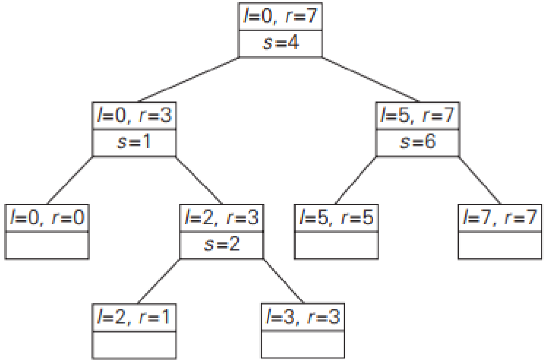
$$T_{\text{worst}}(n) = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2} \Rightarrow T(n) \in \Theta(n^2)$$

Eng: Abdulrahman Hamdi

Divide-and-Conquer Algoritmaları

- **Hızlı Sıralama (QuickSort)**





Ortalama Durum (Average Case)

Pivot rastgele seçilir, dizinin her konumu eşit ihtimalle pivot olabilir.

Amaç: Ortalama kaç karşılaştırma yapılır?

$$T_{avg}(0) = 0, \quad T_{avg}(1) = 0. \quad T_{\text{avg}}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + T(s) + T(n-1-s)]$$

Bu denklem çözümü daha zordur ama yaklaşık olarak:

$$T_{\text{avg}}(n) \approx 2n \ln n \approx 1.39 \cdot n \log_2 n \Rightarrow T(n) \in \Theta(n \log n)$$

Eng: Abdulrahman Hamdi

Algorithm Analysis And Design

Thanks for listening!

Eng: Abdulrahman Hamdi