

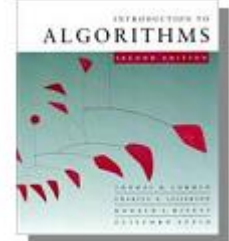
10.Hafta

Dinamik

Programlama

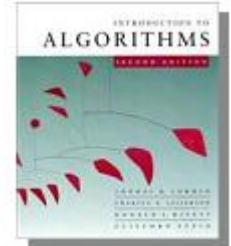
- En uzun ortak altdizi
- En uygun altyapı
- Altproblemlerin çakışması

Dinamik Programlama (Dynamic programming)

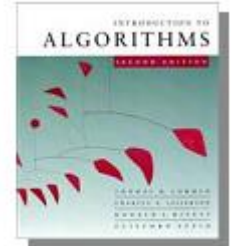


- Fibonacci sayıları örneği
- Optimizasyon problemleri
- Matris çarpım optimizasyonu
- Dinamik programlamanın prensipleri
- En uzun ortak alt sıra (Longest common subsequence)

Dinamik programlama (Dynamic programming)

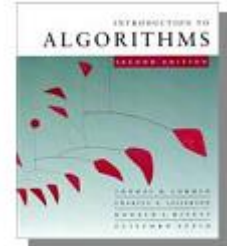


- **Tarihçe:**
- Richard Bellman tarafından 1950' li yıllarda bulunmuştur. "Programming" kelimesi aslında "planning" anlamında kullanılmaktadır (bilgisayar programlama değil)
- Dinamik Programlama (DP), böl ve yönet yöntemine benzer olarak alt problemlerin çözümlerini birleştirerek çözüm elde eden bir yöntemdir.



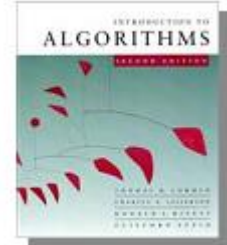
Dinamik programlama

- Böl ve yönet yönteminde alt problemlerin bağımsız olması gerekiyor; bunun tersine DP' da alt problemler bağımsız değilse de, DP uygulanabilir.
- DP her alt problemi bir kez çözer ve çözümleri bir tabloda saklar ve bu şekilde aynı alt problemin birden fazla ortaya çıkma durumunda her seferinde tekrar çözüm yapmaktansa, tabloda saklamış olduğu değeri problemde yerine koyar. Bu şekilde işlemlerin çözümünün hızlanması sağlanmış olur.
- DP, genelde en iyileme (optimizasyon) problemlerine uygulanır. Bu tip problemlerin birden fazla çözümü olabilir. Amaç bu çözümler içinde en iyisini bulmaktır.



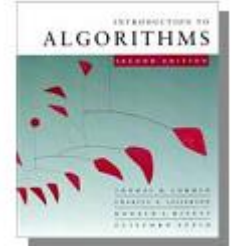
Dinamik programlama

- Dinamik programlama uygulamalarında temel olarak 3 teknikten faydalanılır.
 - 1- Çözümü aynı olan alt-problemler
 - 2- Büyük bir problemi küçük parçalara bölmek ve bu küçük parçaları kullanarak baştaki büyük problemimizin sonucuna ulaşmak
 - 3- Çözdüğümüz her alt-problemin sonucunu bir yere not almak ve gerektiğinde bu sonucu kullanarak aynı problemi tekrar tekrar çözmeyi engellemek.
- Genel olarak dinamik programlama kullanırken aşağıdaki dört adımı göz önüne almamız gerekir ve dört adım DP' nin temelini oluştururlar.



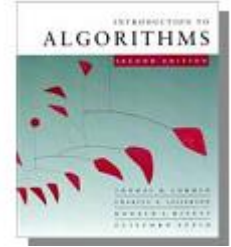
Dinamik Programlama

- 1 . Optimal çözümün yapısının karakteristiği ortaya çıkarılmalı.
- Optimal altyapısının gösterimi -bir optimal sonuç, alt problemlerin optimal sonuçlarını içerir
 - Bir problemin çözümü:
 - Birçok olası çözümünden birisi seçilir
 - Seçilen çözüm bir veya daha fazla alt problemin çözümünün sonucudur.
 - Tüm çözümün optimal olması için alt problemlerin çözümlerinin de optimal olması zorunludur



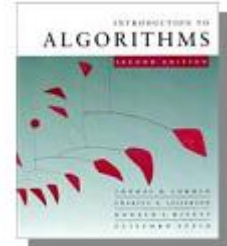
Dinamik Programlama

- 2. Özyinelemeli olarak optimal çözümün değerini tanımlamalı.
 - Optimal çözümün değeri için bir öz yineleme (recurrence) ifade yazılır.
 - $M_{opt} = \text{Min}_k$ 'nın tüm değerleri için
- 3. Optimal çözümün değeri bottom-up(alttan-üste) yaklaşımıyla çözülür, böylece alt problemlerin daha önce çözülmesi gereklidir (veya memoization kullanılır)
 - Eğer bazı alt problemlerin çözümlerinin tutulmamasıyla alan gereksinimi azaltılabiliyorsa bu alt çözümlerin sonuçları hafızaya alınmamalıdır



Dinamik Programlama

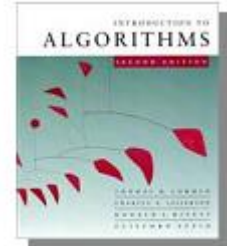
- 4. Elde edilen bilgilerden (optimal çözüme ulaşmak için yapılan seçimlerin sırasıdır) optimal çözüm yapılandırılır.
- Eğer problemin bir tane çözümü varsa, bu durumda bu adım atlanabilir.



Algoritma tasarım teknikleri

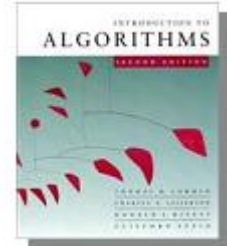
- Şimdiye kadar görülenler:
- Iterative (brute-force) algoritmalar
 - Örnek, insertion sort
- Diğer veri yapılarını kullanan algoritmalar
 - Örnek, heap sort
- Divide-and-conquer algoritmaları
 - Örnek, binary search, merge sort, quick sort

Böl ve Fethet (Divide and Conquer)-Hatırlatma



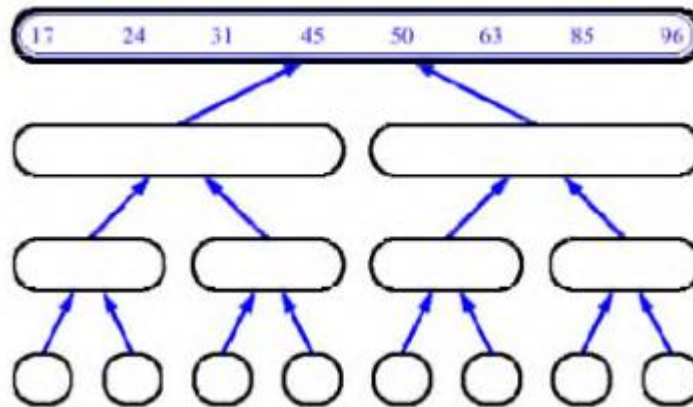
- Böl ve Fethet methoduyla algoritma tasarımı:
- **Böl (Divide):** Eğer giriş boyut çok büyükse, problemi iki veya daha fazla birleşik alt probleme böl.
- **Fethet (Conquer):** Alt problemleri çözmek için böl ve fethet metodunu özyinelemeli olarak kullan
- **Birleştir (Combine):** Orjinal problemin çözümünü oluşturmak için alt problemlerin çözümlerini birleştir "merge".

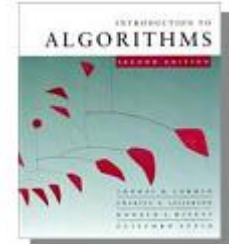
Böl ve Fethet (Divide and Conquer)-Hatırlatma



- Örnek, MergeSort
- Alt problemler bağımsızdırlar

```
Merge-Sort (A, p, r)
  if p < r then
    q ← (p+r) / 2
    Merge-Sort (A, p, q)
    Merge-Sort (A, q+1, r)
    Merge (A, p, q, r)
```



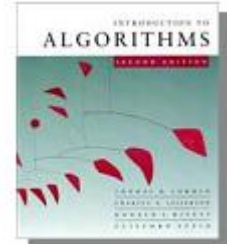


Fibonacci Sayıları

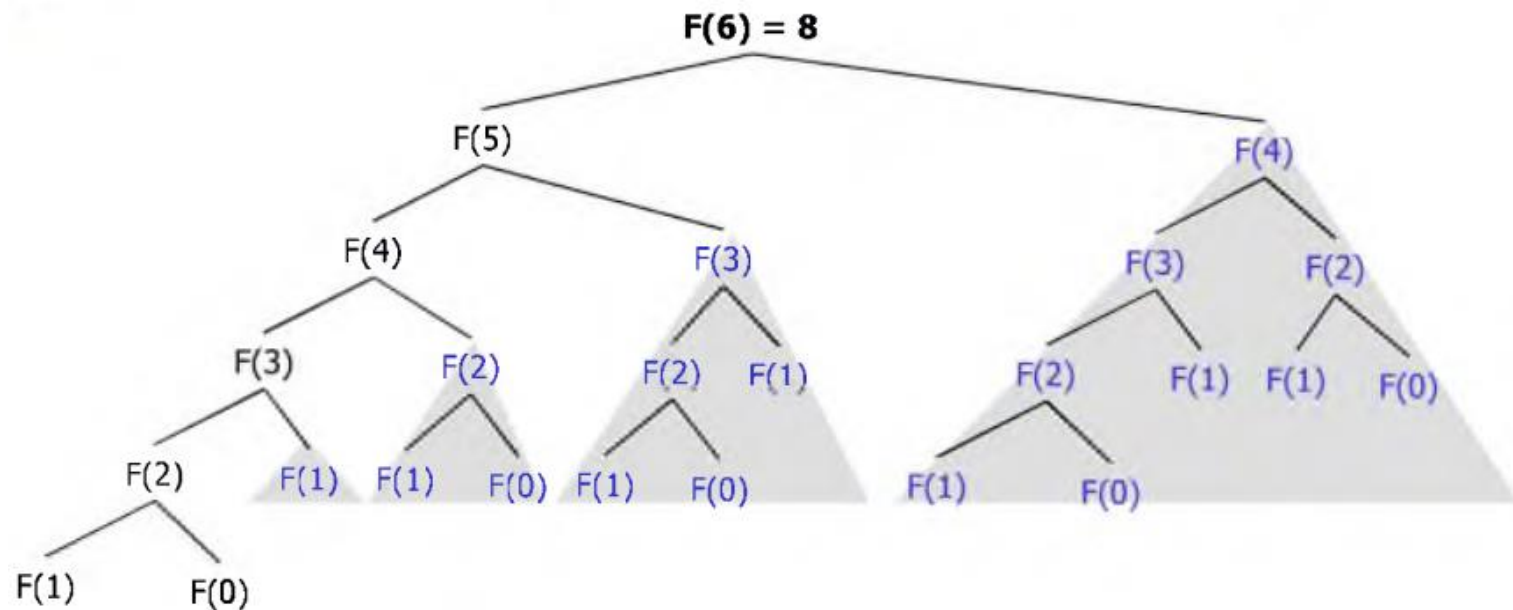
- $F(n) = F(n-1) + F(n-2)$ seri
- $F(0) = 0$, $F(1) = 1$ başlangıç değerleri
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ... seri açılım
- Bu serinin herhangi bir elemanını hesaplamak için iki farklı şekilde çözüm yapılabilir. Bunlardan biri özyineleme yöntemi ile ve diğeri de dinamik programlama mantığı ile yapılabilir.
- Recursive

```
FibonacciR(n)
01 if  $n \leq 1$  then return n
02 else return FibonacciR(n-1) + FibonacciR(n-2)
```

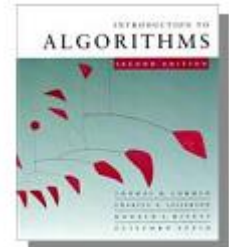
- Recursive işlem çok yavaştır! $T(n) \geq T(n-1) + T(n-2) + \Theta(1)$
- Niçin ? Nasıl yavaş olur ?



Fibonacci Sayıları

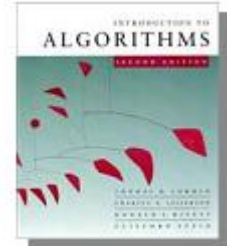


- Her seferinde $F(n)$ sayısından küçük olan aynı değerler tekrar tekrar hesaplanmaktadır!



Karakteristik denklemler kullanarak özyinelemeleri çözme

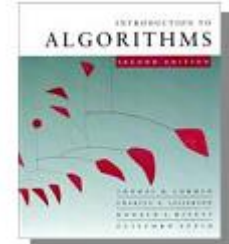
- **Örnek:** Fibonacci özyinelemeli bağıntı
- $t_0 = 0$ ve $t_1 = 1$ ve $n \geq 2$ olarak verildiğine göre
- $t_n - t_{n-1} - t_{n-2} = 0$ için yinelemeli bağıntıyı çözünüz
- Karakteristik denklem: $x^2 - x - 1 = 0$
- Kökler $x_1 = \frac{1+\sqrt{5}}{2}$ ve $x_2 = \frac{1-\sqrt{5}}{2}$, kökler eşit değil. ($\frac{1+\sqrt{5}}{2}$, altın oran)
- Durum 1'i kullanılacak. $t_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$
- $t_n(0) = 0 = c_1 + c_2, t_1 = 1 = c_1 \cdot \frac{1+\sqrt{5}}{2} + c_2 \frac{1-\sqrt{5}}{2}$
- $c_1 = 1/\sqrt{5}$ ve $c_2 = -1/\sqrt{5}$ olarak bulunur.
- Bu değerleri yerine yazarsak
- $t_n = 1/\sqrt{5} \left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right]$ olarak bulunur ve sonuç olarak
- $t_n \in \theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$



Fibonacci Sayıları

- $T(n) \cong \Theta(1.6^n)$ olur. Çalışma süresi **exponential** olur!
- Hesaplanan her terim, gerekli olduğu her yerde tekrar hesaplanmak yerine kullanıldığında bu yavaşlık ortadan kalkacaktır. Alt problemlerin çözümlerinin hafızada tutulmasıyla $F(n)$ 'i doğrusal çalışma süresiyle çözebiliriz – **dynamic programming**
- Çözüm bottom-up yaklaşımıyla çözülür
- Bu işlemi gerçekleştiren algoritma;

Fibonacci Sayıları

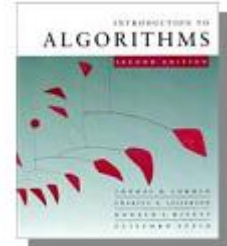


Fibonacci(n)

▷ n hesaplanacak olan terimin numarasını gösterir.

1. Eğer $n \leq 1$ ise
2. Sonuç $\leftarrow 1$
3. değilse
4. Son $\leftarrow 1$
5. Son2 $\leftarrow 1$
6. Cevap $\leftarrow 1$
7. $i \leftarrow 2, \dots, n$ kadar
8. Cevap \leftarrow Son + Son2
9. Son2 \leftarrow Son
10. Son \leftarrow Cevap
11. Sonuç \leftarrow Cevap

- Gerçekte, herhangi bir iterasyonda sadece son iki değer hafızada tutulması yeterlidir.
- Her adım için maliyet $\Theta(1)$ ise en kötü durum için yani n tane elaman için $T(n) = \Theta(n)$ olur. Çalışma süresi **doğrusal** olur!



Binom Katsayıları

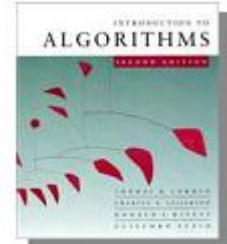
- Binom, polinomların özel hali, iki terimli polinom demek oluyor. Örnek verecek olursak. $(x+y)$, $(x-5)$, $(3x^2+5y)$ gibi iki terimli ifadeler binom diyoruz. Binom açılımı ile, $(x+y)^n$ ifadesinin tek tek terimlerin neler olduğunu gösteren formül kastediliyor.

Örneğin:

$$(x+y)^2 = x^2 + 2xy + y^2$$

$$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$$

ifadelerinde hem her terimin katsayısı, hem de değişkenlerin üslerini hesaplayan formülün adı 'binom teoremi', 'binom formülü', terimlerin katsayıları da 'binom katsayıları' adlarıyla anılırlar. Binom açılımı birkaç değişik şekilde ifade edilebilir:



Binom Katsayıları

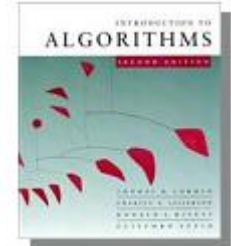
- **Pascal Üçgeni:** Aşağıda görülen üçgene bakarak

```

      1
     1 2 1
    1 3 3 1
   1 4 6 4 1
  1 5 10 10 5 1

```

- $(x+y)^5$ ifadesini $x^5+5x^4y+10x^3y^2+10x^2y^3+5xy^4+y^5$ olarak açabiliyoruz. Ancak, $(x+y)^{18}$ gibi bir açılım yapmak istediğimizde, gerçekten büyük bir üçgen kurmak zorunda kalacağımızı da göz önünde tutalım.



Binom Katsayıları

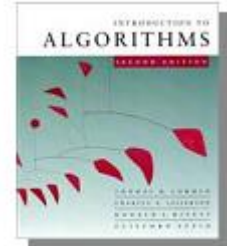
- Binom Teoremi:

$$(x+y)^n = x^n + nx^{n-1}y + n(n-1)x^{n-2}y^2/2! + n(n-1)(n-2)x^{n-3}y^3/3!$$

- $+ \dots + n(n-1)(n-2)\dots(n-k)x^{n-k}y^k/k! + \dots + nxy^{n-1} + y^n$

- $(x+y)^n = \sum_{k=0}^n [n!/k!(n-k)!]x^{n-k}y^k$ şeklinde kapalı olarak yazılabilir.

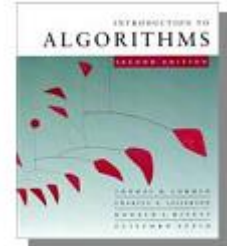
- ! İşareti faktöriyel işaretidir ve $k! = 1*2*3*\dots*(k-1)*k$ anlamını taşır.



Binom Katsayıları

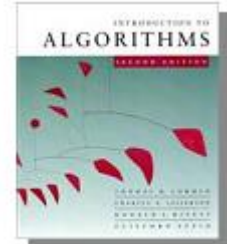
- Binom katsayıları hesaplama, $C(k,n)$ $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
- $C(k,n)$, toplam n nesneden mümkün k nesne seçenekleri sayıyor
- Özellikleri:
 - $C(k,n) = n! / k!(n-k)!$ ($n! = 1 * 2 * 3 * \dots * (n-1) * n \rightarrow$ faktöriyel)
 - $C(k,n) = C(k,n-1) + C(k-1,n-1)$ (altyapı denklemi)
 - $C(0,n) = 1; C(n,n) = 1$
- Büyük k ve n için, faktöriyel hesaplama zor olur, ve $C(k,n) = C(k,n-1) + C(k-1,n-1)$ formül kullanılır.

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$$



Binom Katsayıları

- Direkt $C(k,n)=C(k,n-1)+C(k-1,n-1)$ kullanma çok hesaplama gerekiyor, ama dinamik programlama için gereken bütün durumlar var:
 - Optimum altyapı var – $C(k,n)$, daha küçük k ve n C -katsayıları kullanarak hesaplanır
 - (yani – $C(k,n)=C(k,n-1)+C(k-1,n-1)$)
- Örtüşen altproblemler özelliği var – C katsayılarının hepsi aynı şekilde hesaplanır (yani şu formül tekrar tekrar kullanılır, $C(k,n)=C(k,n-1)+C(k-1,n-1)$)



Binom Katsayıları

$C(k,n)$ hesaplarken onları bir tabloda kaydedince, belirli $C(k,n)$, $O(kn)$ zamanla hesaplanabilir:

Sözde kodu:

$C(0,0)=1,$

döngü $i=0$ 'dan n 'e **kadar**

// döngü

döngü $j=0$ 'den $\min(i,k)$ 'ya **kadar**

//döngü

eğer $j=0$ veya $j=i$ **ise**

$C(j,i)=1$

aksi halde

$C(j,i)=C(j-1,i-1)+C(j,i-1)$

//alt problemi hesaplama

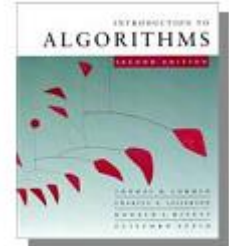
eğer sonu

//ve kaydetme

döngü sonu

döngü sonu

Binom Katsayıları



```

o class pascal_ucgen
o { public void binom()
o { // değişkenleri tanımlama, maksimum değeri int için 19 olmalı, long tipinde daha
  büyük rakamlar elde edilebilir
o   Console.Clear();
o   int[,] binom = new int[19, 19];    int n = 0;
o   for (int i = 0; i < 19 - 1; i++)
o   { binom[i, 0] = 1; binom[i + 1, i + 1] = 1; }
o   do { Console.WriteLine("Binom acilimini gormek istediginiz sayi n=");
o     n = Convert.ToInt16(Console.ReadLine());
o   } while (n < 0 || n >= 19);
o   for (int k = 0; k < n; k++)
o   { for (int j = 0; j <= k + 1; j++)
o     { if (k + 1 == j || j == 0) binom[k, j] = 1;
o       else binom[k + 1, j] = binom[k, j - 1] + binom[k, j];}
o     Console.WriteLine();
o   }
o   for (int k = 0; k < n + 1; k++)
o   { for (int j = 0; j <= k; j++){ Console.Write(binom[k, j] + "\t"); }
o     Console.WriteLine();
o   }
o }

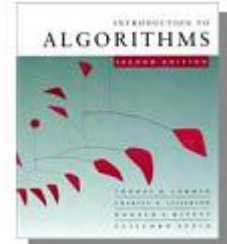
```

Binom acilimini gormek istediginiz sayi n=6

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1

```

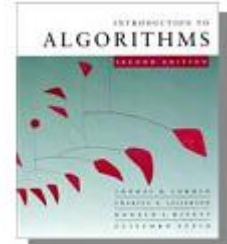


Matrislerin Çarpımı

- A - $n \times m$ ve B - $m \times k$ gibi iki matrisin çarpımından elde edilen C - $n \times k$, matrisi $n.m.k$ adet çarpma işlemi gerektirir.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \dots & \dots & \dots \\ \dots & c_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \quad c_{i,j} = \sum_{l=1}^m a_{i,l} \cdot b_{l,j}$$

- **Problem:** Çok sayıdaki matrisin çarpım sonucunun en az çarpma yaparak elde edilmesi
- Matris, çarpma işleminin birleşme (associative) özelliğine sahiptir
 - $(AB)C = A(BC)$



Matrislerin Çarpımı

- $A \times B \times C \times D$, çarpımını düşünürsek

- $A - 30 \times 1$, $B - 1 \times 40$, $C - 40 \times 10$, $D - 10 \times 25$

p0	p1	p2	p3	p4
30	1	40	10	25

- Çarpma sayısı:

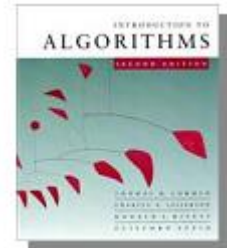
- $(AB)C)D = (30 * 1 * 40) + (30 * 40 * 10) + (30 * 10 * 25) = 20700$

- $(AB)(CD) = (30 * 1 * 40) + (40 * 10 * 25) + (30 * 40 * 25) = 41200$

- $A((BC)D) = 400 + 250 + 750 = 1400$

- Optimal parantezlemeye (işlem sırasına) ihtiyaç vardır.

$A_1 \times A_2 \times \dots \times A_n$, burada A_i $p_{i-1} \times p_i$ matristir



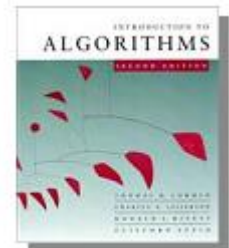
Matrislerin Çarpımı

- $A_1 A_2 A_3 A_4$ matrislerin çarpımı 5 farklı şekilde elde edilebilir. ($n=4$)

$(A_1(A_2(A_3 A_4)))$,
 $(A_1((A_2 A_3) A_4))$,
 $((A_1 A_2)(A_3 A_4))$,
 $((A_1(A_2 A_3)) A_4)$,
 $((A_1 A_2) A_3) A_4$.

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

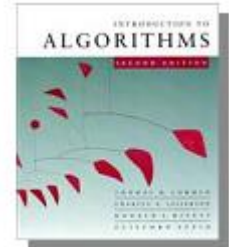
- Recurrence ifadenin çözümü, $\Omega(2^n)$ olur, $n \geq 7$ için
- $n=2$, 1
- $n=3$, 2
- $n=4$, 5
- $n=5$, 14
- $n=6$, 42
- $n=7$, 132



Matrislerin Çarpımı

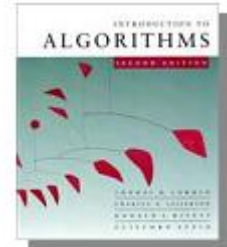
MATRIX-MULTIPLY(A, B)

```
1  if  $A.columns \neq B.rows$ 
2      error “incompatible dimensions”
3  else let  $C$  be a new  $A.rows \times B.columns$  matrix
4      for  $i = 1$  to  $A.rows$ 
5          for  $j = 1$  to  $B.columns$ 
6               $c_{ij} = 0$ 
7              for  $k = 1$  to  $A.columns$ 
8                   $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
9      return  $C$ 
```



Matrislerin Çarpımı

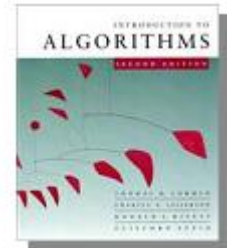
- Bu algoritma sadece optimal olarak kaç tane skaler çarpım yapılacağını hesaplar. Matrislerin çarpımını yapmaz.
- Dinamik programlamanın dördüncü adımında olduğu gibi matris çarpımını yapan optimal çarpım algoritması hazırlanır.



Matrislerin Çarpımı

- $m(i, j)$, $\prod_{k=i}^j A_k$ çarpımlarını hesaplamak için gerekli olan **minimum** çarpma sayısını gösterebilir.
- **Çıkarımlar**
 - (i, j) matrislerinin belirlenen bir k ($i \leq k < j$) değeri için en dıştaki parantezleri şu şekilde ifade edilebilir:
 $(A_i \dots A_k)(A_{k+1} \dots A_j)$
 - (i, j) matrislerini optimal parantezlemek için k 'nın her iki tarafının da optimal parantezlenmesi gerekir.
 - Minimum çarpma işlemi ifadesi

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

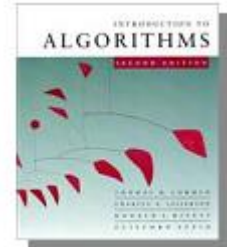


Matrislerin Çarpımı

- Bütün olası k değerleri için denemeler yapılabilir. Özyinelemeli (Recurrence) ifade:

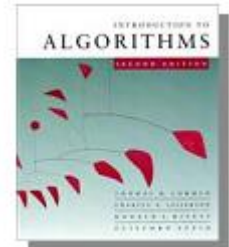
$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

- Doğrudan recursive çalışma exponential çalışma süresine sahiptir - birçok işlem tekrar tekrar yapılır



Matrislerin Çarpımı

- Optimal $m(i, j)$ için $O(n^2)$ kadar alana ihtiyaç duyulur ve $m[1..n, 1..n]$ iki boyutlu dizisinin yarısı kullanılır.
- $m(i, i) = 0, 1 \leq i \leq n$
- Alt problemlerin çözümü büyüyen şekilde yapılır: önce 2 boyutunda alt problemler, sonra 3 boyutunda alt problemler ve 4, 5... şeklinde devam eder.
- Her (i, j) çifti için optimal parantezlemenin elde edilmesi için $s[i, j] = k$ değeri kayıt edilir ve problem (i, k) ve $(k+1, j)$ olarak iki alt probleme bölünür.



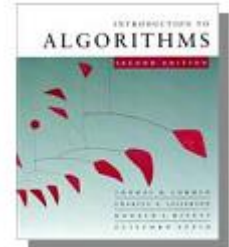
Matrislerin Çarpımı

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

MATRIX-CHAIN-ORDER(p)

```

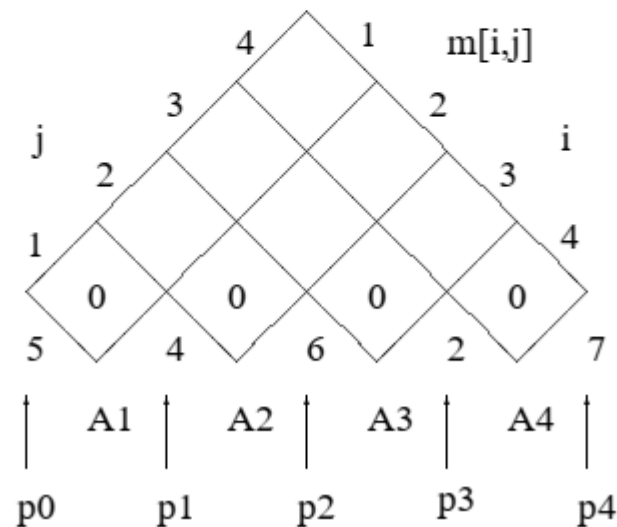
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n-1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$            //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

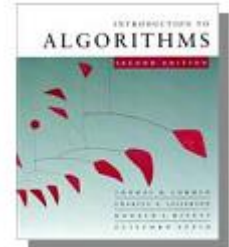
Dynamic Programming: Minimum Weight Triangulation

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4 , with $p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2$ and $p_4 = 7$. Find $m[1, 4]$.

S0: Initialization



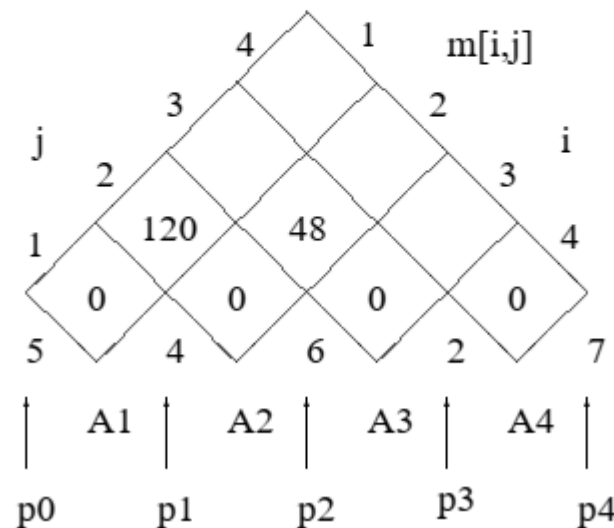
Stp 1: Computing $m[1, 2]$ By definition

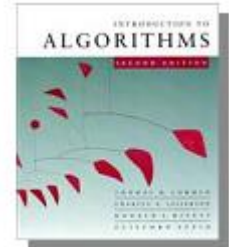


Dynamic Programming: Minimum Weight Triangulation

Stp 2: Computing $m[2, 3]$ By definition

$$\begin{aligned}
 m[2, 3] &= \min_{2 \leq k < 3} (m[2, k] + m[k + 1, 3] + p_1 p_k p_3) \\
 &= m[2, 2] + m[3, 3] + p_1 p_2 p_3 = 48.
 \end{aligned}$$

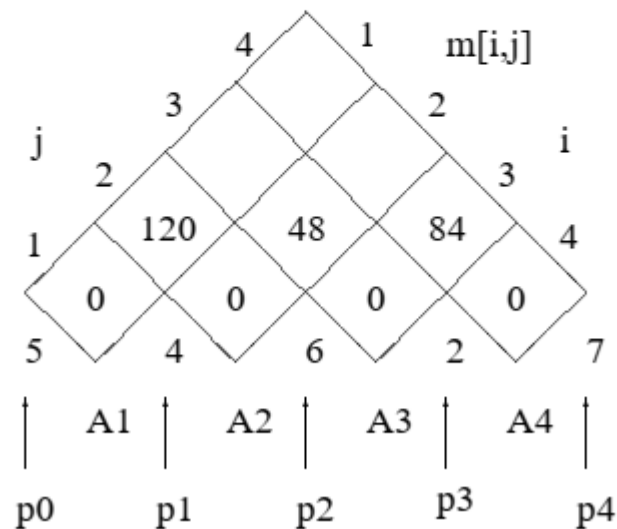


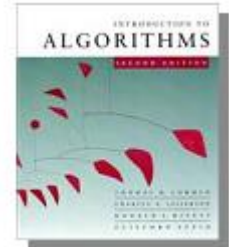


Dynamic Programming: Minimum Weight Triangulation

Step3: Computing $m[3, 4]$ By definition

$$\begin{aligned}
 m[3, 4] &= \min_{3 \leq k < 4} (m[3, k] + m[k + 1, 4] + p_2 p_k p_4) \\
 &= m[3, 3] + m[4, 4] + p_2 p_3 p_4 = 84.
 \end{aligned}$$

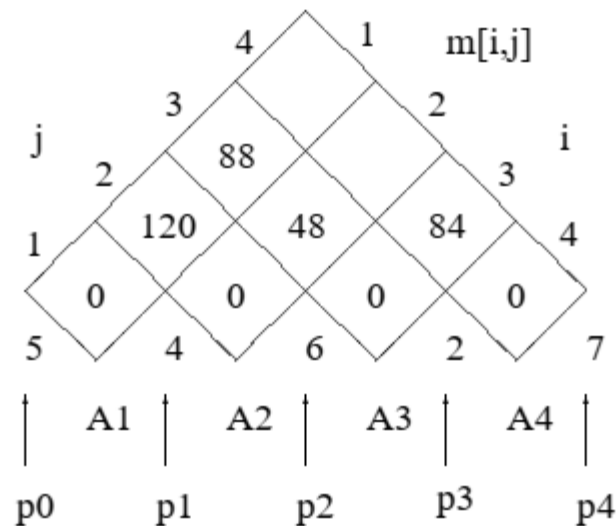


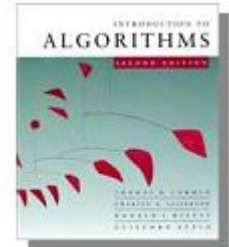


Dynamic Programming: Minimum Weight Triangulation

Step4: Computing $m[1, 3]$ By definition

$$\begin{aligned}
 m[1, 3] &= \min_{1 \leq k < 3} (m[1, k] + m[k + 1, 3] + p_0 p_k p_3) \\
 &= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 3] + p_0 p_1 p_3 \\ m[1, 2] + m[3, 3] + p_0 p_2 p_3 \end{array} \right\} \\
 &= 88.
 \end{aligned}$$

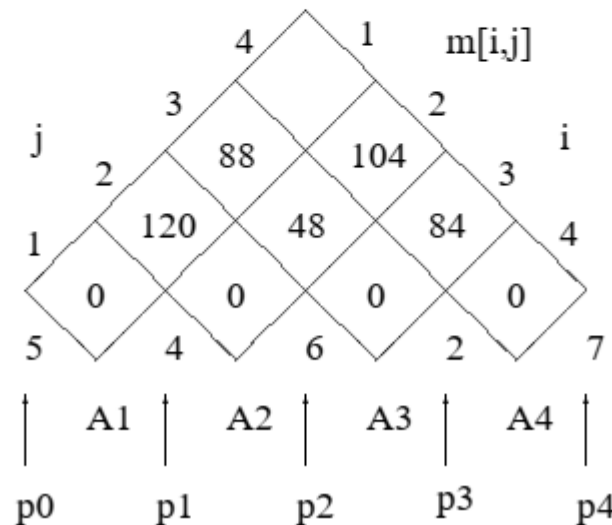


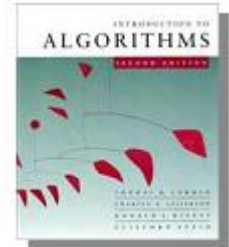


Dynamic Programming: Minimum Weight Triangulation

Step5: Computing $m[2, 4]$ By definition

$$\begin{aligned}
 m[2, 4] &= \min_{2 \leq k < 4} (m[2, k] + m[k + 1, 4] + p_1 p_k p_4) \\
 &= \min \left\{ \begin{array}{l} m[2, 2] + m[3, 4] + p_1 p_2 p_4 \\ m[2, 3] + m[4, 4] + p_1 p_3 p_4 \end{array} \right\} \\
 &= 104.
 \end{aligned}$$

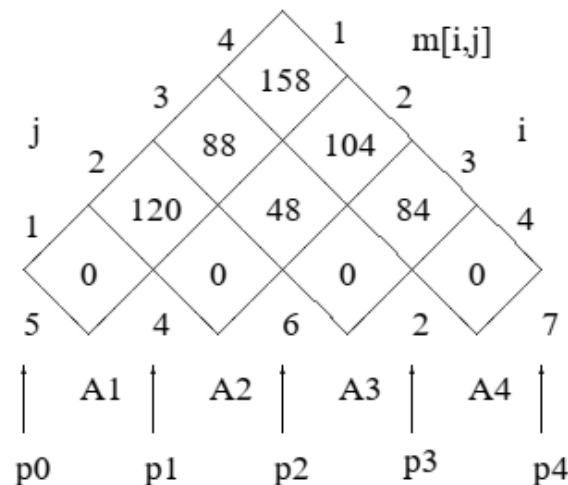


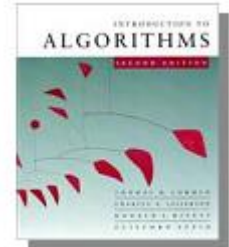


Dynamic Programming: Minimum Weight Triangulation

St6: Computing $m[1, 4]$ By definition

$$\begin{aligned}
 m[1, 4] &= \min_{1 \leq k < 4} (m[1, k] + m[k + 1, 4] + p_0 p_k p_4) \\
 &= \min \left\{ \begin{array}{l} m[1, 1] + m[2, 4] + p_0 p_1 p_4 \\ m[1, 2] + m[3, 4] + p_0 p_2 p_4 \\ m[1, 3] + m[4, 4] + p_0 p_3 p_4 \end{array} \right\} \\
 &= 158.
 \end{aligned}$$





Dynamic Programming: Minimum Weight Triangulation

A_1 (5×4)

A_2 (4×6)

A_3 (6×2)

A_4 (2×7)

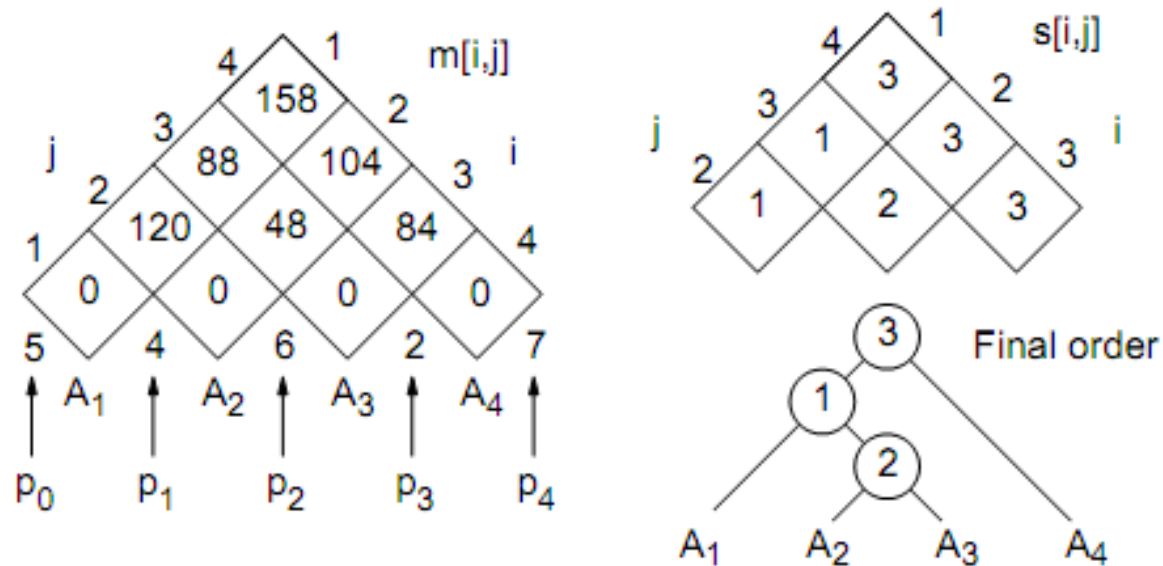
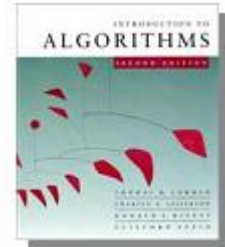


Fig. 9: Chain Matrix Multiplication Example.



Matrislerin Çarpımı

MATRIX-CHAIN-ORDER(p)

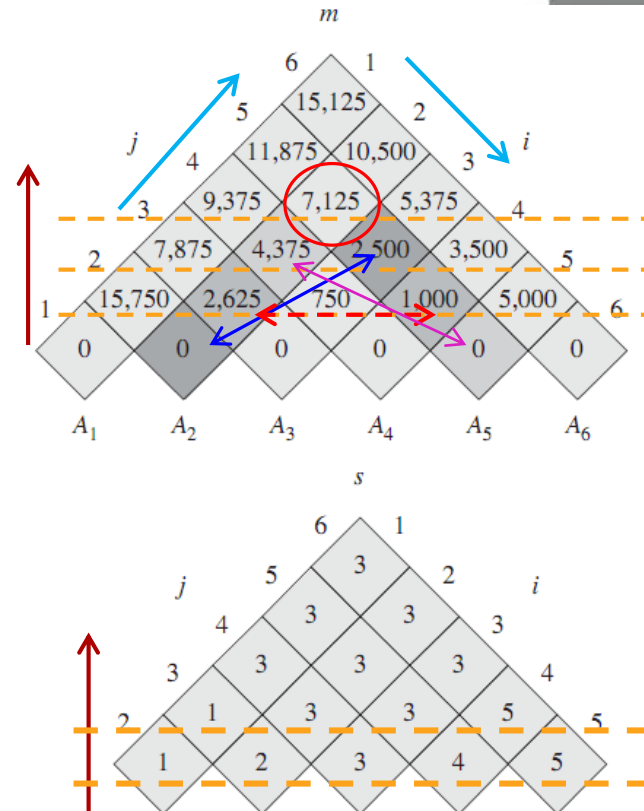
```

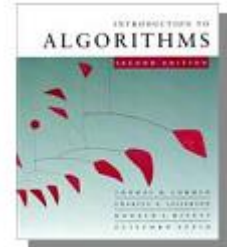
1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l \equiv 2$  to  $n$  //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 

```

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$$\begin{aligned}
 m[2, 5] = \min & \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases} \\
 & = 7125.
 \end{aligned}$$





Matrislerin Çarpımı

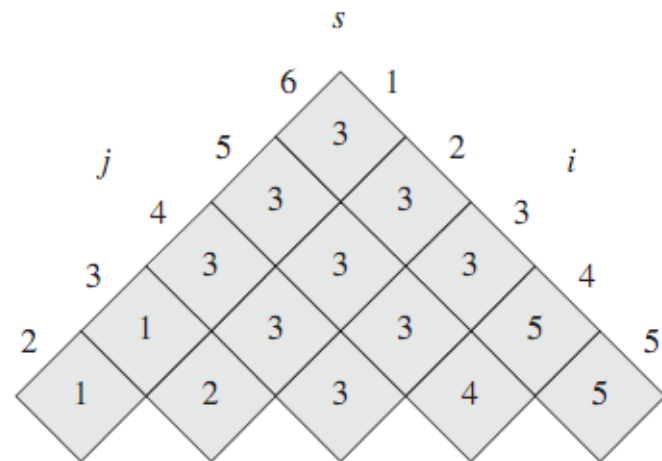
- İç içe **3** tane for döngüsü vardır.
- Çalışma süresi **$O(n^3)$** olur.
- Exponential çalışma süresinden polynomial çalışma süresine indirildi.
- Çalışmanın sonucunda : **$m[1, n]$** optimal çözüm değerini ve **s** ise optimal alt parçaları (**k** nın seçimi) bulundurur. $i=1, j=n$;

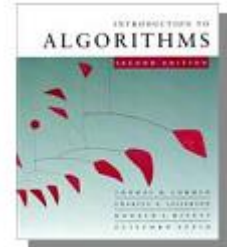
PRINT-OPTIMAL-PARENS(s, i, j)

```

1  if  $i == j$ 
2      print " $A$ " $i$ 
3  else print "("
4      PRINT-OPTIMAL-PARENS( $s, i, s[i, j]$ )
5      PRINT-OPTIMAL-PARENS( $s, s[i, j] + 1, j$ )
6      print ")"
```

$((A_1(A_2A_3))((A_4A_5)A_6))$





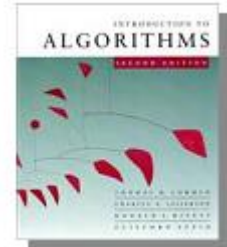
Matrislerin Çarpımı

- Matris çarpımını gerçekleştiren algoritma

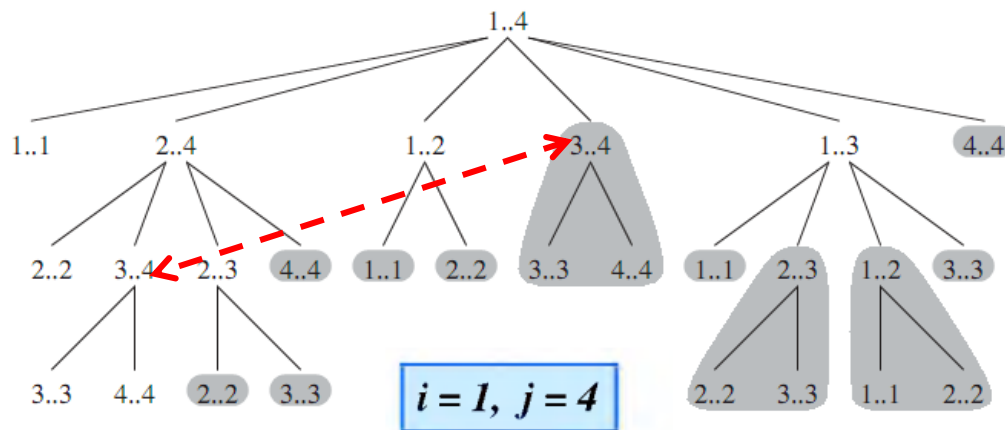
Zincirleme_matris_çarpımı(A, s, i, j)

▷ $A_1 A_2 \dots A_n$ matrislerinin optimal olarak çarpılmasını yapan bir algoritmadır. Bu işlemi gerçekleştirmek için m ve s tablolarını kullanmaktadır.

1. eğer $j > i$ ise
2. $X \leftarrow \text{Zincirleme_Matris_Çarpımı}(A, s, i, s[i, j])$
3. $Y \leftarrow \text{Zincirleme_Matris_Çarpımı}(A, s, s[i, j] + 1, j)$
4. $\text{Matris_Çarpımı}(X, Y)$



Örtüşen alt problemler Overlapping Subproblems

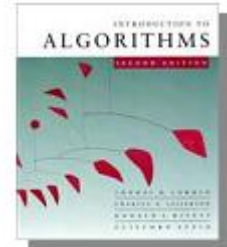


RECURSIVE-MATRIX-CHAIN(p, i, j)

```

1  if  $i == j$ 
2      return 0
3   $m[i, j] = \infty$ 
4  for  $k = i$  to  $j - 1$ 
5       $q = \text{RECURSIVE-MATRIX-CHAIN}(p, i, k)$ 
        +  $\text{RECURSIVE-MATRIX-CHAIN}(p, k + 1, j)$ 
        +  $p_{i-1}p_kp_j$ 
6      if  $q < m[i, j]$ 
7           $m[i, j] = q$ 
8  return  $m[i, j]$ 
```

Bir çok kez aynı değer
çiftleri meydana gelir.



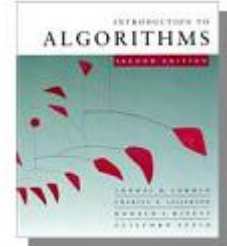
Örtüşen alt problemler

Overlapping Subproblems

- Gerçekte, $m[1..n]$ çalışma süresi n üsteli şeklindedir.
- RECURSIVE-MATRIX-CHAIN tarafından, n matris zincirinin optimal çözüm değerini hesaplamak için geçen süreyi $T(n)$ ile gösterebiliriz. O zaman, 1-2., 6-7. satırlar en az bir kez, 5.satır ise birden fazla işletilir ise;

$$T(1) \geq 1,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1.$$



Örtüşen alt problemler

Overlapping Subproblems

- $i=1,2,\dots,n-1$ için $T(i)$ bir kez, $T(k)$ bir kez ve $T(n-k)$ bir kez olarak ortaya çıkar. $n-1$ tane bölme noktası vardır ve $n-1$ tane 1 toplam içinde görünürken 1 tane 1 dışarıda görünmektedir. Buna göre denklem düzenlendiğinde

$$T(n) \geq 2 \sum_{k=1}^{n-1} T(k) + n$$

- olur. Bu durumda $T(n)=\Omega(2^n)$ olduğu kabul edilsin. Buradan

- $T(n) \geq 2^{n-1}$ olsun ve $T(1) \geq 2^0 = 1$ olur. $T(n)$ ise $T(n) \geq 2 \sum_{k=1}^{n-1} 2^{k-1} + n$

$$= 2 \sum_{k=0}^{n-2} 2^k + n$$

$$= 2(2^{n-1} - 1) + n$$

$$= 2^n - 2 + n \geq 2^{n-1}$$

- şeklinde olur.



Hatırlatma (Memoization) algoritması

Algoritma recursive olarak yapılandırılırsa:

- M elemanları ∞ olarak başlatılır ve **Lookup-Chain**(p, i, j) metodu çağrılır

MEMOIZED-MATRIX-CHAIN(p)

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  be a new table
3  for  $i = 1$  to  $n$ 
4      for  $j = i$  to  $n$ 
5           $m[i, j] = \infty$ 
6  return LOOKUP-CHAIN( $m, p, 1, n$ )

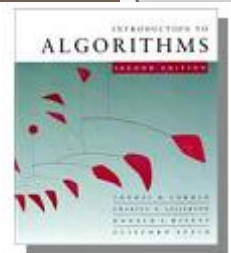
```

LOOKUP-CHAIN(m, p, i, j)

```

1  if  $m[i, j] < \infty$ 
2      return  $m[i, j]$ 
3  if  $i == j$ 
4       $m[i, j] = 0$ 
5  else for  $k = i$  to  $j - 1$ 
6       $q = \text{LOOKUP-CHAIN}(m, p, i, k)$ 
           $+ \text{LOOKUP-CHAIN}(m, p, k + 1, j) + p_{i-1}p_kp_j$ 
7      if  $q < m[i, j]$ 
8           $m[i, j] = q$ 
9  return  $m[i, j]$ 

```



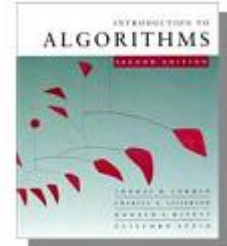
Hatırlatma (Memoization) algoritması

- **Hatırlatma:** Problemleri top-down yaklaşımla çözer, ancak alt problemlerin çözümleri bir tabloda tutulur.
- **Avantajları ve Dezavantajları :**
 - Recursion genellikle döngülerden daha yavaştır ve stack alanı kullanır
 - Anlaşılması kolaydır.
 - Eğer tüm alt problemler çözülmek zorunda değilse sadece gerekli olanlar çözülür.

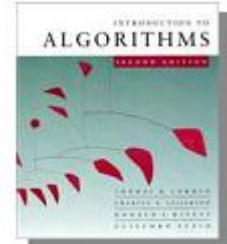
Dinamik Programlama

En uzun ortak alt dizi

(LCS-Longest Common Subsequence)



- x ve y gibi iki text string verilirse:
- Birbirlerine benzerlikleri farklı şekillerde ifade edilebilir:
 - Birisi diğerinin substring' i ise benzerdirler
 - Birini diğerine dönüştürmek için gerekli olan değişiklik azaldıkça benzerlikleri artar
 - Her ikisinden oluşturulan bir z string'in her ikisinde de aynı sırada yer alması (yan yana olması gerekmez) benzerliklerini gösterir.
 - z' nin boyutu arttıkça x ve y' nin benzerliği artar.
- Benzerliklerin ölçümünde kullanılan yöntem **LCS** olarak adlandırılır.
 - Farklı canlılar için DNA sıralamasının benzerliklerinin araştırılmasında ve Yazım kontrolü (Spell checkers) yapılmasında kullanılır.



Dinamik Programlama

Böl-ve-fethet gibi bir tasarım tekniği.

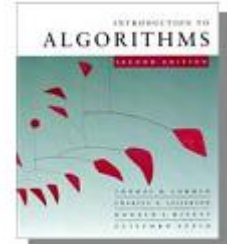
Örnek: *En uzun ortak altdizi (LCS)*

- İki tane, $x[1 \dots m]$ ve $y[1 \dots n]$ dizisi verilmiş, ikisinde de ortak olan en uzun altdiziyi bulun.

En uzun altdizi tek değildir.

$x:$ A B C B D A B

$y:$ B D C A B A



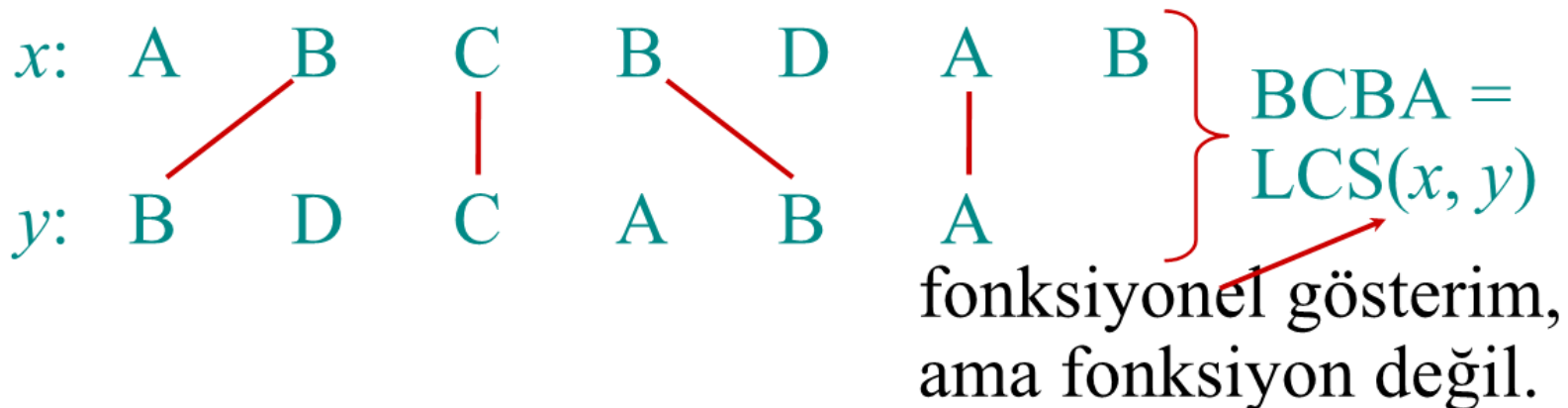
Dinamik Programlama

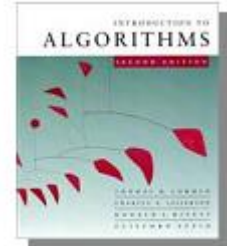
Böl-ve-fethet gibi bir tasarım tekniği.

Örnek: *En uzun ortak altdizi (LCS)*

- İki tane, $x[1 \dots m]$ ve $y[1 \dots n]$ dizisi verilmiş, ikisinde de ortak olan en uzun altdiziyi bulun.

En uzun altdizi tek değildir.



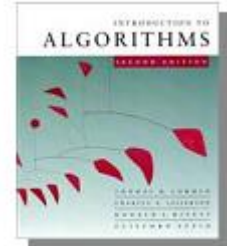


Dinamik Programlama- En uzun ortak alt dizi (LCS)

- Eğer X' te bazı karakterleri atlayarak(veya hiç atlamayarak) Z oluşturula biliyorsa Z , X' in alt dizisi olur.
 - Örnek: $X = \text{"ACGGTTA"} , Y = \text{"CGTAT"}$ ise
 - $\text{LCS}(X,Y) = \text{"CGTA"}$ veya "CGTT"
- LCS problemini çözmek için $\text{LCS}(X,Y)$ için X ve Y 'deki atlamaları oluşturmamız gerekmektedir.
- $X = \langle A, B, C, B, D, A, B \rangle , Y = \langle B, D, C, A, B, A \rangle$
- $Z = \langle B, C, B, A \rangle$
- $\text{LCS} = \langle 2, 3, 4, 6 \rangle$

Kaba-kuvvet LCS algoritması

Brute-force LCS algorithm

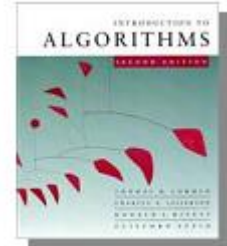


$x[1 \dots m]$ ' nin her altdizisini
 $y[1 \dots n]$ ' nin de altdizisi mi diye kontrol edin.

Analiz

- Kontrol etme = her altdizi için $O(n)$ zamanı.
- x' in 2^m sayıda altdizisi var. (m uzunluğunda her bir bit-vektörü, x' in farklı bir altdizisini belirler.)

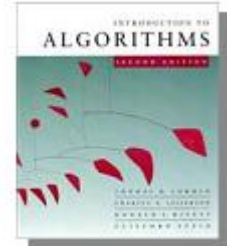
En kötü durum koşma süresi = $O(n2^m)$
= üstel zaman.



Dinamik Programlama-

LCS: Optimal'in Bulunması

- $X = \langle x_1, x_2, \dots, x_m \rangle$ ve $Y = \langle y_1, y_2, \dots, y_n \rangle$ string dizileri olmak üzere, $Z = \langle z_1, z_2, \dots, z_k \rangle$ X ve Y nin herhangi bir LCS olur.
- Eğer $x_m = y_n$ ise, bu sembol $z_k = x_m = y_n$ dizisine eklenir, ve $Z_{k-1} = \text{LCS}(X_{m-1}, Y_{n-1})$ olur. (Z_{k-1} , X_{m-1} ve Y_{n-1} 'in LCS'sidir)
- Eğer $x_m \neq y_n$ ise, $z_k \neq x_m$, $Z = \text{LCS}(X_{m-1}, Y)$ olur
- Eğer $x_m \neq y_n$ ise, $z_k \neq y_n$, $Z = \text{LCS}(X, Y_{n-1})$ olur bu durumda
 - x veya y' den bir karakter atlanır
 - $\text{LCS}(X, Y_{n-1})$ ve $\text{LCS}(X_{m-1}, Y)$ 'den büyük olan alınır.



Daha iyi bir algoritmaya doğru

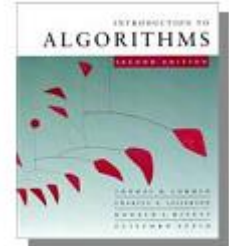
Basitleştirme:

1. En uzun ortak bir altdizinin uzunluğuna bakalım.
2. Algoritmayı LCS' yi kendisi bulacak şekilde genişletin.

Simgelem: s dizisinin uzunluğunu $|s|$ ile belirtin.

Strateji: x ve y 'nin örneklerini düşünün.

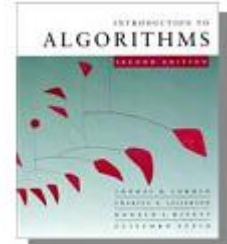
- Define(tanımlama) $c[i, j] = |\text{LCS}(x[1 \dots i], y[1 \dots j])|$.
- Then(sonra), $c[m, n] = |\text{LCS}(x, y)|$.



LCS: Özyineleme

• $c[i,j] = \text{LCS}(x_i, y_j)$ olursa

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

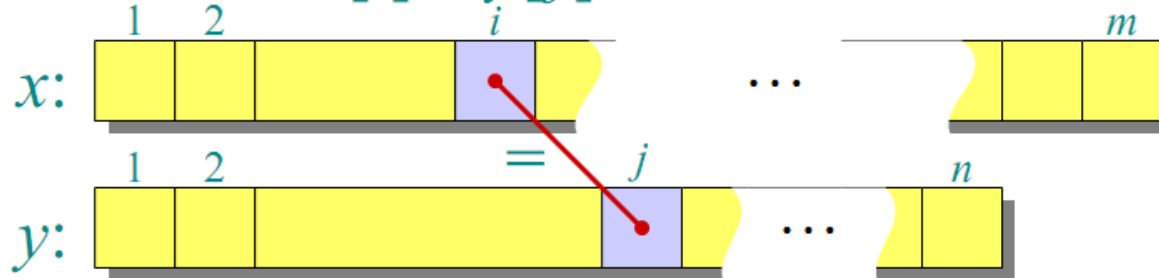


Özyinelemeli formülleme

Teorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{(eğer) if } x[i] = y[j] \text{ ise,} \\ \max \{c[i-1, j], c[i, j-1]\} & \text{aksi takdirde.} \end{cases}$$

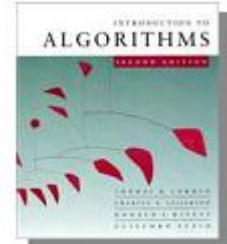
Kanıt. Durum $x[i] = y[j]$:



$c[i, j] = k$ iken $z[1 \dots k] = \text{LCS}(x[1 \dots i], y[1 \dots j])$ olsun.

Sonra, $z[k] = x[i]$, ya da başka z genişletilebilir.

Böylece, $z[1 \dots k-1]$, $x[1 \dots i-1]$ ve $y[1 \dots j-1]$ 'nin CS'sidir.



Kanıt (devamı)

İddia: $z[1 \dots k-1] = \text{LCS}(x[1 \dots i-1], y[1 \dots j-1])$.

Diyelim ki w , $x[1 \dots i-1]$ ve $y[1 \dots j-1]$ 'nin daha uzun bir CS'si, yani $|w| > k-1$. Sonra, **kes ve**

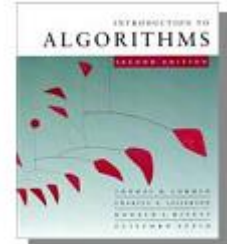
yapıştır: $w \parallel z[k]$ (w , $z[k]$ ile bitleştirilmiş.)

$x[1 \dots i]$ ve $y[1 \dots j]$ 'nin ortak altdizisidir ve $|w \parallel z[k]| > k$ dır. Çelişki, iddiayı kanıtlar.

Böylece, $c[i-1, j-1] = k-1$ 'dir ki bu $c[i, j] = c[i-1, j-1] + 1$ anlamına gelir.

Diğer durumlar benzerdir. ■

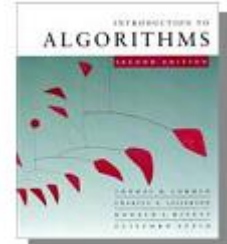
Dinamik-programlama kalite işareti #1



Optimal altyapı

*Bir probleme optimal çözüm
(örnek) alt-problemlere optimal
çözümler içerir.*

Eğer $z = \text{LCS}(x, y)$ ise, z nin her öneki x' in ve y' nin öneklerinin LCS' sidir.



LCS için özyinelemeli algoritma

$\text{LCS}(x, y, i, j)$

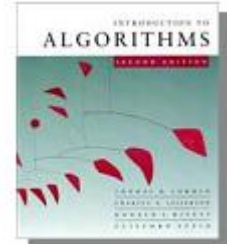
(eğer) **if** $x[i] = y[j]$

(sonra) **then** $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$

(başka) **else** $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$

return $c[i, j]$

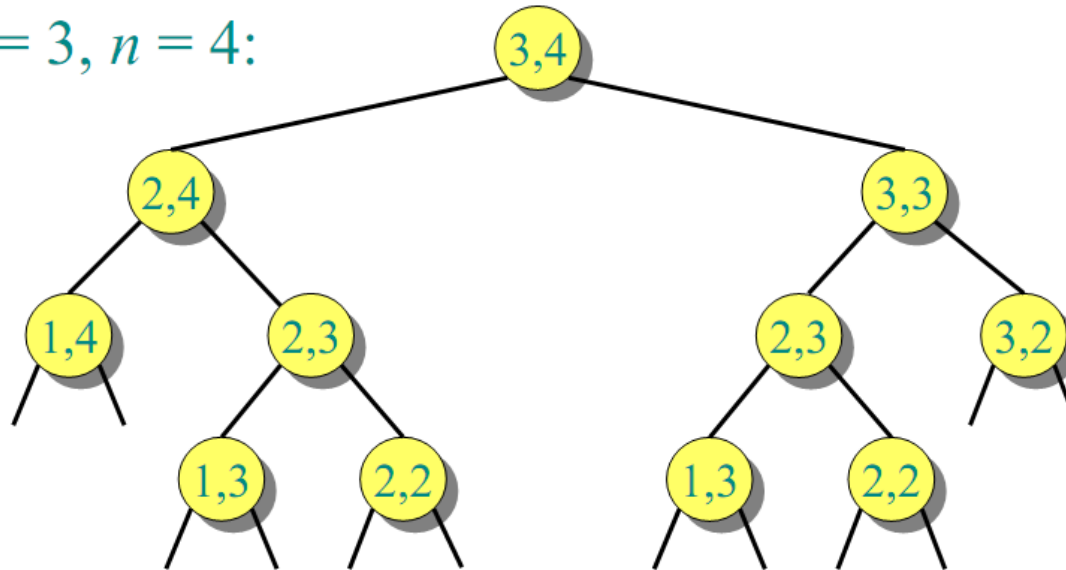
En kötü durum: $x[i] \neq y[j]$, bu durumda algoritma, tek parametrenin azaltıldığı iki alt-problemi değerlendirir.



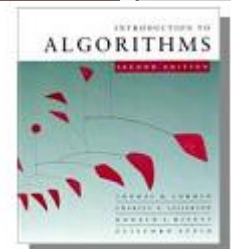
Özyineleme ağacı

- Burada bu problem için ne olduğunu anlamamıza yarayacak öz yineleme ağacı çizilir. (m ve n örnek ağaç için en kötü durumda yürütmek için verilen değerleridir.)

$m = 3, n = 4$:

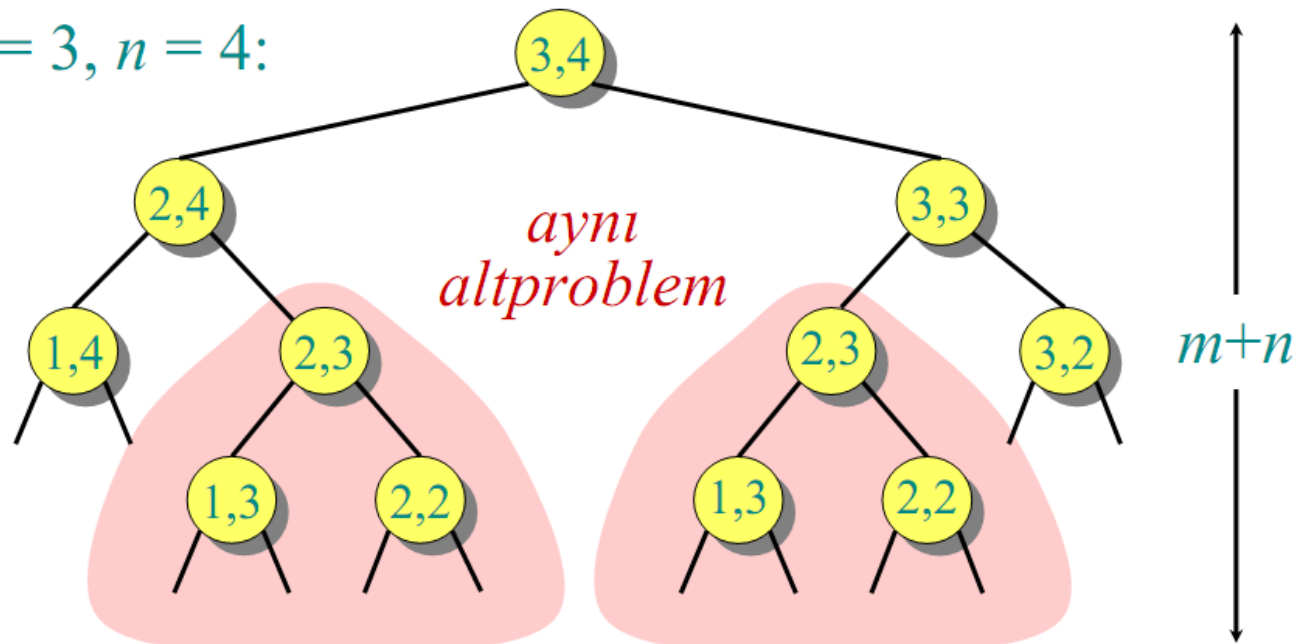


- Ağacın yüksekliği m ve n cinsinden ne olur?



Özyineleme ağacı

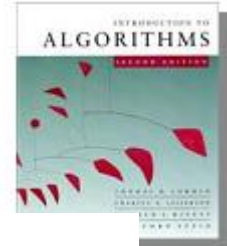
$m = 3, n = 4$:



Yükseklik = $m + n \Rightarrow$ iş potansiyel olarak üsteldir,
ama biz zaten çözülmüş olan altproblemleri çözüyoruz! (2^{m+n})

2^{m+n} aynı zamanda toplam problem sayısıdır. Bir birinden farklı kaç alt problem var?

Dinamik-programlama kalite işareti #2

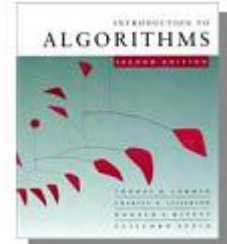


Altproblemlerin çakışması

Özyinelemeli bir çözüm, az sayıda birbirinden farklı altproblemin birçok kere tekrarlanmasını içerir.

m ve n uzunluklarının 2 dizgisi için farklı LCS altproblemlerinin sayısı mn ' dir.

- Bu 2^m ve 2^n den daha küçüktür. Çünkü i ve j ile karakterize ediliyor. $i = \langle 1..m \rangle$, $j = \langle 1..n \rangle$



Hatırlama Algoritması

Memoization(Hatırlama): Bir altprobleme ait çözümü hesapladıktan sonra onu bir tabloda depolayın. Ardışık çağrılar, işlemi tekrar yapmamak için tabloyu kontrol eder.

```

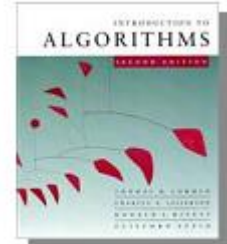
LCS( $x, y, i, j$ )
  (eğer) if  $c[i, j] = \text{NIL}$ 
  (sonra eğer) then if  $x[i] = y[j]$ 
    (sonra) then  $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$ 
    (başka) else  $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}$ 
  return  $c[i, j]$ 
  
```

} *önceki gibi*

Eğer $c[i, j] = \text{nil}$ ise hesaplıyoruz aksi taktirde hesaplamadan $c[i, j]$ geri döndürüyoruz.

Süre = $\Theta(mn)$ = her tablo girişi için sabit miktarda iş.
Yer(alan) = $\Theta(mn)$.

Dinamik-programlama algoritması



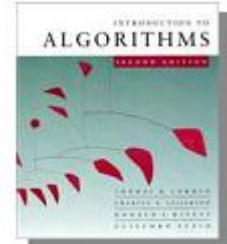
Fikir:

Aşağıdan yukarıya tabloyu hesaplayın.

$$\text{Süre} = \Theta(mn).$$

Başlangıçta sıfır uzunluğunda bir dizi olduğundan ilk satır ve sütunlar 0 olur

	A	B	C	B	D	A	B
B	0	0	0	0	0	0	0
D	0	0	1	1	1	2	2
C	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	3	4



Dinamik-programlama algoritması

Fikir:

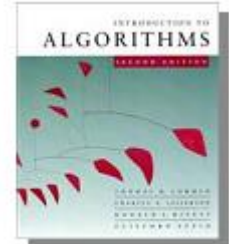
Aşağıdan yukarıya
tabloyu hesaplayın

Süre = $\Theta(mn)$.

LCS' yi, geriye
giderek
tekrar oluşturun.

Alan = $\Theta(mn)$.

	A	B	C	B	D	A	B
B	0	0	0	0	0	0	0
D	0	0	1	1	1	1	1
C	0	0	1	2	2	2	2
A	0	1	1	2	2	2	3
B	0	1	2	2	3	3	4
A	0	1	2	2	3	3	4



LCS'nin Elde Edilmesi

LCS-LENGTH(X, Y)

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = "\nwarrow"$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = "\uparrow"$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = "\leftarrow"$ 
18  return  $c$  and  $b$ 

```

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	\uparrow 0	\uparrow 0	\uparrow 0	\nwarrow 1	\leftarrow 1	\nwarrow 1
2	B		0	\nwarrow 1	\nwarrow 1	\nwarrow 1	\uparrow 1	\nwarrow 2	\leftarrow 2
3	C		0	\uparrow 1	\uparrow 1	\nwarrow 2	\nwarrow 2	\uparrow 2	\uparrow 2
4	B		0	\nwarrow 1	\uparrow 1	\uparrow 2	\uparrow 2	\nwarrow 3	\leftarrow 3
5	D		0	\uparrow 1	\nwarrow 2	\uparrow 2	\uparrow 2	\nwarrow 3	\uparrow 3
6	A		0	\uparrow 1	\uparrow 2	\uparrow 2	\nwarrow 3	\uparrow 3	\nwarrow 4
7	B		0	\nwarrow 1	\uparrow 2	\uparrow 2	\uparrow 3	\nwarrow 4	\nwarrow 4



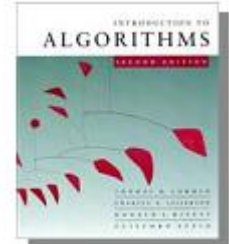
LCS'nin Yazdırılması

PRINT-LCS(b, X, i, j)

```

1  if  $i == 0$  or  $j == 0$ 
2      return
3  if  $b[i, j] == "\nwarrow"$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  elseif  $b[i, j] == "\uparrow"$ 
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )
  
```

		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
i	x_i								
0			0	0	0	0	0	0	0
1	A		0	\uparrow 0	\uparrow 0	\uparrow 0	\nwarrow 1	\leftarrow 1	\nwarrow 1
2	B		0	\nwarrow 1	\nwarrow 1	\nwarrow 1	\uparrow 1	\nwarrow 2	\leftarrow 2
3	C		0	\uparrow 1	\uparrow 1	\nwarrow 2	\nwarrow 2	\uparrow 2	\uparrow 2
4	B		0	\nwarrow 1	\uparrow 1	\uparrow 2	\uparrow 2	\nwarrow 3	\leftarrow 3
5	D		0	\uparrow 1	\nwarrow 2	\uparrow 2	\uparrow 2	\nwarrow 3	\uparrow 3
6	A		0	\uparrow 1	\uparrow 2	\uparrow 2	\nwarrow 3	\uparrow 3	\nwarrow 4
7	B		0	\nwarrow 1	\uparrow 2	\uparrow 2	\uparrow 3	\nwarrow 4	\nwarrow 4



Örnek

		Y: 0	1	2	3	4=n	
			B	D	C	B	
0		0	0	0	0	0	
1	B	0	1	1	1	1	
2	A	0	1	1	1	1	
3	C	0	1	1	2	2	
4	D	0	1	2	2	2	
m=5	B	0	1	2	2	3	

X = BACDB
Y = BDCB
LCS = BCB

LCS Length Table

		Y: 0	1	2	3	4=n	
			B	D	C	B	
0		0	0	0	0	0	
1	B	0	1	1	1	1	
2	A	0	1	1	1	1	
3	C	0	1	1	2	2	
4	D	0	1	2	2	2	
m=5	B	0	1	2	2	3	start here

with back pointers included

Fig. 6: Longest common subsequence example for the sequences $X = \langle BACDB \rangle$ and $Y = \langle BDCB \rangle$. The numeric table entries are the values of $c[i, j]$ and the arrow entries are used in the extraction of the sequence.

LCS Örnek

- $X = \text{ABCB}$
- $Y = \text{BDCAB}$
- X ve Y nin LCS (En uzun ortak alt dizisi) nedir?

$\text{LCS}(X, Y) = \text{BCB}$

$X = \text{A} \mathbf{B} \mathbf{C} \mathbf{B}$

$Y = \mathbf{B} \mathbf{D} \mathbf{C} \mathbf{A} \mathbf{B}$

LCS Örnek

								ABCB
								BDCAB
	j	0	1	2	3	4	5	
i		Y _j	B	D	C	A	B	
0	X _i							
1	A							
2	B							
3	C							
4	B							

$X = ABCB; \quad m = |X| = 4$

$Y = BDCAB; \quad n = |Y| = 5$

Tahsis edilen dizi $c[5,4]$

LCS Örnek

		ABCBA					
		BDCAB					
	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

for i = 1 to m c[i,0] = 0
 for j = 1 to n c[0,j] = 0

LCS Örnek

ABCB
BDCAB

		j					
		0	1	2	3	4	5
		Y _j	B	D	C	A	B
i	X _i						
0		0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

$\text{if } (X_i == Y_j)$
 $\quad c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Örnek

ABCB
BDCAB

i	j	Y _j	0	1	2	3	4	5
				B	D	C	A	B
0	X _i		0	0	0	0	0	0
1	A		0	0	0	0		
2	B		0					
3	C		0					
4	B		0					

$\text{if } (X_i == Y_j)$
 $\quad c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Örnek

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	
2	B		0					
3	C		0					
4	B		0					

Note: In the original image, a green circle highlights the 'A' in row 1, column 4, and an arrow points from the cell (0,4) to (1,4).

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

ABCB
BDCABB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	→ 1
2	B		0					
3	C		0					
4	B		0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i	0	0	0	0	0	0	0
1	A	0	0	0	0	0	1	1
2	B	0	0	1				
3	C	0						
4	B	0						

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

ABCB
BDCAB

i	j	Y _j	0	1	2	3	4	5
				B	D	C	A	B
0	X _i		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

Arrows indicating the path for the longest common subsequence (LCS) starting from (2,4) and moving back to (0,0):
 (2,4) → (2,3) → (2,2) → (1,4) → (0,0)

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

A^BCB
 BDCAB^B

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

Note: In the original image, the 'B' at (2,5) and the 'B' at (5,5) are circled in green. An arrow points from (1,4) to (2,5).

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

ABCB
 BDCAB

	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
i	X _i						
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	↓ 1	↓ 1			
4	B	0					

(Note: In the original image, the cell at (3,1) contains a red '1' with an arrow pointing down from the cell at (2,1). The cell at (3,2) contains a red '1' with an arrow pointing left from the cell at (3,1). The cells at (1,3) and (2,3) are circled.)

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```


LCS Örnek

ABCB
 BDCAB

	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
i	X _i						
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

Note: In the original image, the cell at (3,5) containing '2' is circled in green, and an arrow points from the cell at (2,4) containing '1' to it. The character 'C' in the header row is also circled in green.

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Örnek

ABCB
 BDCAB

	j	0	1	2	3	4	5
i		Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

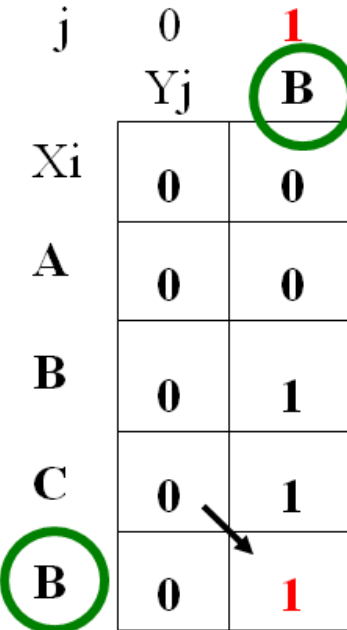
(Note: In the original image, the cell at (3,4) containing '2' has a red arrow pointing to the cell at (3,5) containing '2'. The cell at (2,5) containing '2' has a black arrow pointing down to the cell at (3,5).)

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Örnek

ABCB
BDCAB

		j	0	1	2	3	4	5
			Y _j	B	D	C	A	B
i	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				



```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
  
```

LCS Örnek

ABCB

BDCAB

	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
i	X _i						
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	→ 1	↓ 2	→ 2	

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Örnek

LCS nin Bulunması

		j	0	1	2	3	4	5
i			Y _j	B	D	C	A	B
		X _i						
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

LCS Örnek

LCS nin Bulunması

		j	0	1	2	3	4	5
i		Y _j		B	D	C	A	B
	X _i							
0			0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3



11.Hafta

Veri sıkıştırma ve Aç gözlü algoritmalar

LCS C#

```

○ using System;
○ using System.Collections.Generic;
○ using System.Linq;
○ using System.Text;

○ namespace LCS_ornek
○ { class Program
○ { int [,] c;
○   string [,] b;
○   public void Lcs(string [] x, string [] y)
○   {
○       int m = x.Length;
○       int n = y.Length;
○       c=new int[m+1,n+1];
○       b=new string [m,n];
○       for (int i = 1; i < c.GetLength(0); i++)
○           c[i, 0] = 0;
○       for (int j = 0; j < c.GetLength(1); j++)
○           c[0,j] = 0;

```

```

○       for (int i = 0; i < m; i++)
○           for (int j = 0; j < n; j++)
○               if (x[i] == y[j])
○                   {
○                       c[i + 1, j + 1] = c[i, j] + 1;
○                       b[i, j] = "Çapraz";
○                   }
○               else if (c[i, j + 1] >= c[i + 1, j])
○                   {
○                       c[i + 1, j + 1] = c[i, j + 1];
○                       b[i, j] = "Yukarı";
○                   }
○               else
○                   {
○                       c[i + 1, j + 1] = c[i+1, j];
○                       b[i, j] = "Sol";
○                   }
○       }

```


LCS C#

```

    public void Print_Lcs(string[,] b, string[] x, int
    i, int j)
    {
        if (i == 0 || j == 0)
        { Console.Write(x[i] + ", \t"); return; }
        if (b[i, j] == "Çapraz")
        {
            Print_Lcs(b, x, i - 1, j - 1);
            Console.Write(x[i] + ", \t");
        }
        else if (b[i, j] == "Yukarı")
        { Print_Lcs(b, x, i - 1, j); }
        else
        { Print_Lcs(b, x, i, j - 1); }
    }

    static void Main(string[] args)
    {
        Program lcs_xy = new Program();
        string[] x = { "A", "B", "C", "B", "D", "A", "B" };
        string[] y = { "B", "D", "C", "A", "B", "A" };
        lcs_xy.Lcs(x,y);
        for (int i = 0; i < x.Length + 1; i++)
        {
            for (int j = 0; j < y.Length + 1; j++)
            {
                Console.Write(lcs_xy.c[i, j] + ", "+" \t");
                Console.WriteLine();
            }
        }
    }

```

```

    for (int i = 0; i < x.Length; i++)
    {
        for (int j = 0; j < y.Length; j++)
        {
            Console.Write(lcs_xy.b[i, j] + ", "+" \t");
            Console.WriteLine();
        }
        Console.WriteLine("\n");
        lcs_xy.Print_Lcs(lcs_xy.b, x, x.Length-1,
        y.Length-1);
    }
}
}

```