Dinamik Programlama

Bölüm 1 - Temeller

Giriş

- Belirli bir algoritma olmayıp, bir *problem çözme tekniği* dir.
- DP deki “programlama” adlandırması, günümüzde anladığımız program yazma kavramı olmayıp çizelgeleme (tablolama) anlamında kullanılmıştır.
- İsimlendirme ve ilk sistematik çalışmalar 1940 larda Richard Bellman tarafından yapılmıştır.

Kullanım

- Optimizasyon problemlerinin birçoğunda kullanılabilir.
  - Hem minimizasyon, hem maksimizasyon
- Fakat, tüm optimizasyon problemlerinde kullanılamaz.
- Optimizasyon problemleri dışında kullanımı da vardır.
  - Örneğin, özyineli bir formülün hesaplanması

Adımlar

- Optimizasyon problemlerinin çözümünde dört temel adımdan ibarettir.
  1. Optimal çözümün yapısının karakterini çıkar
  2. Optimal çözümün değerini özyineli olarak tanımla
  3. Optimal çözümün değerini aşağıdan-yukarı (*bottom-up*) hesapla
  4. Optimal çözümü hesaplanan değerleri kullanarak çıkar

Adımlar

- Dördüncü adım eğer optimal çözümün kendisi değil, sadece değeri isteniyorsa gerekli değildir.
  - Optimal çözüm ve optimal çözümün değeri farklı kavramlardır
  - Optimal çözümün değeri her zaman tektir fakat aynı değere sahip birden çok optimal çözüm olabilir

Prensip

- Optimizasyon problemi şu üç özelliğe sahipse DP kullanılır, aksi halde DP kullanılmaz.
  1. Optimal altyapı (*optimal substructure*)
  2. Bağımsız altproblemler (*independent subproblems*)
  3. Örtüşen altproblemler (*overlapping subproblems*)
- Problemde ilk iki özellik varsa fakat üçüncüsü yoksa böl-yönet (*divide-conquer*) kullanılır.

Prensip

- *Optimal altyapı* özelliği olduğunu göstermek
  - Problemin çözümü birçok seçenekten bir seçim gerektirir
  - Optimal çözüme götüren seçimin seçildiği varsayılır
  - Bu seçim geride çözülmesi gereken bir yada daha fazla altproblem bırakır
  - Optimal çözümde kullanılan altproblemlerin de optimal çözülmüş olması gerektiği gösterilir
    - kes-ve-yapıştır (*cut-and-paste*) yaklaşımı: altproblemlerin birisinin çözümünün optimal olmadığını varsay, bu çözümü kes ve onun yerine optimal çözümü yapıştır, orijinal problemin daha iyi bir çözümü olduğunu göster.
    - Yukarıdaki bir çelişki yaratır, çünkü orijinal problemin çözümünün optimal olduğundan yola çıkarak ondan daha iyi bir çözümün olduğunu göstermiş olduk. Bu çelişkiye sebep olan tek varsayım “altproblemin çözümünün optimal olmadığı” yanlıştır. Yani, “altproblemin çözümü de optimal” dir doğrudur.

Prensip

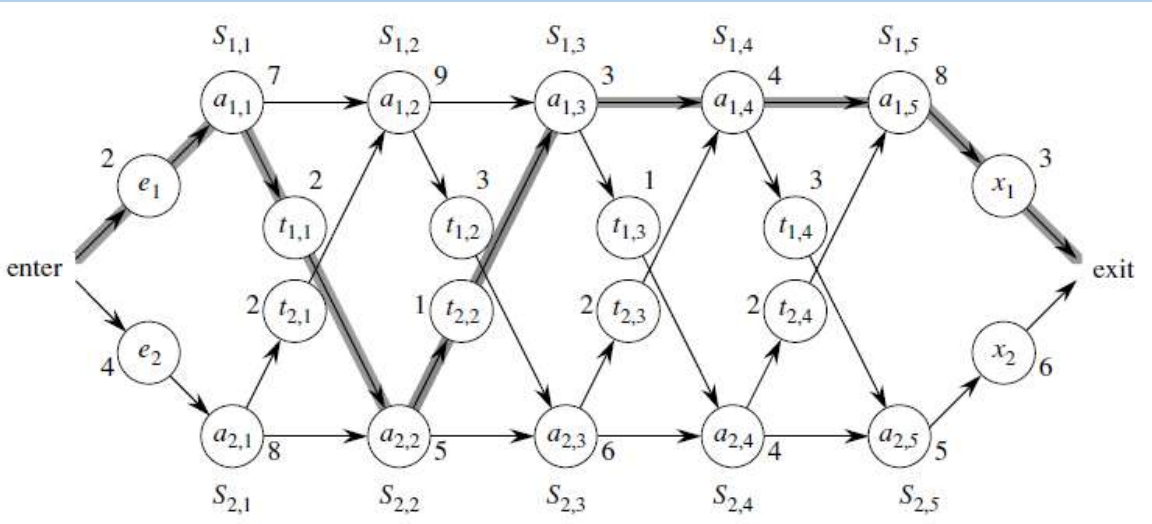
- *Bağımsız altproblemler* özelliği olduğunu göstermek
  - Optimal çözümde kullanılan altproblemlerin birinin optimal çözümünün diğerlerinin optimal çözümünü etkilemeyeceği gösterilir

Prensip

- *Örtüşen altproblemler* özelliği olduğunu göstermek
  - Bir altproblemin farklı altproblemlerin çözümünde ortaya çıktığı gösterilir
  - Bu altproblemleri dolayısıyla birden çok kez çözmek gerekir
  - Bağımsız altproblemler özelliği gereği bir altproblemin her çözümü aynı değeri vereceğinden bu altproblem bir kez çözülüp değeri tabloda saklanır; sonraki ihtiyaçlarda tekrar çözmek yerine saklanmış olan çözüm kullanılır. Böylelikle hesaplama çok daha hızlı yapılır.

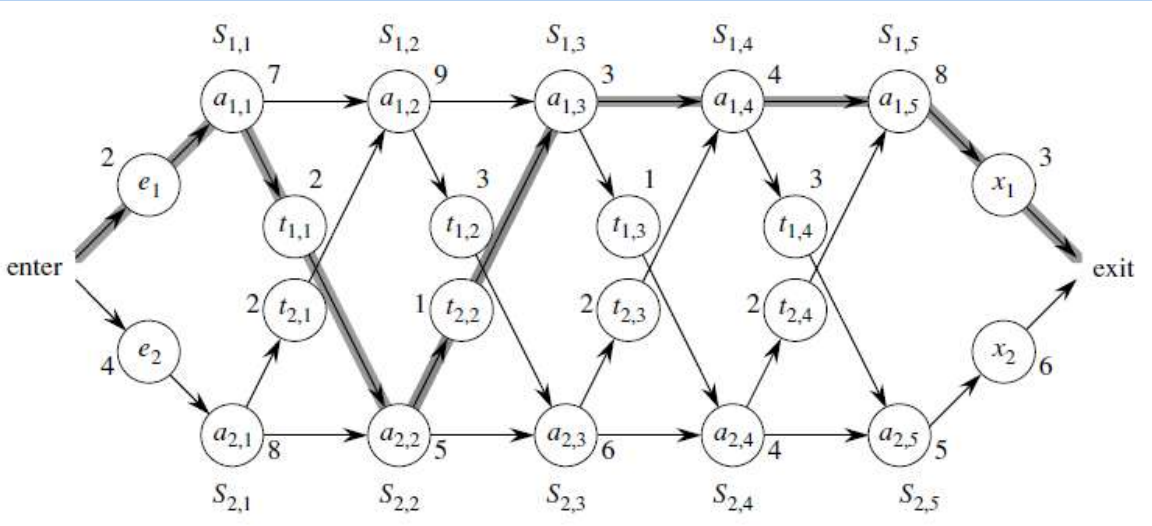
Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- Problem
  - Araba üretmek için herbirinde *n* istasyon olan 2 hat var.
  - $S_{i,j}$: *i*. hattın *j*.istasyonu. *i*=1, 2 ve *j*=1, 2, …,*n*. $S_{1,j}$ *ve* $S_{2,j}$ *de aynı iş yapılıyor*
  - $a_{i,j}$: $S_{i,j}$ de geçen süre
  - $t_{i,j}$: $S_{i,j}$ den indirip $S_{k,j+1}$ e yükleme süresi, $i \neq k$.
    - $S_{i,j}$ den indirip $S_{i,j+1}$ e yükleme süresi sıfır
  - e_i: $S_{i,1}$ e yükleme süresi
  - x_i: $S_{i,n}$ den indirme süresi



Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- İlk istasyon olarak $S_{1,1}$ yada $S_{2,1}$ kullanılabilir
- *$S_{i,j}$ den yalnız $S_{i,j+1}$ yada $S_{k,j+1}$ e transfer yapılabilir, i≠k.*
- Son istasyon olarak $S_{1,n}$ yada $S_{2,n}$ kullanılabilir
- İstenen: En kısa süreli (optimal) üretim zamanını ve kullanılması gereken istasyonları bul



Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- Kaba kuvvet çözümü
  - *n* uzunluğunda tüm farklı rota sıralılarını üret, bunların üretim süresini hesapla ve minimum sonucu vereni seç
  - Çalışma zamanı: $O(n2^n)$
    - 2^n farklı sıralı var ve her biri için O(n) zamanda toplam süre hesaplanır.
    - n>20 için pratik değil.

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP?
  - Optimal altyapı özelliği:
    - Optimal çözüm ya $S_{1,n}$ yada $S_{2,n}$ yi kullanır. Hangisi olursa olsun buraya $S_{1,n-1}$ yada $S_{2,n-1}$ den gelinmiş olunmalıdır.
    - Optimal seçim örneğin $S_{1,n-1}$ olsun, buraya da en baştan itibaren optimal gelinmesi gerekir. Aksi halde kes-ve-yapıştır gereği orijinal çözüm optimal olmazdı.

Orijinal problemin optimal çözümü alt problemlerin optimal çözümünden elde edilebilir

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP?
  - *Bağımsız altproblemler* özelliği:
    - $S_{i,j}$ den geçen optimal çözüm, $S_{i,j-1}$ den yada $S_{k,j-1}$ den geçer. Bu iki altproblemin çözümü ise birbirinden bağımsızdır. Yani, birinin çözümü diğerini etkilemez.

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP?
  - *Örtüşen altproblemler* özelliği:
    - Hem $S_{1,j}$ den hemde $S_{2,j}$ den geçen optimal çözümlerin hesaplanmasında $S_{1,j-1}$ den geçen optimal çözüme ihtiyaç duyulur.

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

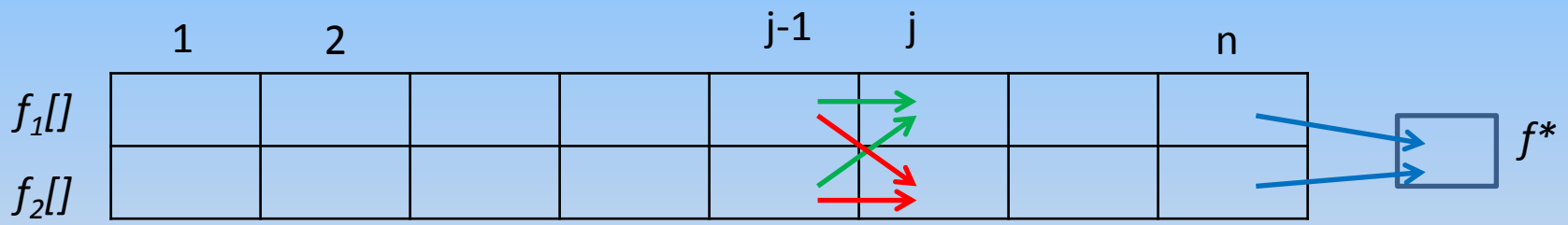
- DP çözümü
  - Optimal çözümün yapısının karakteri
    - $S_{1,j}$ den geçen en hızlı yol ya $S_{1,j-1}$ yada $S_{2,j-1}$ den gelir
    - $S_{2,j}$ den geçen en hızlı yol ya $S_{1,j-1}$ yada $S_{2,j-1}$ den gelir
    - Dolayısıyla, $S_{1,j-1}$ yada $S_{2,j-1}$ için optimal çözümler verildiğinde $S_{1,j}$ ve $S_{2,j}$ kolaylıkla bulunabilir. İki seçeneğide değerlendir hangisi iyiyse onu seç.

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP çözümü
  - Optimal çözümün değerini özyineli tanımla (!Dikkat burada sadece tanımlama yapılıyor hesaplama yapılmıyor)
    - $S_{i,j}$ den geçen en hızlı yolun süresi $f_i[j]$ ve f^* da optimal çözümün değeri olsun
    - j=1 için
      $f_1[1] = e_1+a_{1,1}$
      $f_2[1] = e_2+a_{2,1}$
    - j>1 için
      $f_1[j] = min(f_1[j-1]+a_{1,j}, f_2[j-1]+t_{2,j-1}+a_{1,j})$
      $f_2[j] = min(f_2[j-1]+a_{2,j}, f_1[j-1]+t_{1,j-1}+a_{2,j})$
    - $f^* = min(f_1[n]+x_1, f_2[n]+x_2)$

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP çözümü
  - Optimal çözümün değerini aşağıdan-yukarı hesapla
    - Önce $f_i[1]$, sonra $f_i[2]$, …$f_i[n]$ ve en son f^*

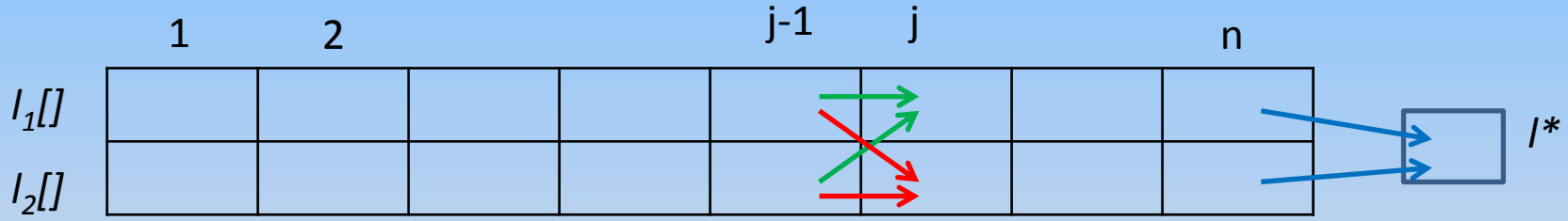


Zaman karmaşıklığı: O(n)
Hafıza karmaşıklığı: O(n)

Şekildeki her bir kutu bir altproblemin optimal çözüm değeridir.
Toplam $2n+1$ = O(n) altproblem vardır.

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- DP çözümü
  - Bir optimal çözümün kendisini bul
    - $l_i[j] = 1$, eğer $f_1[j] < f_2[j]$ ve $l_i[j] = 2$ diğer durum, olarak tanımlansın
    - Önce $l_i[1]$, sonra $l_i[2]$, … $l_i[n]$ ve en son l^* hesaplanır



Zaman karmaşıklığı: O(n)
Alan karmaşıklığı: O(n)

f ve *l* tabloları aynı sırada
aynı döngü içinde hesaplanır

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

Optimal çözüm ve değerini hesaplama

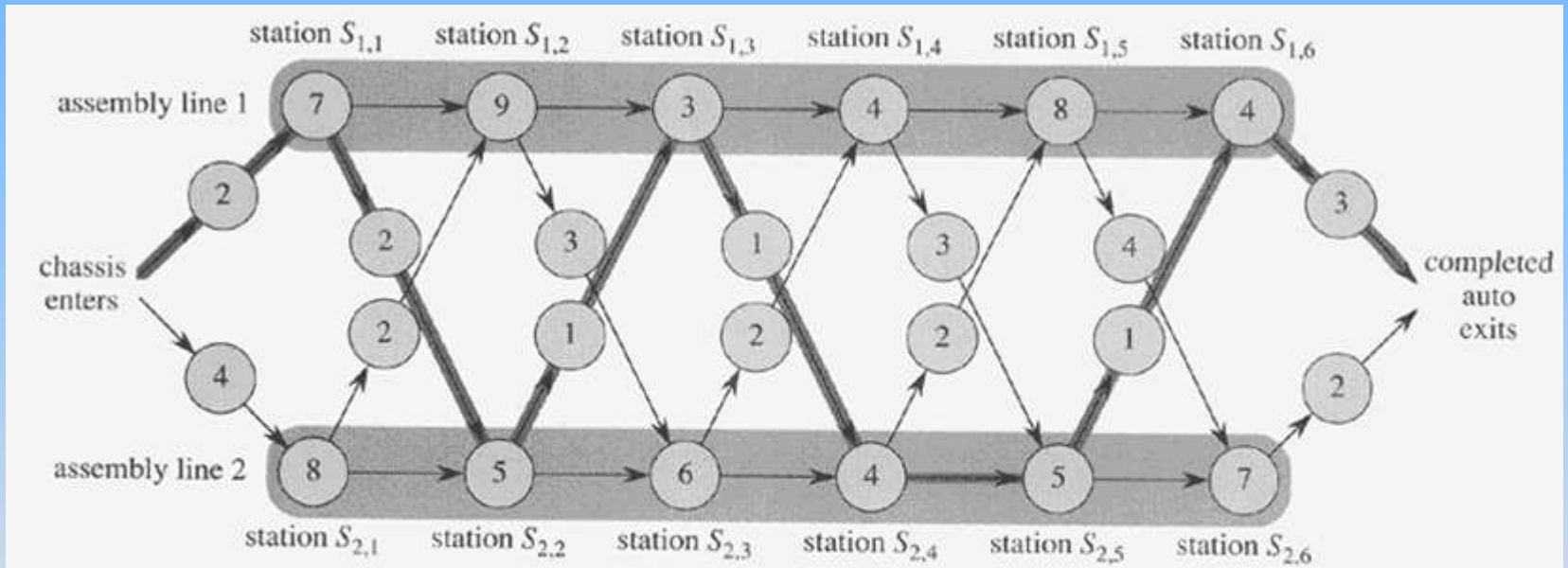
```
FASTEST-WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4       do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5             then f1[j] ← f1[j − 1] + a1,j
 6                  l1[j] ← 1
 7             else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                  l1[j] ← 2
 9          if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10             then f2[j] ← f2[j − 1] + a2,j
11                  l2[j] ← 2
12             else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                  l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```

Optimal çözümün yazdırılması

```
PRINT-STATIONS(l, n)
1  i ← l*
2  print "line " i ", station " n
3  for j ← n downto 2
4       do i ← li[j]
5          print "line " i ", station " j − 1
```

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- Örnek



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

Üretim hattı çizelgeleme (Assembly-line scheduling, ALS)

- Gözlemler
  - 2. aşamadaki özyineli formül doğrudan (yukarıdan-aşağı, *top-down*) gerçekleştirilirse
    - Zaman karmaşıklığı= $O(2^n)$, çünkü çalışma zamanı $T(n) = 2T(n-1) + 1$ dir.
    - Hafıza karmaşıklığı =özyineleme derinliği =O(n)
  - Eğer optimal çözümün sadece değeri isteniyorsa aşağıdan yukarı hesaplama ile hafıza karmaşıklığı O(1) e indirilebilir.
    - Çünkü her bir $f_i[j]$ değeri sadece $f_1[j-1]$ ve $f_2[j-1]$ i kullanıyor, daha öncekilere ihtiyaç duyulmuyor.

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Problem
  - $A_1, A_2, \ldots, A_n$ *n* adet matris içeren bir zincir, öyleki
    - Her bir A_i ve A_{i+1} çarpma uyumlu
  - Bu zinciri çarpıp tek bir matris elde etmek istiyoruz
  - Minimum sayıda çarpma işlemi ile bu işi yapacak optimal parentezlemeyi bul
- Çözüm için matrislerin içeriğini bilmeye gerek yok, sadece boyutlarını bilmek yeterli
  - $P=<P_0, P_1, \ldots, P_n>$ matris zincirinin boyutları olsun, öyleki P_i değeri = A_i nin sütun sayısı= A_{i+1} in satır sayısı
- $A_i x A_{i+1}$ için çarpma sayısı= $P_{i-1} * P_i * P_{i+1}$

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Örnek zincir (*n=3*)
  - A_1, A_2, A_3 ve P=<P_0, P_1, P_2 , P_3> = <10, 100, 5, 50>, yani A_1=10x100, A_2=100x5, ve A_2=5x50
  - Çarpım sonucu matris A= 10x50
  - Parentezleme 1: (A_1x(A_2xA_3)) için 75.000 çarpma
  - Parentezleme 2: ((A_1xA_2)xA_3) için 7.500 çarpma
  - Tüm parentezlemeler aynı sonucu verir (matris çarpmanın birleşme özelliği) fakat farklı sayıda çarpma işlemi gerektirir

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Kaba kuvvet çözümü
  - *n* tane matrisin parentezleme sayısını *Q(n)* ile gösterelim,
  - $Q(n) = 1 \qquad n=1,$

    $= \sum_{k=1..n-1} Q(k)Q(n-k) \qquad n>1$

Fakat $Q(n) > 2^n$

n> 15 için pratik değil

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP?
  - *Optimal altyapı özelliği:*
    - Problemin optimal çözümü zinciri bir k değerinden bölmek zorundadır. Bölme sonrası $(A_1, A_2, \ldots, A_k)$, $(A_{k+1} \ldots A_n)$ iki altproblem elde edilir.
    - Her iki altproblemde optimal olarak çözülmüş olması gerekir, aksi halde orijinal problem optimal çözülmüş olmazdı (kes-ve-yapıştır).
    - Fakat *k* değerini bilmiyoruz, ama bildiğimiz çok önemli bir şey var *k*=1, 2, …, *n*-1 seçimlerinden birini alabilir.
    - Dolayısıyla her bir seçimi deneyip minimum sonucu veren seçimi *k* nın değeri olarak belirleyebiliriz

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP?
  - *Bağımsız altproblemler* özelliği:
    - Optimal seçim geride $(A_1, A_2, \ldots, A_k)$, $(A_{k+1} \ldots A_n)$ iki altproblem bırakır.
    - Bunlardan birinin optimal çözümü diğerinin optimal çözümünü etkilemez, çünkü bu iki alt problem tamamen ayrıktır.

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP?
  - *Örtüşen altproblemler* özelliği:
    - Her bir altproblem $A_i, A_{i+1}, \ldots, A_j$ nin optimal parentezlemesi dir.
    - Örneğin, hem $A_1, A_2, \ldots, A_j$ hem de $A_i, A_{i+1}, \ldots, A_n$ nin optimal parentezlemesi $A_i, A_{i+1}, \ldots, A_j$ nin optimal parentezlemesinin hesabına ihtiyaç duyar.

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP çözümü
  - Optimal çözümün yapısının karakteri
    - $A_1, A_2, \ldots, A_k, A_{k+1} \ldots A_n$ nin optimal parentezlemesi zinciri bir k değerinden böler. Bölme sonrası kalan iki alt problemde ($A_1, A_2, \ldots, A_k$ve $A_k, A_{k+1} \ldots A_n$) kendi içinde optimal olarak parentezlenmiş olmalıdır.

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

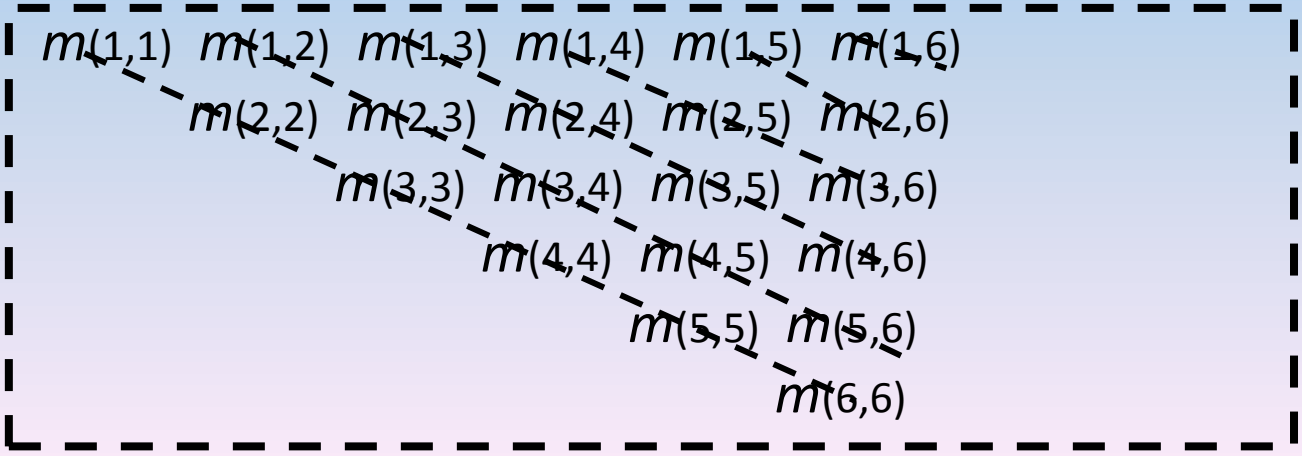
- DP çözümü
  - Optimal çözümün değerini özyineli tanımla
    - m[i, j] = A_i, A_{i+1}, …, A_j nin optimal parentezlenmesindeki çarpma sayısı olsun

$$m[i,j] = 0 \qquad i=j$$

$$= \min_{i \le k < j} \{m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j\} \qquad j>i$$

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP çözümü
  - Optimal çözümün değerini aşağıdan-yukarı hesapla
    - Her bir *i>j* için *m(i, j)* bir altproblemin optimal değeri aşağıdaki matriste diyagonal gezinilerek hesaplanır, yani önce 1 uzunluktakiler, sonra 2 ve en son *n*



Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- DP çözümü
  - Bir optimal çözümün kendisini bul
    - *m[i, j]* yanında optimal sonucu veren $i \leq k < j$ değerini *s[i, j]* de tut.
    - Optimal çözümü *s* matrisinden üret.

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

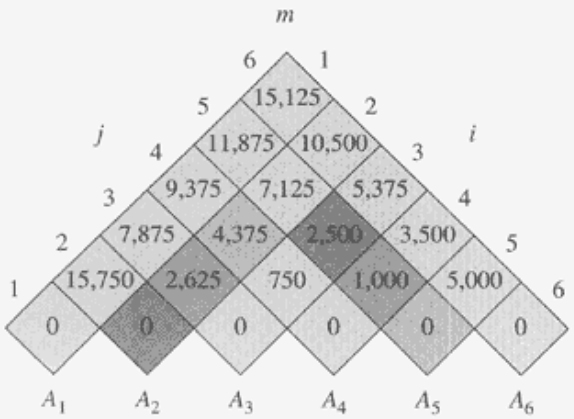
```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n                ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m and s
```

Zaman Karmaşıklığı: $O(n^3)$

Hafıza karmaşıklığı: $O(n^2)$

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- örnek



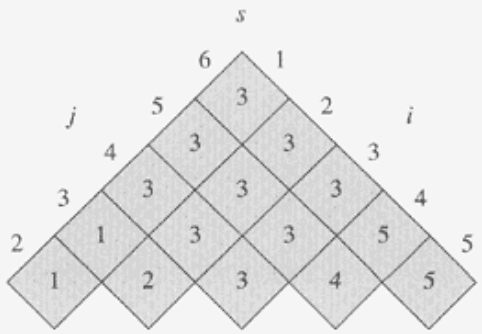


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= 7125\,.$$

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Optimal çözümün yazdırılması

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2      then 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 ")"
```

Zaman karmaşıklığı: O(n)

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Gözlem
  - Eğer recursive çözüm doğrudan kodlansaydı

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

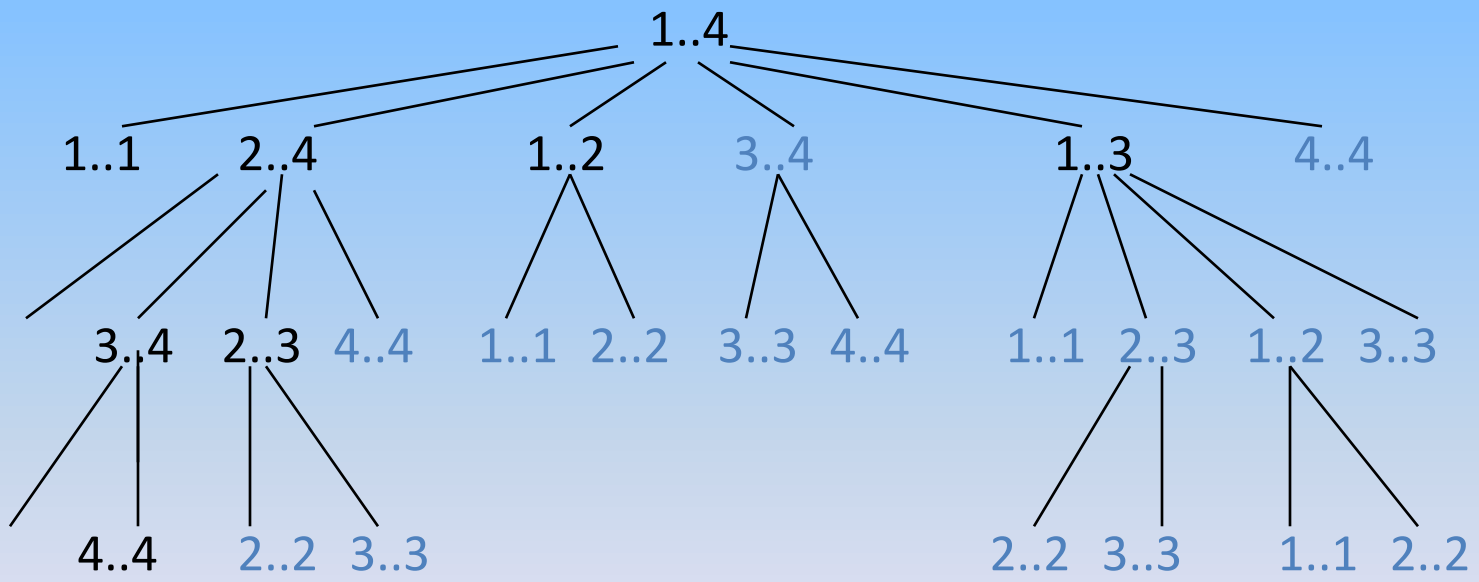
1. **if** $i=j$ **then return** 0
2. $m[i,j] \leftarrow \infty$
3. **for** $k \leftarrow i$ to j-1
4. **do** $q \leftarrow$ RECURSIVE-MATRIX-CHAIN(p,i,k)+
RECURSIVE-MATRIX-CHAIN(p,k+1,j)+$p_{i-1}p_kp_j$
5. **if** $q < m[i,j]$ **then** $m[i,j] \leftarrow q$
6. **return** $m[i,j]$

En baştaki çağırma:
RECURSIVE-MATRIX-CHAIN(p, *1*, n)

Zaman karmaşıklığı: $O(2^n)$

Matris Zinciri Çarpımı (Matrix-Chain Multiplication, MCP)

- Gözlem
  - Örtüşen altproblemler nedeniyle hesaplama ağacı



RECURSIVE-MATRIX-CHAIN(*p*, *1*, *4*) için

Memoization

- DP problemini yukarıdan-aşağı çözmedir
  - Özyineli tanımı doğrudan kodla ve çözülen her altproblemin çözümünü bir tabloda sakla
  - Bir altproblemi çözmeden önce daha önce çözülüp sonucu saklanmış mı diye bak,
    - Eğer daha önce çözülmüşse onu kullan,
    - Eğer daha önce çözülmemişse özyineli olarak çöz
  - Avantajı (Aşağıdan-yukarı DP ye göre)
    - Tablonun hangi sırada (satır satır, sürün sütün ya da diyagonal) doldurulması gerektiğini bilmeye gerek yok
    - Eğer orijinal problemin çözümü için tüm altproblemlerin değil çok azının çözülmesi gerekiyorsa
  - Dezavantajı (Aşağıdan-yukarı DP ye göre)
    - Özyinelemeden dolayı çağırma yığıtına ihtiyaç duyması, yani özyineleme derinliği büyükse fazla hafıza kullanılması

Memoization
(Matris Zinciri Çarpımı, MCP)

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3       do for j ← i to n
4            do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)
```

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

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

Altproblem daha önce çözülmüşse bu değeri kullan

Altproblem daha önce çözülmemişse çöz ve değerini sakla

Zaman karmaşıklığı: $O(n^3)$

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- Problem
  - İki tane sıralı $X=<x_1, x_2,..., x_m>$ ve $Y=<y_1, y_2,..., y_n>$ *veriliyor. Öyle bir* $Z=<z_1, z_2,..., z_k>$ *bulki hem X hem de Y nin bir alt sıralısı olsun ve uzunluğu en büyük olsun.*
  - İki örnek LCS problemi gösterilmiştir
    - Altsıralı bitişik olmak zorunda değildir

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- Kaba Kuvvet Çözümü
  - X in tüm altsıralılarını üret ve her birinin Y nin bir alt sıralısı olup olmadığını test et ve olanlar içinde en uzunu kaydet
  - Zaman karmaşıklığı: $O(n2^m)$
    - X in altsıralı sayısı $O(2^m)$
    - Bir altsıralının Y nin altsıralısı olup olmadığı testi O(n)
  - Basit bir iyileştirme ile
    - Zaman karmaşıklığı: *min*($O(n2^m)$, $O(m2^n)$)

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- DP Çözümü
  - Problemin bir DP problemidir: Bahsedilen 3 özelliğe sahip
- Optimal çözümün yapısının karakteri
  - X_i notasyonu X in i-uzunluğundaki önek (*prefix*) i olsun.
  - $X=X_m=\langle x_1, x_2,\ldots, x_m\rangle$, $Y=Y_n=\langle y_1, y_2,\ldots,y_n\rangle$ ve $LCS(X, Y)=Z=Z_k=\langle z_1, z_2,\ldots, z_k\rangle$ ise, şunlar sağlanır:
    1. Eğer $x_m= y_n$ ise, $z_k= x_m= y_n$ ve $Z_{k-1}=\text{LCS}(X_{m-1},Y_{n-1})$ bir fazlası
    2. Eğer $x_m\neq y_n$ ve $z_k\neq x_m$ ise $Z=\text{LCS}(X_{m-1},Y_n)$ ikisinin en iyisi
    3. Eğer $x_m\neq y_n$ ve $z_k\neq y_n$ ise $Z=\text{LCS}(X_m ,Y_{n-1})$

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- DP çözümü
  - Optimal çözümün değerini özyineli tanımla
    - c[i, j] = X_i ve Y_j nin optimal çözümünün uzunluğu olsun, yani c[i, j]= |LCS(X_i, Y_j)|. Bu durumda,

$$
\begin{aligned}
c[i, j] &= 0 && i=0 \text{ veya } j=0 \\
&= c[i-1, j-1]+1 && i,j>0 \text{ ve } x_i = y_j \\
&= \max\{c[i-1, j], c[i, j-1]\} && i,j>0 \text{ ve } x_i \neq y_j
\end{aligned}
$$

Aradığımız değer = $c[m, n]$= |LCS(X_m, Y_n)|= |LCS(X, Y)|

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- DP çözümü
  - Optimal çözümün değerini aşağıdan-yukarı hesapla
    - Her bir *c[i, j]* bir alt problemin optimal çözüm değeri
    - Tabloyu satır satır ya da sütun sütün gezerek hesaplayabiliriz. Çünkü her bir altproblem *c[i, j]* yi çözmek için *c[i-1, j-1], c[i-1, j]* ve *c[i, j-1]* i bilmek yeterli.
  - Bir optimal çözümün kendisini bul
    - *c[i, j]* yi çözmek için üç seçimden hangisinin kullanıldığını *b*[*i*,*j*] değerinde tutalım.
      - *b*[*i*,*j*] =“↑ ” eğer *c[i-1, j]* kullanıldıysa
      - *b*[*i*,*j*] =“←” eğer *c[i, j-1]* kullanıldıysa
      - *b*[*i*,*j*] =“↖” eğer *c[i-1, j-1]* kullanıldıysa

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                        then c[i, j] ← c[i − 1, j]
14                             b[i, j] ← "↑"
15                        else c[i, j] ← c[i, j − 1]
16                             b[i, j] ← "←"
17  return c and b
```

Optimal çözüm ve değerinin bulunması

Zaman Karmaşıklığı: O(nm)
Hafıza karmaşıklığı: O(nm)

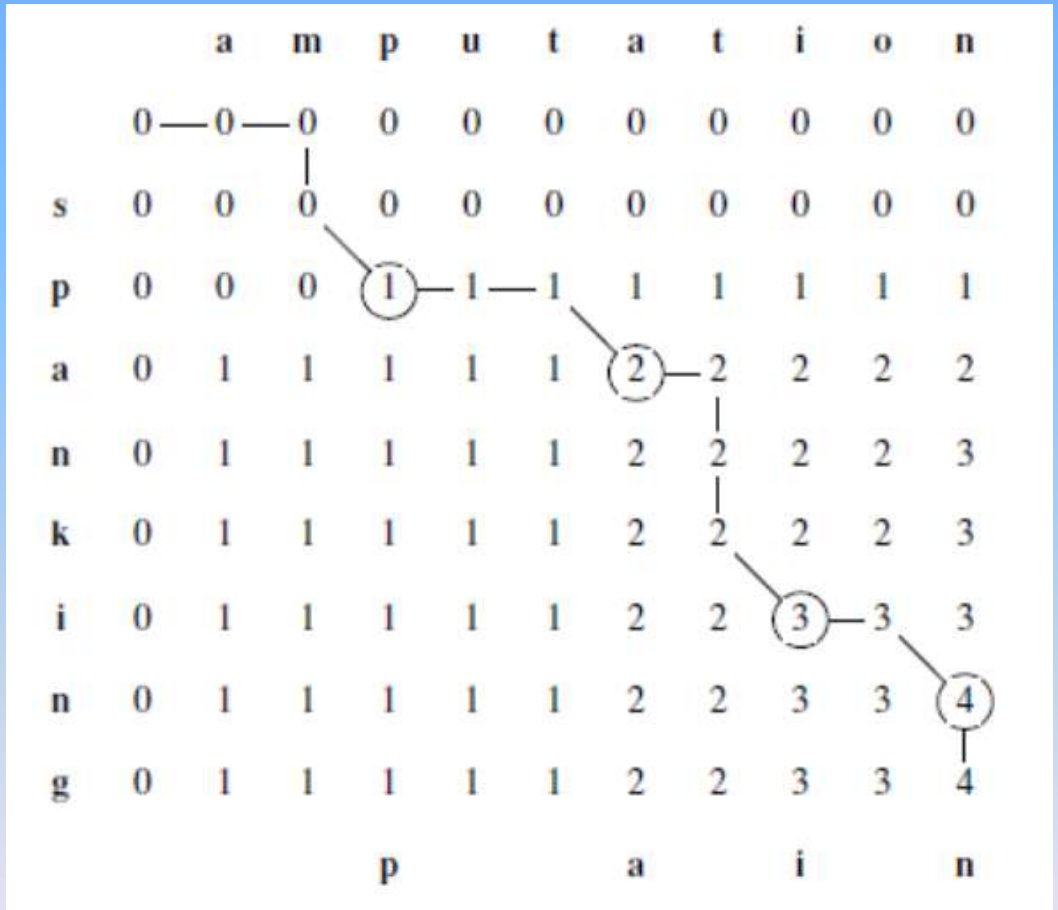
```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Optimal çözümün yazdırılması

Zaman Karmaşıklığı: O(max(n,m))
Hafıza karmaşıklığı: O(nm)

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- Örnek



		a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4
				p			a		i		n

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- Gözlem
  - *b* tablosu kullanmadan LCS bulmak

```
Print_LCS_without_b(c,X,i,j) {
    If (i=0 or j=0) return;
    If (c[i,j]==c[i-1,j-1]+1)
        {Print_LCS_without_b(c,X,i-1,j-1); print x_i}
    else if(c[i,j]==c[i-1,j])
        {Print_LCS_without_b(c,X,i-1,j);}
    else
        {Print_LCS_without_b(c,X,i,j-1);}
}
```

En Uzun Ortak Altsıralı (Longest Common Subsequence, LCS)

- Gözlem
  - Eğer sadece optimal çözümün değeri isteniyorsa
    - Hafıza gereksinimi $2min\{m, n\}$ e indirilebilir, çünkü daha önceki satır/sütunları tutmaya gerek yok. Hatta biraz daha şık bir çözümle hafıza karmaşıklığı $min\{m, n\}+1$ olur. Her iki durumda da karmaşıklık $O(min(m, n))$ dir.

DP Zaman Karmaşıklığı

- Bir DP probleminin zaman karmaşıklığı kabaca
  - Alt problem sayısı: a
  - Bir altproblemi çözmek için kullanılan diğer altproblem sayısı, diğer bir deyişle seçim sayısı: k

  olmak üzere O(ak) dır.

Problem	Altproblem sayısı (a)	Seçim sayısı (k)	Zaman karmaşıklığı
ALS	n	2	$2n=O(n)$
MCP	n^2	n	$O(n^3)$
LCS	nm	3	$3nm=O(nm)$

Optimizasyon Problemleri Harici DP kullanımı

- DP optimizasyon harici diğer özyineli ya da iteratif ilişkilerin çözümünde de kullanılır
  - Alt problemleri çöz tabloda sakla ihtiyaç olduğunda tekrar hesaplama yerine tablodan kullan

Optimizasyon Problemleri Harici DP kullanımı

- Fibonacci sayıları
  - Tanım: F(n) ile n. Fibonacci sayısını gösterelim

$$F(n) = n \qquad n=0 \text{ veya } n=1$$
$$= F(n-1) + F(n-2) \qquad n \geq 2$$

Özyineli çözüm

```
RFibo (n)
if n = 0 or n=1 then return n
else return RFibo (n-1) + RFibo (n-2)
```

Zaman Karmaşıklığı < $O(2^n)$,
fakat eksponansiyel

DP çözüm

```
DPFibo (n)
    f[0] ← 0
    f[1] ← 1
    for  i ← 2 … n do
        f[i] ← f[i-1] + f[i-2]
    return f[n]
```

Zaman Karmaşıklığı: O(n)

Her iki çözümünde hafıza karmaşıklığı: O(n)

Optimizasyon Problemleri Harici DP kullanımı

- Binom katsayıları
  - Tanım: *n* nin *k* lı kombinasyon sayısı *C(n, k)* ile gösterilir ve binom katsayısı olarak bilinir. Özyineli tanımı,

$$C(n,k) = 1 \quad k=n \text{ veya } k=0$$
$$= C(n-1,k-1) + C(n-1,k) \quad n > k > 0$$

```
DPBinomial(n,k)
for i ← 0 to n do
  for j ← 0 to min(i, k) do
     if j = 0 or j = i then
           C[i, j] ← 1
      else
           C[i,j] ← C[i-1, j-1] + C[i-1,j]
return C[n,k]
```

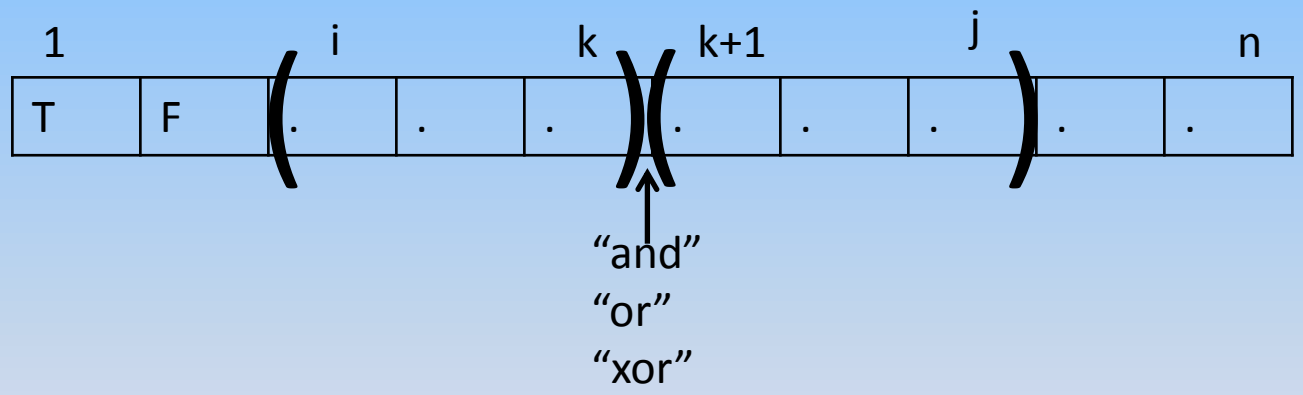
Zaman karmaşıklığı: $O(n^2)$
Hafıza karmaşıklığı: $O(n^2)$

Optimizasyon Problemleri Harici DP kullanımı

- Uzunluğu *n* olan bir True/False (T/F) sembol sıralısı veriliyor. Her iki sembol arasına “or”, “and” ya da “xor” ikili operatörü yerleştirerek kaç farklı parentezleme ile ifadenin sonucu True olur.
- Örn, sıralı `TTFFFT` olsun, sonucu True olan birkaç parentezleme
  - `((T and T) or F) xor ((F or F) and T)`
  - `((T and T) and F) or (F or (F xor T))`
  - `((T and T) or (F and F)) or (F or T)`

Optimizasyon Problemleri Harici DP kullanımı

- DP çözümü
  - *i*. ve *j*. semboller (*i*<*j*) arasını True yapan parentezleme sayısını T(*i*, *j*) ve False yapan parentezleme sayısını F(*i*, *j*) ile gösterelim
    - *i* ve *j* arası *i*≤*k*<*j* dan bölünebilir ve araya üç operatörden birisi gelebilir



$$T(i,j) = \sum_{i \le k < j} T(i,k)T(k+1,j) \quad \text{"and"}$$

$$+ \sum_{i \le k < j} T(i,k)T(k+1,j) + T(i,k)F(k+1,j) + F(i,k)T(k+1,j) \quad \text{"or"}$$

$$+ \sum_{i \le k < j} F(i,k)T(k+1,j) + T(i,k)F(k+1,j) \quad \text{"xor"}$$

Optimizasyon Problemleri Harici DP kullanımı

- DP çözümü
  - F(i, j) de T(i, j) ye benzer şekilde tanımlanır
  - Temel durumlar
    - T(i,i) =1 eğer i. sembol T ise, değilse T(i,i) =0
    - F(i,i) =1 eğer i. sembol F ise, değilse F(i,i) =0
  - T(1,n) değeri problemin çözümüdür

Altproblem sayısı = n^2
Seçim sayısı = O(n)

Zaman Karmaşıklığı: $O(n^3)$
Hafıza Karmaşıklığı: $O(n^2)$