BÖLÜM 24. DİNAMİK PROGRAMLAMA

Dinamik Programlama (DP), böl ve yönet yöntemine benzer olarak alt problemlerin çözümlerini birleştirerek çözüm elde eden bir yöntemdir. 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. DP gelişimi dört adımda özetlenebilir ve bu dört adım DP' nin temelini oluştururlar.

a) Optimal çözümün yapısının karakteristiği ortaya çıkarılmalı.
b) Özyinelemeli olarak optimal çözümün değerini tanımlamalı.
c) Alttan-üste (bottom-up) mantığı ile optimal çözümün değerini hesaplamalı.
d) Hesaplanan bilgilerden optimal çözümün elde edilir. Eğer problemin bir tane çözümü varsa, bu durumda bu adım atlanabilir.

Dinamik programlama örnekler üzerinde daha detaylı olarak açıklanabilir.

24.1. İki Basit Problem

Bu bölümde dinamik programlamanın mantığının anlaşılması için iki tane basit problem ele alınacaktır.

Örnek 24.1.

Fibonacci sayıları $F_n=F_{n-1}+F_{n-2}$ şeklinde tanımlanan bir seridir ve $F_0=0$, $F_1=F_2=1$ şeklinde başlangıç değerleri vardır. 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. Özyineleme ile yapılacak olan hesaplama Algoritma 24.1 ile yapılabilir. Bu algoritmanın mertebesi

Algoritma 24.1. Fibonacci(n)

▷ n hesaplanacak olan terimin numarasını gösterir.

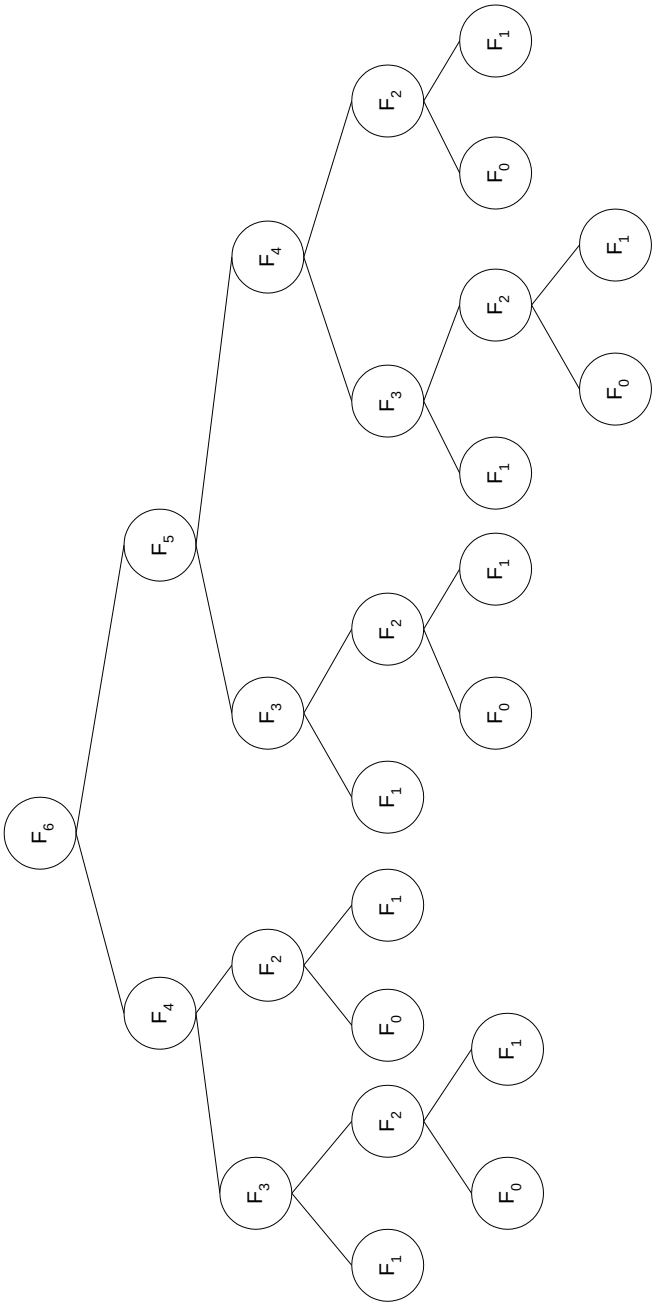
```
1- eğer n≤1 ise
2-       sonuç←1
3- değilse
4-      sonuç←Fibonacci1(n-1)+Fibonacci1(n-2)
```

$T(n)\geq T(n-1)+T(n-2)+\Theta(1)$

olur. T(n) bağıntısı Fibonacci sayıları gibi artar ve artış üsteldir. Fibonacci sayılarından F_n sayısını hesaplamak için F_{n-1} ve F_{N-2} sayılarına ihtiyaç vardır.

Fibonacci sayılarını hesaplamak için Algoritma 24.1 çok yavaş çalışır ve yavaş çalışmasının sebebi her seferinde F_n sayısından küçük olan bütün Fibonacci sayılarını hesaplamasıdır ve bu durum Şekil 24.1' de görülmektedir.

Dikkat edilirse, aynı terim birden fazla terimin hesaplanmasında kullanıldığından her seferinde tekrar-tekrar hesaplanmasından dolayı bu işlem yavaş yapılmaktadır. Algoritma 24.1' in yavaş çalıştığı aynı algoritmanın zaman bağıntısının mertebesinin elde edilmesiyle de elde edilir.



Şekil 24.1. *F_6 Fibonacci sayısının hesaplanması ağacı.*

$T(n) \geq T(n-1)+T(n-2)+\Theta(1)$

tekrarlı bağıntısının homojen çözümü için karakteristik denklem

$r^2-r-1=0$

olur. Bu denklemin kökleri

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

olur. Bu durumda denklemin homojen çözümü

$$T(n)_h = c_1\left(\frac{1+\sqrt{5}}{2}\right)^n + c_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

olur. c_1 ve c_2 birer sabittir. Aynı denklemin özel çözümünün üstel artması mümkün değildir, bundan dolayı algoritmanın mertebesini homojen çözüm belirler ve görüldüğü gibi üstel bir bağıntıdır.

Algoritma 24.2. Fibonacci(n)

▷ n hesaplanacak olan terimin numarasını gösterir.

```
1.  Eğer n≤1 ise
2.     Sonuç←1
3.  değilse
4.     Son←1
5.     Son2←1
6.     Cevap←1
7.     i←2, ..., n kadar
8.        Cevap←Son+Son2
9.        Son2←Son
10.       Son←Cevap
11.  Sonuç←Cevap
```

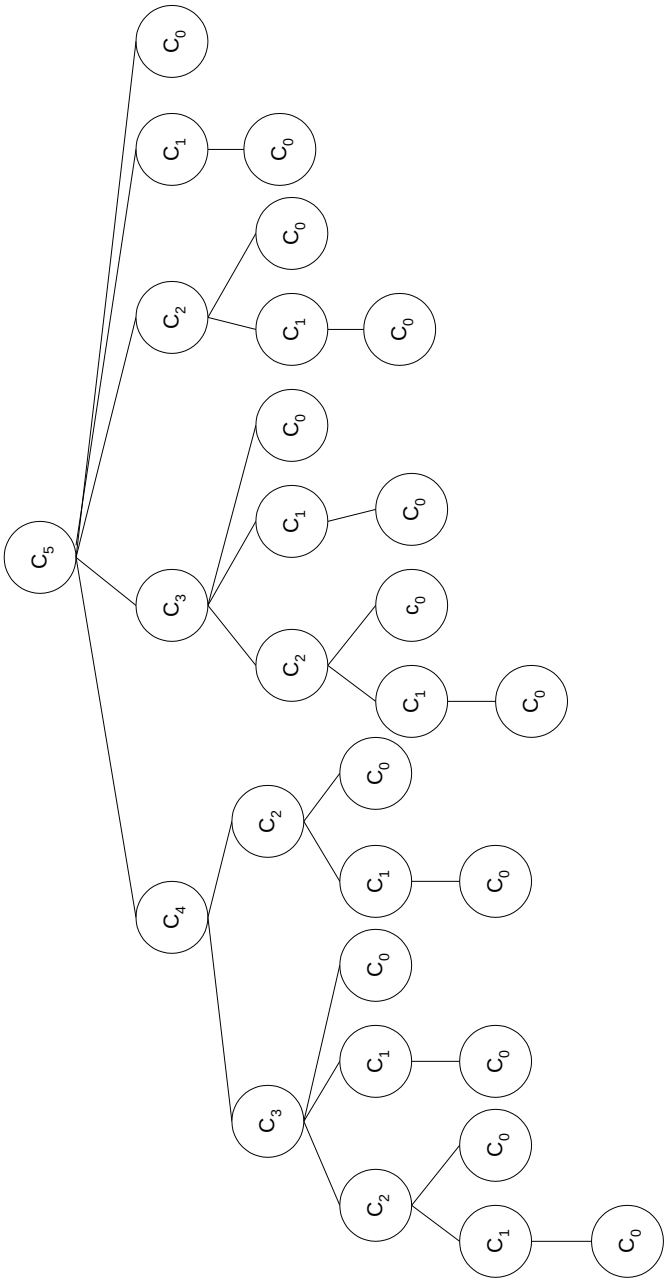
Bu yavaşlığı ortadan kaldırmak için hesaplanan her terim, lazım olduğu her yerde tekrar hesaplanmak yerine kullanıldığında bu yavaşlık ortadan kalkacaktır. Bunun için dinamik programlama yöntemi kullanılır. Bu amaçla, hesaplanan Fibonacci sayıları saklanıp bir sonraki adımda kullanılacak olursa, F_n sayısını hesaplamak için daha az zaman harcanacaktır. Bu işlemi gerçekleştiren algoritma Algoritma 24.2' de görüldüğü gibidir.

Algoritma 24.2' nin zaman bağıntısı doğrusal olur. Çünkü bir tane n değişkenine bağlı döngü vardır.

Örnek 24.2.
Diğer basit bir problemde

$$C(n) = \frac{2}{n}\sum_{i=0}^{n-1} C(i) + n$$

sayısının hesaplanmasıdır. Bu sayının özyinelemeli hesaplanması Algoritma 24.3' te görüldüğü gibi olur ve Şekil 24.2' de 5. terimin hesaplanmasının adımları görülmektedir.



Şekil 24.2. *C_5 sayısının hesaplanması ağacı.*

Algoritma 24.3. CnHesapla(n)

▷ n hesaplanacak olan terimin numarasını gösterir.

```
1.  Eğer n=0 ise
2.     Sonuç←1
3.  değilse
4.     Toplam←0.0
5.     i←0, ..., n-1 kadar
6.        Toplam←Toplam+CnHesapla(i)
7.  Sonuç←2*Toplam/n+n
```

Şekil 24.2' den de görüleceği gibi C_5 sayısının hesaplanması için 16 tane C_0, 8 tane C_1, 4 tane C_2, 2 tane C_3 ve 1 tane C_4' e ihtiyaç vardır. Bundan dolayı, her terim ilk hesaplandığında bir tabloya atılır ve bu şekilde yapılan bie hesaplamada bir kez hesaplanan değer bir daha hesaplanmak zorunluluğunu ortadan kaldıracaktır. Algoritma 24.3' ün zaman bağıntısı

$$T(n) = \sum_{i=0}^{n-1} T(i) + n$$

olur. Bu bağıntı çözüldüğünde, çözümün üstel olduğu görülecektir. Çünkü bu denklemin homojen çözümünün karakteristik denklemi

$$r^n - r^{n-1} - r^{n-2} - \ldots - r - 1 = 0$$

olur. Görüldüğü gibi n. dereceden bir bilinmeyenli bir denklemdir ve bu denklemin n tane kökü olacaktır. Çözüm ise bu köklerin üstel bağıntısı olacaktır. Bu problemi gidermek için hesaplanan değerlerin tabloya atılmasıyla elde edilen çözüm Algoritma 24.4' te görülmektedir.

Algoritma 24.4. CnHesapla(n)

▷ n hesaplanacak olan terimin numarasını gösterir.

```
1. C(0) ←1.0
2. i←1,…, n' e kadar
3.    Toplam←0.0
4.    j=0, …, i-1' e kadar
5.       Toplam←Toplam+C(j)
6.   C(i)←2.0*Toplam/i+I
7. Sonuç←C(n)
```

Algoritma 24.4' teki algoritmanın analizi yapıldığında n' e bağlı ve iç-içe iki tane döngü olduğundan zaman bağıntısı $T(n)=\Theta(n^2)$ olacaktır.

24.2. Zincir Matris Çarpımı

$A_{50\times50}, B_{10\times40}, C_{40\times30}, D_{30\times5}$ matrislerinin çarpılması için minimum sayıda çarpma işleminin yapılabilmesi için nasıl bir parantezleme işlemi yapılmalıdır? Bu bölümde bu sorunun cevabı bulunacaktır.

$p\times q$ ve $q\times r$ boyutlarındaki iki matrisin çarpımında pqr tane skaler çarpım işlemi gerçekleştirilir. Buna göre verilen dört tane matris için parantezlemeler ve yapılacak olan skaler çarpım sayısı aşağıdaki gibi olur.

(A((BC)D)) için 12.000 çarpım
(A(B(CD))) için 6.000 çarpım →en iyi
((AB)(CD)) için 6.000 çarpım →en iyi
(((AB)C)D) için 20.000 çarpım
((A(BC))D) için 12.000 çarpım

$<A_1, A_2, ..., A_n>$ matris zinciri olsun. $C=A_1A_2.....A_n$ çarpımı hesaplanmak isteniyor. Matris çarpımının birleşme özelliği vardır.

Bundan dolayı her türlü paranteze alma her zaman için aynı sonucu verecektir.

Örneğin $<A_1,A_2,A_3,A_4>$ için beş tane farklı hesaplama yöntemi vardır.

$(A_1(A_2(A_3A_4)))$
$(A_1(A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3)A_4)$
$(((A_1A_2)A_3)A_4)$

} Seçilecek olan parantezlemenin hesaplama maliyeti üzerinde önemli bir etkisi vardır.

İlk olarak iki matrisin çarpımının maliyeti düşünülsün ve standart algoritması Algoritma 24.5' teki gibi olur. Bu algoritma A ve B gibi iki matrisin çarpımını yapar ve sonuç olarak C matrisini verir.

Algoritma 24.5. MatrisÇarpımı(A,B)

▷ A ve B matrisleri çarpıldıktan sonra C matrisi elde edilir. Çarpımın yapılabilmesi için A ve B matrislerinin boyutlarının uygun olması gerekir.

```
1-  Eğer sütun[A]≠satır[B] ise
2-      hata("uygunsuz boyuttur")
3-  değilse
4-     i←1, ...., satır[A] kadar
5-        j←1, ...., sütun [B] kadar
6-           C[i,j]←0
7-        k←1, ....., sütun [A] kadar
8-              C[i,j]←C[i,j]+A[i,k]B[k,j]
```

Eğer A_{pxq} ve B_{qxr} ise, C_{pxr} olur. C matrisini hesaplarken zaman yönünde baskın olan 7. satırdaki skaler hesaplamadır. Bunun zaman değeri matris boyutları türünden pqr olur.

Seçilen yolun maliyet üzerindeki etkisini daha rahat görebilmek için $<A_1,A_2,A_3>$ zinciri ele alınsın. Boyutlar sırasıyla 10x100, 100x5, 5x50 olsun.

Bu üç matrisin çarpımı $((A_1A_2)A_3)$ parantezlemesine göre yapılırsa, A_1A_2 matrislerinin çarpımında 10.100.5=500 tane skaler çarpım yapılır ve elde edilen sonuç matrisin boyutları 10x5 olur ve bu matris ile A_3 matrisinin çarpımı içinde 10.5.50=2500 tane skaler çarpım yapılır. Toplam olarak 7500 tane skaler çarpım yapılır.

Eğer matrisler $(A_1(A_2A_3))$ şeklinde çarpılırlarsa, $X=A_2A_3$ matrislerinin çarpımı için 10.100.50=25000 tane skaler çarpıma ihtiyaç vardır. Elde edilen sonuç matrisinin boyutları 10x50 olur ve bu matris ile A_1 matrisinin çarpımı için de 10.100.50=50000 tane skaler çarpıma ihtiyaç duyulur. Toplam olarak 75000 tane skaler çarpıma ihtiyaç duyar.

Çarpılacak olan n tane matris için yapılacak olan çarpım sayısı T(n) olsun. Bu durumda T(1)=T(2)=1 olur ve T(3)=2, T(4)=5 olur. Fakat n büyüdüğü zaman yapılacak olan skaler çarpım sayısı da oldukça hızlı artacaktır ve

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

kadar skaler çarpım yapılacaktır. Bunun böyle olduğunu anlamak için $A_1, A_2, \ldots, A_n$ matrisleri çarpılacak olsun. İlk önce matrisler iki gruba ayrılırlar ve

$$\underbrace{(A_1A_2...A_i)}_{\substack{i \text{ tane farklı} \\ \text{yol var}}}\underbrace{(A_{i+1}A_{i+2}...A_n)}_{\substack{n-i \text{ tane farklı} \\ \text{yol var}}}$$

şeklinde parantezleme yapılır. Bu durumda $(A_1A_2...A_i)(A_{i+1}A_{i+2}...A_n)$ çarpımı T(i)T(n-i) farklı yol ile yapılabilir. Bu tekrarlı bağıntının çözümü Catalan sayılarını verir ve bu sayılar üstel olarak artarlar. Bundan dolayı çok büyük n değerleri için en iyi parantezlemeyi aramak anlamsız olur. C(n) bir Catalan sayısı ise, bu sayı

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

olur. Yapılan parantezleme sayısı T(n)=C(n-1) olur.

$1 \le i \le n$ için C_i sayısı A_i matrisinin sütun sayısı olsun ve A_i matrisinin satır sayısı C_{i-1} olsun. C_0 sayısı A_1 matrisinin satır sayısı olur.
$A_{sol}\ A_{sol+1}\ \ldots\ldots A_{sag}$ çarpımında yapılan skaler çarpım sayısı $m_{sol,sag}$ olsun.Tutarlılık için $m_{sol,sol}=0$ olsun. $sol \le i \le sag$ için $(A_{sol}\ldots A_i)(A_{i+1}\ldots A_{sag})$ en son yapılan çarpım olsun. Toplam çarpım sayısı

$m_{sol,i}+m_{i+1,sag}+C_{sol-1}C_iC_{sag}$

olur ve bunun anlamı

$$\underbrace{\underbrace{(A_{sol}\ldots A_i)}_{m_{sol,i}}\underbrace{(A_{i+1}\ldots A_{sag})}_{m_{i+1,sag}}}_{C_{sol-1}C_iC_{sag}}$$

şeklinde olmasıdır.

Optimal parantezlemede yapılan çarpım sayısı $M_{sol,sag}$ ise, sol<sağ olmak üzere

$$M_{sol,sağ} = \min_{sol \le i < sağ}\left\{M_{sol,i} + M_{i+1,sağ} + C_{sol-1}C_iC_{sağ}\right\}$$

şeklinde olur.

24.3. Özyineleme çözüm

Dinamik programlamanın ikinci adımı da optimal çözümün değerini özyinelemeli olarak belirlemesidir. (Ama çözüm alt problemlerin çözümlerinin özyinelemeli bir bağıntısıdır)

$A_iA_{i+1}.....A_j$ parantezlemenin minimum maliyetli olmalıdır ($1 \le i \le j \le n$).

$M_{i,j}$ değeri $A_i.....A_j$ çarpımında minimum skaler çarpım olsun. $A_1....A_n$ hesaplamanın en ucuz yolu $M_{1,n}$ değeri olur. $M_{i,j}$ şu şekilde tanımlanır. Eğer i=j ise zincir sadece bir matristen oluşur ve bu durumda $A_i.....A_j=A_i$ olur. Bundan dolayı skaler çarpıma gerek yoktur. Böylece $M_{i,i}=0$ olur. Eğer i<j için $M_{i,j}$ hesaplanacaksa, ilk adımın avantajından (optimal çözümün yapısının avantajı) faydalanılınır.

$A_iA_{i+1}.....A_j \rightarrow C_k$ ve C_{k+1} şeklinde bölünsün ve $i \le k < j$. Buradan $M_{i,j}=A_i...A_k$ hesaplama maliyeti $+A_{k+1}...A_j$ hesaplama maliyeti+elde edilen son iki matrisin çarpım maliyeti şeklinde tanımlanır ($C_{i-1}C_kC_j$). Bu tanımdan yola çıkarak

$$M_{sol,sağ} = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \left\{ M_{i,k} + M_{k+1,j} + C_{i-1}C_kC_J \right\} \end{cases}$$

olur.

Bu bağıntı k'yı bildiğimizi kabul eder. k'nın alabileceği değer sayısı j-i olur (k=1,i+1,....j-1).
Optimum çözüm bu değerlerden birini kullanacaktır. Bundan dolayı bu değerler kontrol edilir. Buradan $M_{i,j}$ değerleri, alt problemlerin optimal çözümlerinin maliyetlerini verir. k' nın alacağı değer $S_{i,j}$ olsun. $A_iA_{i+1}.....A_j$ bölünmesinde optimal parantezleme kullanılıyorsa, $S_{i,j}=k$ olur ve öyle ki

$M_{i,j}=M_{i,k}+M_{k+1,j}+C_{i-1}C_kC_j$

24.4. Optimal maliyetin hesaplanması

Amaç az sayıda var olan alt problemleri kullanarak optimal $M_{1,n}$ nin hesaplamaktır ve alt problemlerin durumu seçilen i ve j değerlerine bağlıdır veya

$$\binom{n}{2} + n = \Theta\left(n^2\right)$$

farklı seçilebilecek i ve j değeri vardır.

Özyinelemeli olarak tanımlanan algoritma ağacın değişik dallarında birden fazla aynı alt problem ile karşılaşabilir. Bu durum dinamik programlamanın uygulama gerekçelerinden biridir.

$M_{i,j}$ için verilen tekrarlı bağıntıyı çözmektense, dinamik programlamanın 3. adımı alttan-üste (bottom-up) mantığı ile optimal maliyet hesaplanır. A_i matrisi için boyutlar $P_{i-1}xP_i$, i=1,1....n olur. Bundan dolayı giriş $<P_0,P_1,.....P_n>$ şeklinde bir dizi olur.

Verilecek olan algoritma m[1...n,1..n] yardımcı tablosunu $M_{i,j}$ sıralamak için kullanıyor ve s[1...n,1...n] tablosuna da hangi k değerinin optimal sonuç verdiğini tutmak için kullanılıyor.

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.

$S_{i,j}$ tablosu verilen $A^i...A_j$ çarpımının hangi endeksten bölüneceğini gösterir. Bundan dolayı

$$A1...An = A1....A_{S_{1,n}} A_{S_{1,n}+1}...A_n$$

Algoritma 24.6. Sıra_zincir_matris(P)

▷ $A_1A_2\ldots A_n$ matrislerinin optimal olarak çarpılabilmesi için parantezlemeye karar veren bir algoritmadır. $M_{1,n}$ ve $S_{1,n}$ değerlerini hesaplar.

```
1.  n←uzunluk [p]-1
2.  i←1,…, n kadar
3.     m[i,j]←0
4.  t←2, …, n kadar
5.     i←1,…, n-t+2 kadar
6.        j←i+t-1
7.           m[i,j]←∞
8.           k←i, …, j-1 kadar
9.              q←m[i,k]+m[k+1,j]+Pi-1PkPj
10.             eğer q<m[i,j] ise
11.                m[i,j]←q
12.                s[i,j]←k
```

olur. Bu çarpım özyinelemeli olarak yapılabilir.

$s[1,s[1,n]] \rightarrow A_1 \ldots A_{s[1,n]}$
$s[s[1,n]+1,n] \rightarrow A_{s[1,n]+1} \ldots A_n$

Hala matris çarpımı yapılmadı ve matris çarpımını gerçekleştiren algoritma Algoritma 24./' de görülmektedir.

Algoritma 24.7. Zincirleme_matris_çarpımı(A,s,i,j)

▷ $A_1A_2\ldots A_n$ matrislerinin optimal olarak çarpılmasını yapan bir algoritmadır. Bu işlemi gerçekleştirmek için m ve s tablolarını kullanmaktadır.

```
1.  eğer j>i ise
2.     X←Zincirleme_Matris_Çarpımı(A,s,i,s[i,j])
3.     Y← Zincirleme_Matris_Çarpımı(A,s,s[i,j]+1,j)
3.     Matris_Çarpımı(X,Y)
```

24.5. Dinamik Programlamanın Temelleri

Bir problemin dinamik programlama ile çözülebilmesi için

-Optimal altyapı
-Altproblem örtüşmesi

olmalıdır. Böl ve yönet tasarım yönteminde altproblemlerin bağımsız olmaları gerekirken, bu tasarım yönteminde buna ihtiyaç yoktur.

Bir problemin dinamik programlama ile çözülebilmesi için optimal çözümün yapısını belirlemedir. Optimal çözüm, altproblemlerin optimal çözümlerini içeriyorsa, buna optimal altyapı denir. Optimal altyapı içeren her probleme dinamik programlama uygulanabilme ihtimali vardır. İteratif olarak problem uzayı belirlenebilir.

Dinamik programlamanın uygulanabilmesi için problem uzayının küçük olması gerekir. Çünkü özyinelemeli çalışan bir yöntemde her seferinde yeni altproblemler üretilirse, bu durumda özyinelemeli bir tasarım yöntemi kullanılmamalıdır. Toplam altproblem sayısının giriş verisinin boyutu türünden polinomsal olmalıdır. Özyinelemeli çalışan bir algoritmada tekrar-tekrar aynı altproblem meydana geliyorsa, buna altproblem örtüşmesi denir.

Böl ve yönet yönteminde her adımda yeni altproblemler üretilir.

Örneğin matris carpımında m[3,4], 4 kez ortaya çıkar, bunlar sırasıyla m[2,4], m[1,4], m[3,5] ve m[3,6] şeklinde olur. Bunun için m[i,T] hesaplayan özyinelemeli algaritma Algoritma 24.7' de görüldüğü gibi olur ve her seferinde alt problem çözülüyor.

Bunun için m[i,j] değerini hesaplayan özyinelemeli algoritma Algoritma 24.8' de görülmektedir.

Algoritma 24.8. Özyinelemeli_Matris_Zinciri(P,i,j)

▷ $A_1A_2\ldots A_n$ matrislerinin optimal olarak çarpılabilmesi için parantezlemeye karar veren bir algoritmadır. $M_{1,n}$ ve $S_{1,n}$ değerlerini hesaplar.

1. eğer i=j ise
2. Sonuç←0
3. m[i,j] ← ∞
4. k ←i,…, j-1 kadar
5. q← özyineleme –matris-zinciri (P,i,k)
   +özyineleme –matris-zinciri(P,k+1,j)
   $+P_{i-1}P_kP_j$
6. eğer q<m[i,j] ise
7. m[i,j]←q
8. Sonuç←m[i,j]

Bu özyinelemeli algoritma ile m[1,n] hesaplamak için geçen zaman T(n) olsun. Bu durumda T(1)≥1 olur ve n>1 için

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

olur. i=1,2,…,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.

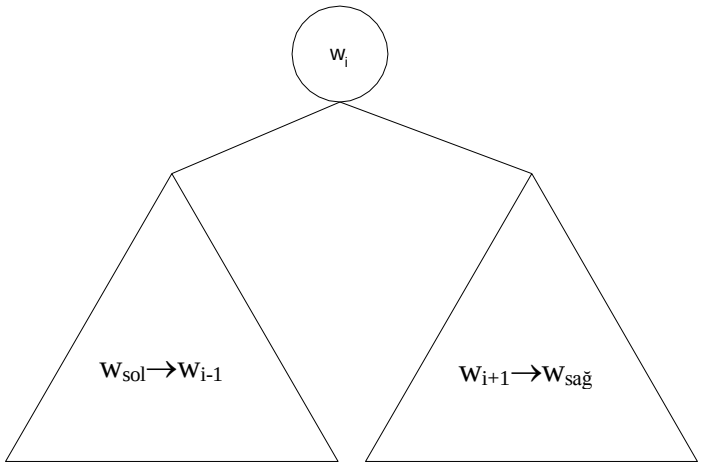
24.6.Optimal İkili Arama Ağacı

$w_1, w_2, \ldots\ldots, w_n$ kelimeleri verilsin ve kelimelerin oluşma olasılıkları sırasıyla $p_1, p_2, \ldots\ldots, p_n$ olsun. Amaç, bu kelimeleri ikili ağaca yerleştirmek ve erişim zamanını minimum yapmaktır. d derinliğindeki bir kelimeyi bulmak için yapılacak karşılaştırma sayısı d+1 olur, böylece eğer w_i kelimesi d_i derinliğine yerleştirildiyse,

$$\sum_{i=1}^{n} p_i(1 + d_i)$$

değeri minimum yapılmalıdır. Bunun için olasılığı en yüksek olan kelime kök düğümüne yerleştirilir. Kelime listesi $w_{sol}, w_{sol+1}, \ldots, w_{sag-1}, w_{sag}$ şeklinde olsun ve bu liste ağaca eklensin. Optimal ikili ağacın kök düğümünde w_i kelimesi olsun ve sol≤i≤sag. Sol alt ağaçta $w_{sol}, w_{sol+1}, \ldots, w_{i-1}$ kelimeleri bulunur ve sağ alt ağaçta ise $w_{i+1}, \ldots, w_{sag}$ kelimeleri bulunur. Bu iki alt ağacında optimal ikili ağaç olması gerekiyor. $C_{sol,sag}$ terimi optimal ikili ağacın maliyetini göstersin.

Eğer sol>sağ ise, ağacın maliyeti 0 olur (bu null olduğu durumdur). Kök düğümün maliyeti p_i olsun. Sol ve sağ alt ağaçların maliyetleri $C_{sol,i-1}$ ve $C_{i+1,sağ}$ olur ve kök düğüme göre hesaplama yapılır.



Şekil 24.3. *w_i kök düğümine sahip optimal ikili arama ağacı.*

Bu durumda

$$\sum_{j=sol}^{i-1} p_j \qquad \text{ve} \qquad \sum_{j=i+1}^{sağ} p_j$$

toplanır ve

$$C_{sol,sağ} = \min_{sol \le i \le sağ} \left\{ p_i + C_{sol,i-1} + C_{i+1,sağ} + \sum_{j=sol}^{i-1} p_j + \sum_{j=i+1}^{sağ} p_j \right\}$$

$$= \min_{sol \le i \le sağ} \left\{ C_{sol,i-1} + C_{i+1,sağ} + \sum_{j=sol}^{sağ} p_j \right\}$$

olur.