Dinamik Programlama

Bölüm 2- Problemler

0/1 Knapsack

- Problem
  - Verilen: *n* tane nesne için ağırlık vektörü $w_1, w_2, \ldots, w_n$ ve değer vektörü $p_1, p_2, \ldots, p_n$, sırtçantası kapasitesi *m*
  - İstenen: Nesnelerden öyle bir altküme oluşturki ağırlık toplamları *m* yi geçmesin ve değerleri toplamı maksimum olsun. Yani, optimizasyon problemi olarak $x_1, x_2, \ldots, x_n$ değerlerini bul öyleki

$$\sum_{i=1}^{n} p_i \cdot x_i$$

Maksimum olsun

$$\sum_{i=1}^{n} w_i \cdot x_i \leq m$$

$$x_i \in \{0, 1\}, 1 \leq i \leq n$$

Kısıtları sağlansın

Kaba kuvvet çözümünün zaman karmaşıklığı: $O(n2^n)$

0/1 Knapsack

- DP çözümü
  - KNAP(1, *n*, *m*) nin optimal çözümü $x_1, x_2, \ldots, x_n$ olsun.
    - Eğer x_n=0 (n. nesne sırtçantasında yok) ise $x_1, x_2, \ldots, x_{n-1}$, KNAP(1, *n*-1, *m*) nin optimal çözümü olmalıdır
    - Eğer x_n=1 (n. nesne sırtçantasında var) ise $x_1, x_2, \ldots, x_{n-1}$, KNAP(1, *n*-1, *m*-w_n) nin optimal çözümü olmalıdır
    - Aksi halde $x_1, x_2, \ldots, x_n$, KNAP(1, *n*, *m*) in optimal çözümü olmaz (Kes-ve-yapıştır).

0/1 Knapsack

- DP çözümü
  - *C[n, m]* değeri KNAP(1, n, m) nin optimal çözümünün toplam değeri olsun.
  - $C[n, m] = max\{C[n-1, m], C[n-1, m-w_n] + p_n\}$ dir.
  - Genelleyecek olursak,

    $C[i, k] = max\{C[i-1, k], C[i-1, k-w_i] + p_i\}$, $i>1$ ve $k>0$

    $C[i, 0] = 0$, $i>1$

    $C[1, k] = 0$, $k>0$ ve $k<w_1$

    $C[1, k] = p_1$, $k>0$ ve $k \geq w_1$

0/1 Knapsack

- DP çözümü
  - Altproblem sayısı: O(nm)
  - Seçim sayısı: O(1)

Çözümün karmaşıklığı polinomsal gibi görünsede aslında eksponansiyeldir. Örneğin 32-bit bir *m* sayısı en fazla 2^{32} değerine sahip olabilir. Bu türden problemlere sözde-polinomsal denir.

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

Dengeli İkiye Bölme

- Problem
  - Verilen: Her biri *0..K* aralığında *n* tane tamsayı: A_1, A_2,..., A_n
  - İstenen: Bu *n* sayıyı iki gruba (S_1 ve S_2) ayır öyleki, |Sum(S_1)-Sum(S_2) | minimum olsun, burada

    Sum(S_i)= $\sum_{A \in S_i} A$
  - Yani, sayıları toplamları birbirine olabildiğince yakın iki gruba ayırmamız istenmektedir.

Kaba kuvvet çözümünün zaman karmaşıklığı: $O(n2^n)$

Dengeli İkiye Bölme

- DP çözümü

$P(i, j) = 1$ $\{A_1, \ldots, A_i\}$ de herhangi bir altkümenin toplamı j ise
$= 0$ aksi halde

$i = 1 \ldots n$
$j = 0..nK$

Özyineli olarak, $P(i, j) = 1$ eğer $P(i-1, j)=1$ veya $P(i-1, j-A_i)=1$
$P(i, j) = 0$ aksi halde
olarak tanımlanır.

Ya da tek bir ifade ile,
$$P(i, j) = \max \{P(i-1, j), P(i-1, j-A_i)\}$$

P(i,j) yi hesaplamanın zaman karmaşıklığı: $O(n^2K)$

Dengeli İkiye Bölme

- DP çözümü
  - $S = (\sum A_i)/2$ olsun. Bu durumda öyle bir altküme bulmalıyız ki toplamı *S* ye en yakın olsun.
  - Yani, aradığımız toplam $\text{argMin}_j \{S\text{-}j : j<S \text{ ve } P(n, j)=1\}$
  - Bu durumda $\text{Sum}(S_1) = j$, ve $\text{Sum}(S_2) = 2S\text{-}j$ dir. $|\text{Sum}(S_1)\text{-Sum}(S_2)| = 2S\text{-}2j = 2(S\text{-}j)$ bulunur.

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

Dengeli İkiye Bölme

- DP çözümü iyileştirme
  - Tüm tabloyu doldurmaya gerek yoktur. Tablonun yarısını doldurmak yeterli yani, j=0..nK/2
  - K yerine max{A_1, A_2, …, A_n} kullanılabilir.
- Bir optimal çözüm bulma
  - Optimal çözümün değerinde kullanılan P(n, j) den başlanarak,
    $P(n, j)$= max {$P(n\text{-}1, j)$, $P(n\text{-}1, j\text{-}A_{n-1})$} takip edilir.
    birinci maksimumsa A_n değeri S_1 de yer almaz, aksi durumda yer alır.

Optimal İkili Arama Ağacı (BST)

- Problem
  - Verilen: n tane sıralı anahtar kümesi (K) $k_1 < k_2 < \cdots < k_n$, ve her biri için aranma ihtimali, yani k_i için p_i.
    - Anahtarlar sıralı verildiği için bu problemde değerleri önemsiz
  - İstenen: Bu anahtar değerlerinden bir ikili arama ağacı (BST) oluştur, öyleki arama maliyetinin beklenen değeri minimum olsun.

Kaba kuvvet çözümünün zaman karmaşıklığı> $O(2^n)$,
çünkü farklı BST sayısı $=\Omega(4^n/n^{3/2})$

Optimal İkili Arama Ağacı (BST)

- Problem
  - BST T de anahtar k_i nin arama maliyeti = depth$_T$(k_i)+1, burada depth$_T$(k_i) = k_i nin T deki derinliği (köke olan uzaklığı) .
  - Minimize edilecek ifade (optimal çözümün değeri) ve buna karşılık gelen T (optimal çözüm)

$$E[T \text{ için arama maliyeti}]$$

$$= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i$$

$$= \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

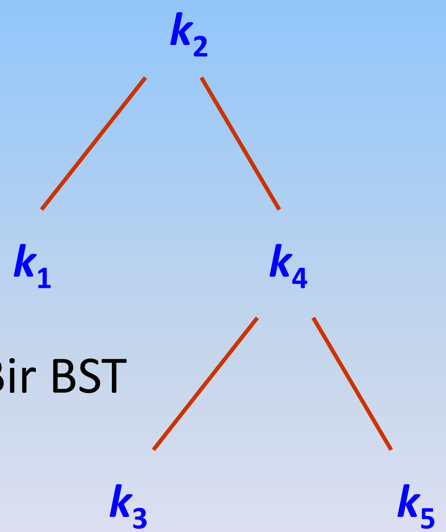
İhtimaller toplamı=1

$$= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i$$

Optimal İkili Arama Ağacı (BST)

- Örnek
  - 5 anahtar ve bunların aranma ihtimalleri

$p_1 = 0.25,\ p_2 = 0.2,\ p_3 = 0.05,\ p_4 = 0.2,\ p_5 = 0.3$



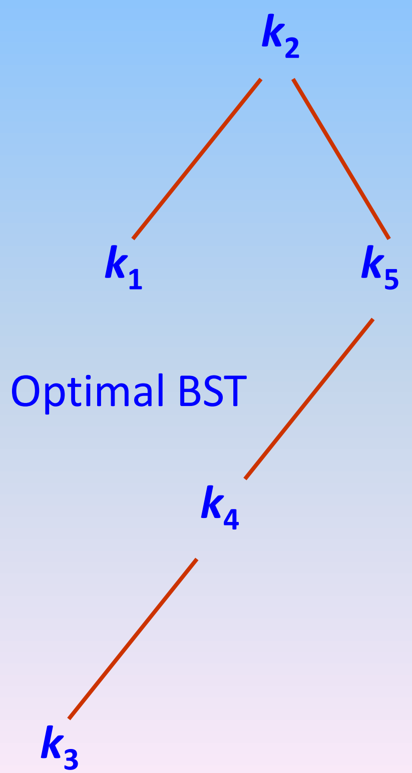
Bir BST

i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Sonuç olarak, E[arama maliyeti] = 2.15

Optimal İkili Arama Ağacı (BST)

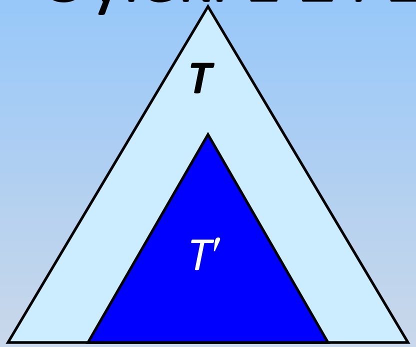
- Bu örnekten gözlemler
  - Optimal BST en küçük yüksekliğe sahip olmak zorunda değil
  - Optimal BST de ihtimali en büyük olan anahtar kök de olmak zorunda değil



Açgözlü (Greedy) bir strateji işe yaramaz

Optimal İkili Arama Ağacı (BST)

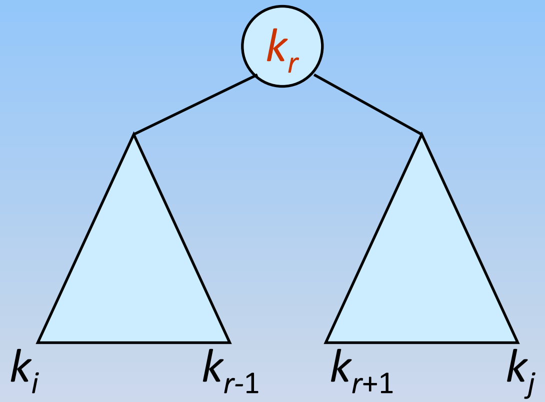
- DP çözümü
  - Bir BST de herhangi bir altağaç takip eden anahtarları içerir. *T'* deki anahtarlar $k_i, \ldots, k_j$ dir. Öyleki $1 \le i \le j \le n$.



  - Eğer T optimal BST ($k_1, \ldots, k_n$ için) ise *T'* de $k_i, \ldots, k_j$ için bir BST olmalıdır (kes-ve-yapıştır).

Optimal İkili Arama Ağacı (BST)

- DP çözümü
  - k_i, …,k_j aralığı için optimal bir BST düşünelim. Optimal BST öyle bir k_r değerini ($i \le r \le j$) kök olarak seçmek zorundadır.
  - Bu seçim geride iki alt problem bırakır. Bunlarında optimal çözülmesi gerekir.
    - k_i,…,k_{r-1} anahtarlarından sol alt ağaç oluştur
    - k_r+1, …,k_j anahtarlarından sağ alt ağaç oluştur



  - Fakat k_r yi bilmiyoruz. Bildiğimiz çok önemli bir şey var alabileceği değerleri biliyoruz. Dolayısıyla tüm hepsini dener en iyisini alırız.

Optimal İkili Arama Ağacı (BST)

- DP çözümü
  - $e[i, j\,]$ değeri $k_i,\ldots,k_j$ için optimal BST nin değeri olsun,
    - $j = i-1$ ise ağaç boş ve $e[i, j\,]=0$
    - $j \geq i$ ise k_r seç ve
      - $k_i,..,k_{r-1}$ den sol alt ağaç oluştur
      - $K_{r+1},..,k_j$ den sağ alt ağaç oluştur

$$e[i,j] = \begin{cases} 0 & \text{eger } \; j = i-1 \\ \min\limits_{i \le r \le j} \{ e[i, r-1] + e[r+1, j] + w(i,j) \} & \text{eger } \; i \le j \end{cases}$$

$$w(i,j) = \sum_{l=i}^{j} p_l$$

İşlem sonrası tüm anahtarların derinliği 1 artar,
Yeni arama maliyeti, iki alt problemin arama maliyeti
artı aralıktaki her bir ihtimalin toplamı olur

Optimal İkili Arama Ağacı (BST)

- DP çözümü
  - $e[1\,..n+1\;,\;0\,..n\;]$ tablosunu doldurmak gerekir. Aradığımız değer $e[1, n]$ dir.
  - Optimal çözümü bulmak için ise bir aralığın optimal bölündüğü r değerini tutmamız yeterlidir. Bunu da root[i, j] tablosunda tutalım.
  - *e* ve *root* tablolarını diyagonal doldurmamız gerekir. Çünkü küçük altproblemler uzunluğu daha kısa altproblemlerdir. Önce 0 uzunluktakiler, sonra 1, daha sonra 2 vb.

Optimal İkili Arama Ağacı (BST)

- DP çözümü

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i–1] ← 0
       w[i, i–1] ← 0
for l ← 1 to n
    do for i ← 1 to n–l + 1
        do j ←i + l–1
           e[i, j ]←∞
           w[i, j ] ← w[i, j–1] + p_j
           for r ←i to j
               do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                  if t < e[i, j ]
                     then e[i, j ] ← t
                          root[i, j ] ←r
return e and root
```

l tane anahtar içeren tüm ağaçlar

başlangıç anahtar indisini sabitle

bitiş anahtar indisini sabitle

Optimal altağacın kökünü belirle

Optimal İkili Arama Ağacı (BST)

- DP çözümü
  - Gözlem: Algoritmada $w(i, j)$ değeri her bir i-j çifti için sıfırdan değil, artımlı olarak aşağıdaki özdeşlikten yararlanılarak hesaplanmıştır, $w(i, j) = w(i, j\text{-}1) + p_j$
    - Bu yapılmamış olsaydı zaman karmaşıklığı O(n) kat artardı
    - Alternatif olarak pre-computation da yapılabilirdi
  - Altproblem sayısı: $O(n^2)$
  - Seçim sayısı: O(n)

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

Optimal Bozukpara Toplama

- Problem
  - Verilen: Bir masa üzerinde çubuk biçiminde yerleştirilmiş *n* (çift) tane bozuk para var. Paraların değeri soldan sağa v_1, v_2, …, v_n. İki kişilik bir oyun oynanıyor, sırası gelen en soldan ya da en sağdan istediği parayı alabiliyor. Oyun paralar bitene kadar devam ediyor. Her iki oyuncunun da amacı aldığı parayı maksimize etmek.
  - İstenen: Maksimum alabileceğiniz para için en iyi oyun stratejisini bulun.

Optimal Bozukpara Toplama

- DP çözümü
  - *V(i, j), $1 \leq i \leq j \leq n$:* Masada kalan paraların en solunda v_i en sağında v_j varken oyun sırası bizdeyse geri kalan paralarda maksimum kazancımız olsun.
  - *V*(*i*, *i*) ve *V*(*i*, *i*+1) temel durumlardır. Eğer oyuna ben başlıyorsam problemin çözümü V(1, n) dir.

$$V(i,j) = \max\left\{ \begin{array}{l} \min\{V(i+1, j-1), V(i+2, j)\} + v_i, \\ \min\{V(i, j-2), V(i+1, j-1)\} + v_j \end{array} \right\} \begin{array}{l} \leftarrow v_i \text{ yi alırsam} \\ \leftarrow v_j \text{ yi alırsam} \end{array}$$

Rakibin de optimal oynadığı varsayılmalı dolayısıyla benim maksimize ettiğim şeyi onun minimize etmesi gerekir.

Optimal Bozukpara Toplama

- DP çözümü
  - Optimal çözüm ifadeyi sırasıyla maksimize ve minimize eden seçimler kaydedilerek belirlenir.
- Altproblem sayısı: $n^2/2$
- Seçim sayısı: 4

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

Optimal Para Üstü

- Problem
  - Verilen/istenen/kısıt: Değeri birbirinden farklı *n* tane bozuk para var: v(1)< v(2)<…< v(*n*), en az sayıda bozuk para kullanarak *C* lik para üstünü oluşturmak.
    - Her bir türden bozuk para sayısı sınırsız
    - Her zaman çözüm olsun diye v(1)=1
  - Örnek1: 1<5<7<12 ve *C*=26 için optimal çözüm 3 bozuk para kullanır (2 tane 7+1 tane 12)
  - Örnek2: 1<5<7<12 ve *C*=21 için optimal çözüm 3 bozuk para kullanır (3 tane 7)
    - Optimal çözümde en yüksek değerli bozuk para olmayabilir. Yani açgözlü (*greedy*) yöntem çalışmaz.

Optimal Para Üstü

- DP çözümü
  - $M(j)$: j lik parayı bozmak için gerekli en az bozuk para sayısı olsun.
  - Temel durum $M(0)=0$

$$M(j) = \min_i \{M(j - v(i)\} + 1$$

$$j - v(i) \geq 0$$

Optimal Para Üstü

- DP çözümü
  - *M* tablosu *j* artan sırada doldurulur
  - Optimal çözümün değeri *M*(*C*) dir
  - Optimal çözümün kendisi seçilen *i* değerlerinin saydırılması ile bulunur
- Altproblem sayısı: *C*
- Seçim sayısı: *n*

Zaman Karmaşıklığı: O(nC)
Hafıza Karmaşıklığı: O(C)

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- Problem
  - Verilen: n uzunluğunda tamsayı listesi $x[1], x[2], \ldots, x[n]$
  - İstenen: En uzun artan altzincir, yani $x[k_1] < x[k_2] < \ldots < x[k_m]$, öyleki $1 \leq k_1 < k_2 < \ldots < k_m \leq n$

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü
  - $L(i)$: $x[1]$, $x[2]$, ..., $x[i]$ için LIS in uzunluğu olsun
  - Temel durum $L(1)=1$

$$L(i) = \max\{0, \max_{j=1}^{j=i-1}\{L(j) : x[j] < x[i]\}\}+1$$

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü
  - Optimal çözümün değeri $L(n)$ dir
  - Optimal çözüm maksimum değeri veren *j* değerleri geriye doğru takip edilerek bulunur
- Altproblem sayısı: *n*
- Seçim sayısı: $O(n)$

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

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü
  - İyileştirme
    - Seçim sayısında tasarruf yapılarak çözümün zaman karmaşıklığı O(n logn) e indirilebilir. Şöyleki,
      Altproblem sayısı: n
      Seçim sayısı: O(logn)

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü (iyileştirilmiş)
  - İki yeni tablo tanımlayalım,
    - $M[j]$: j uzunluğundaki artan altzincirlerden bitiş değeri en küçük $x[k]$ olanın indisi k
    - $P[k]$: $x[k]$ da biten LIS in bir önceki indisi (*optimal çözümün kendisini bulmak için*)
  - Gözlem (herhangi bir L için)
    - $x[M[1]], x[M[2]], ..., x[M[L]]$ değerleri artan sıradadır
    - Dolayısıyla ikili arama yapılabilir

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü (iyileştirilmiş)

L = 0
for *i* = 1, 2, ... n:
  İkili arama ile en büyük $j \leq$ L bul öyleki $x[M[j]] < x[i]$
  (böyle bir değer yoksa $j = 0$)
  P[*i*] = M[*j*]
  if *j* == L or $x[i] < x[M[j+1]]$:
    M[*j*+1] = *i*
    L = max(L, *j*+1)
return *L*

x[i] de biten j+1 uzunluğunda yeni bir artan zincir bulundu.
Eski en küçük ile bitiş değerini karşılaştır.

L uzunluğunda ilk kez artan zincir bulundu

En Uzun Artan Altzincir (Longest Increasing Subsequence, LIS)

- DP çözümü (iyileştirilmiş)
  - Optimal çözümün kendisi (optimum çözümün değeri L ise)
    - ..., x[P[P[M[L]]]], x[P[M[L]]], x[M[L]].

Zincirin bitiş değeri

Zincirin bitişinin bir öncesinin değeri

Optimal çözümü yazdırma zamanı: O(n)

DAG için En Kısa Yol (Shortest Path for DAGs)

- Problem
  - Verilen: Bir yönlü ağırlıklı döngüsüz çizge G=(V,E) ve bir kaynak düğümü $s \in V$
  - İstenen: *s* den diğer tüm düğümlere olan en kısa yol uzunluklarının bulunması
    - Single-Source-Shortest-Path problemi

Şimdiye kadar problemler doğrusal yapılar üzerinden verilmişti. Burada ise problemin tanımı doğrusal değildir fakat doğrusallaştırılabilirdir. Bu problemin dahil edilmesinin nedeni budur.

DAG için En Kısa Yol (Shortest Path for DAGs)

- DP çözümü

```
Dag-Shortest-Paths(G, s)
G nin düğümlerini topolojik olarak sırala
Initialize-Single-Source(G, s)
Her bir düğüm (topolojik sırada) için yap
    Her bir düğüm v ∈ Adj[u] için yap
        Relax(u, v)
```

Zaman Karmaşıklığı: O(V+E)