طراحی الگوریتم ها

۹ مهر

ملکی مجد

برنامه نویسی پویا(ادامه)

Topic	Reference
Recursion and Backtracking	Ch.1 and Ch.2 JeffE
Dynamic Programming	Ch.3 JeffE and Ch.15 CLRS
Greedy Algorithms	Ch.4 JeffE and Ch.16 CLRS
Amortized Analysis	Ch.17 CLRS
Elementary Graph algorithms	Ch.6 JeffE and Ch.22 CLRS
Minimum Spanning Trees	Ch.7 JeffE and Ch.23 CLRS
Single-Source Shortest Paths	Ch.8 JeffE and Ch.24 CLRS
All-Pairs Shortest Paths	Ch.9 JeffE and Ch.25 CLRS
Maximum Flow	Ch.10 JeffE and Ch.26 CLRS
String Matching	Ch.32 CLRS
NP-Completeness	Ch.12 JeffE and Ch.34 CLRS

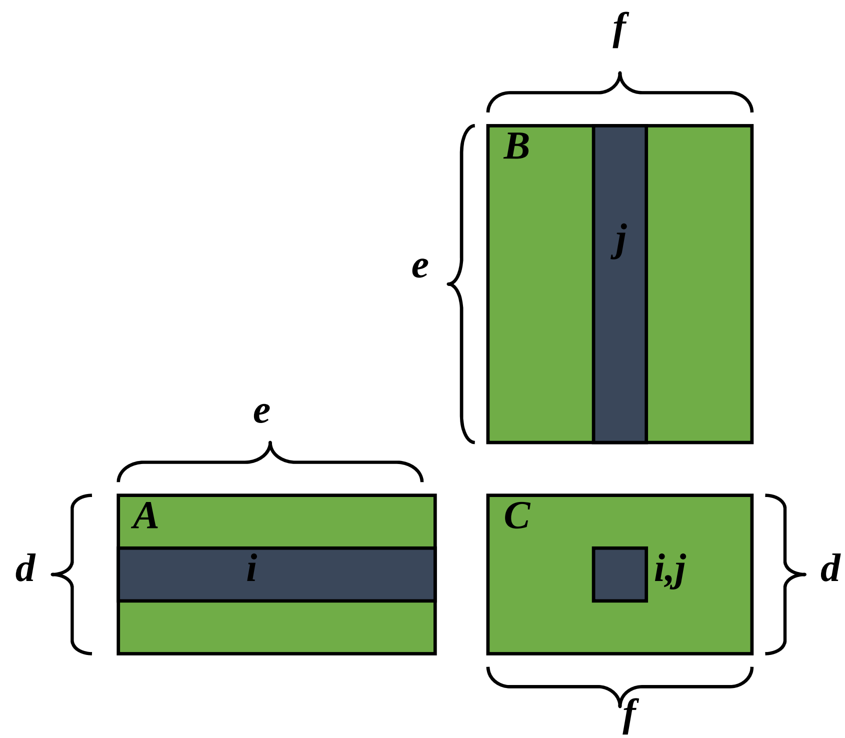
When Dynamic Programming applies?

- Optimal substructure
  - Proof: cut and paste!
  - how many subproblems + how many choices
- Overlapping subproblems
  - the total number of distinct subproblems is a polynomial in the input size.

ضرب ماتریسی

- $C = A_{d\,.\,e} * B_{e\,.\,f}$
- $O(d \cdot e \cdot f\,)$ time

$$C[i, j] = \sum_{k=0}^{e-1} A[i,k] * B[k, j]$$



Matrix-chain multiplication ضرب زنجیره ماتریس

- ترتیب انجام ضرب ماتریسی با پرانتزگذاری مشخص می شود
  - ابعاد ماتریس ها در زنجیره ضرب باید همخوانی داشته باشد
  - نمونه ای از پرانتزگذاری ضرب ۴ ماتریس

$$
\begin{aligned}
&(A_1(A_2(A_3A_4)))\\
&(A_1((A_2A_3)A_4))\\
&((A_1A_2)(A_3A_4))\\
&((A_1(A_2A_3))A_4)\\
&(((A_1A_2)A_3)A_4)
\end{aligned}
$$

Matrix-chain multiplication ضرب زنجیره ماتریس

- پرانتزگذاری بر تعداد ضرب های نهایی تاثیر می گذارد

- A1 with dimensions 10×100
- A2 with dimensions 100×5
- A3 with dimensions 5×50

- ((A1 A2)A3) -> (10.100.5)+(10.5.50)=7500
- (A1(A2 A3)) -> (100.5.50)+(10.100.50)=75000

Matrix-chain multiplication ضرب زنجیره ماتریس
(تعداد حالت های مختلف پرانتزگزاری)

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

$\boxed{\textbf{\textit{Catalan numbers}}\text{, which grows as } \Omega(4^n/n^{3/2})}$

Matrix-chain multiplication ضرب زنجیره ماتریس

- ساختار پرانتزگذاری بهینه:
- برای محاسبه ضرب ماتریس های $A_i A_{i+1} \ldots A_j$ ابتدا برای یک k ضرب $A_i \ldots A_{k-1}$ و $A_k \ldots A_j$ محاسبه می شوند
  - $A_i \ldots A_{k-1}$ و $A_k \ldots A_j$ باید به صورت بهینه پرانتزگذاری شده باشند
  - چرا؟ (قاعده cut and paste)

Matrix-chain multiplication ضرب زنجیره ماتریس

- رابطه بازگشتی (محاسبه ضرب ماتریس های $A_i \; A_{i+1} \cdots A_j$)

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \, , \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \, . \end{cases}$$

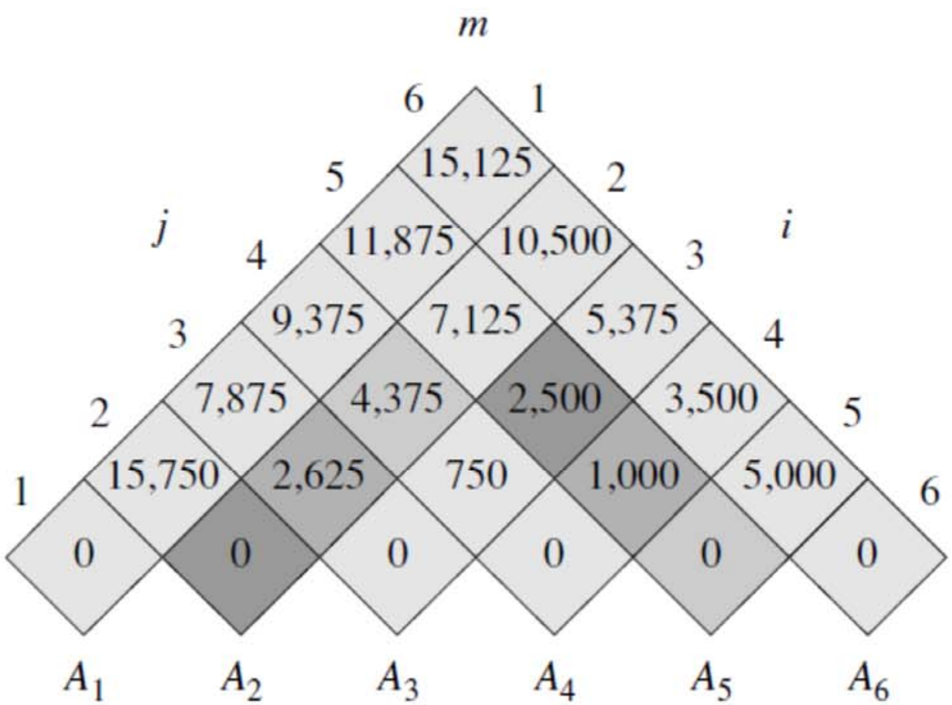
Matrix-chain multiplication ضرب زنجیره ماتریس
محاسبه مقدار بهینه

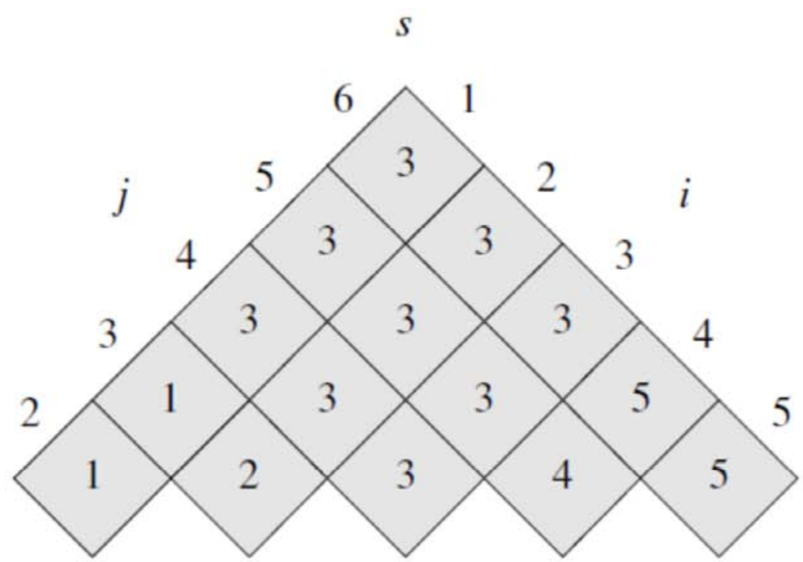
```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
     do m[i, i] ← 0
for l ← 2 to n                    ▷ l is the chain length.
     do for i ← 1 to n − l + 1
            do j ← i + l − 1
               m[i, j] ← ∞
               for k ← i to j − 1
                   do q ← m[i, k] + m[k +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

تعیین مقدار $i \le k \le j$ برای تقسیم زنجیره ماتریس های $A_i \dots A_j$

Matrix-chain multiplication ضرب زنجیره ماتریس

مثال





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

Matrix-chain multiplication ضرب زنجیره ماتریس
محاسبه مقدار بهینه

- تعداد ضرب ها را محاسبه می کنیم (نه اینکه مقدار ضرب را محاسبه کنیم)
- الگوریتم از مرتبه زمانی $O(n^3)$ است و فضای مورد نیاز $O(n^2)$ است

چاپ پرانتز گذاری ضرب ماتریس ها

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
    then print "A"_i
    else print "("
         PRINT-OPTIMAL-PARENS(s, i, s[i, j])
         PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
         print ")"
```

Longest common subsequence

- *S*1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- *S*2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA

- GTCGTCGGAAGCCGGCCGAA

Longest common subsequence

Characterizing a longest common subsequence

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Longest common subsequence

A recursive solution

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

Longest common subsequence

Computing the length of an LCS

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
             do if x_i = y_j
                   then c[i, j] ← c[i − 1, j − 1] + 1
                        b[i, j] ← "↖"
                   else if c[i − 1, j] ≥ c[i, j − 1]
                           then c[i, j] ← c[i − 1, j]
                                b[i, j] ← "↑"
                           else c[i, j] ← c[i, j − 1]
                                b[i, j] ← "←"
return c and b
```

Longest common subsequence

Computing the length of an LCS

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

Look-up time
Optimal Binary Search Trees

- designing a program to translate text from English to French
  - For each occurrence of each English word in the text, we need to look up its French equivalent
- total time spent searching to be as low as possible
  - ensure an $O(\lg n)$ search time per occurrence by using a red-black tree
- case that a frequently used word such as "the" appears far from the root while a rarely used word such as "mycophagist" appears near the root
  - slow down the translation

Optimal Binary Search Trees

given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order

$k_1 < k_2 < \cdots < k_n$

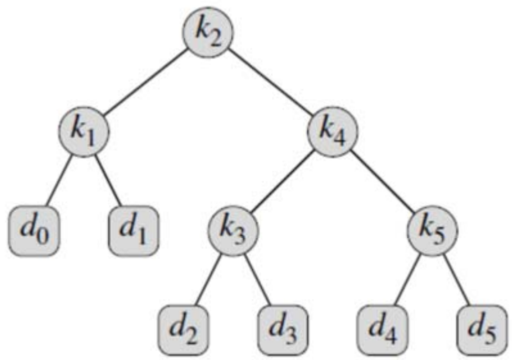
$d_0, d_1, d_2, \ldots, d_n$ representing values not in K

d_i represents all values between k_i and k_{i+1}

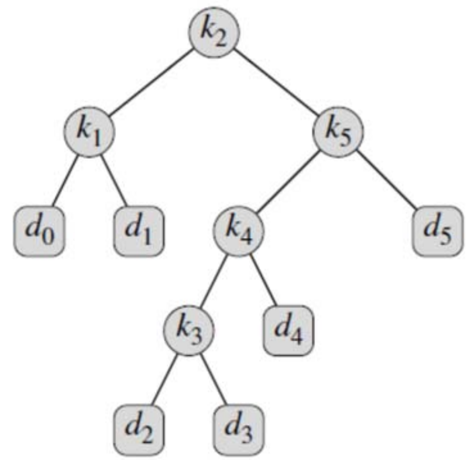
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

$$\begin{aligned} \mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \ , \end{aligned}$$

Optimal Binary Search Trees example



expected search cost 2.80.



expected search cost 2.75.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal Binary Search Trees

The structure of an optimal binary search tree (1)

range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$

a subtree — keys $k_i, \ldots, k_j$; leaves — dummy keys $d_{i-1}, \ldots, d_j$

- if an optimal BS tree *T* has a subtree T' containing keys *i to j*
  - then this subtree T' must be optimal as well
  - cut-and-paste argument applies

Optimal Binary Search Trees

The structure of an optimal binary search tree (2)

$k_i, \ldots, k_j$, one of these keys, say k_r $(i \le r \le j)$, will be the root of an optimal subtree

left subtree: $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$)

right subtree: keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$)

k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$

keys $k_i, \ldots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}

Optimal Binary Search Trees
A recursive solution (1)

- e[1 , n]
- easy case occurs when $j = i - 1$.

Optimal Binary Search Trees

A recursive solution (2)

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \qquad w(i, j) = w(i, r-1) + p_r + w(r+1, j)$$

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) .$$

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

Optimal Binary Search Trees

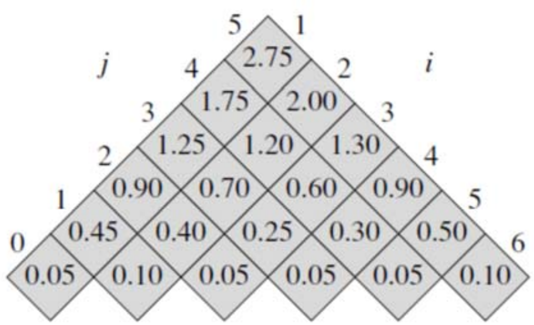
Computing the expected search cost of an optimal binary search tree

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i − 1] ← q_{i−1}
       w[i, i − 1] ← q_{i−1}
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 + q_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
```

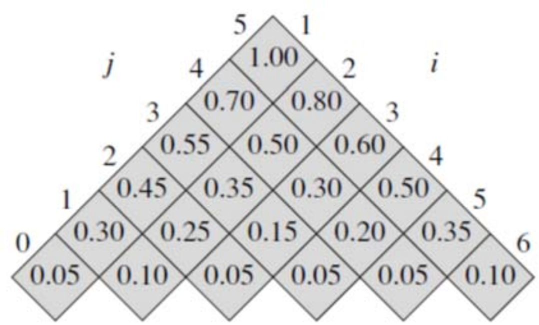
Optimal Binary Search Trees

example

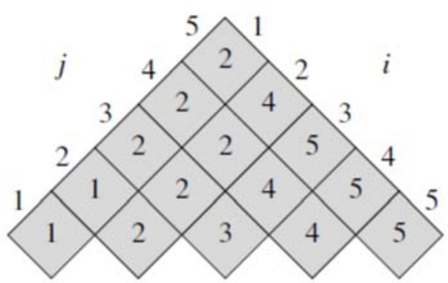
e



w



root



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10