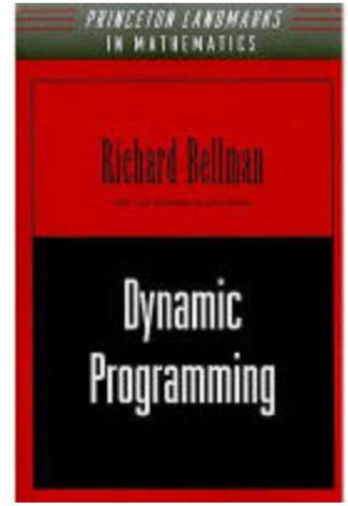
15 Dynamic Programming

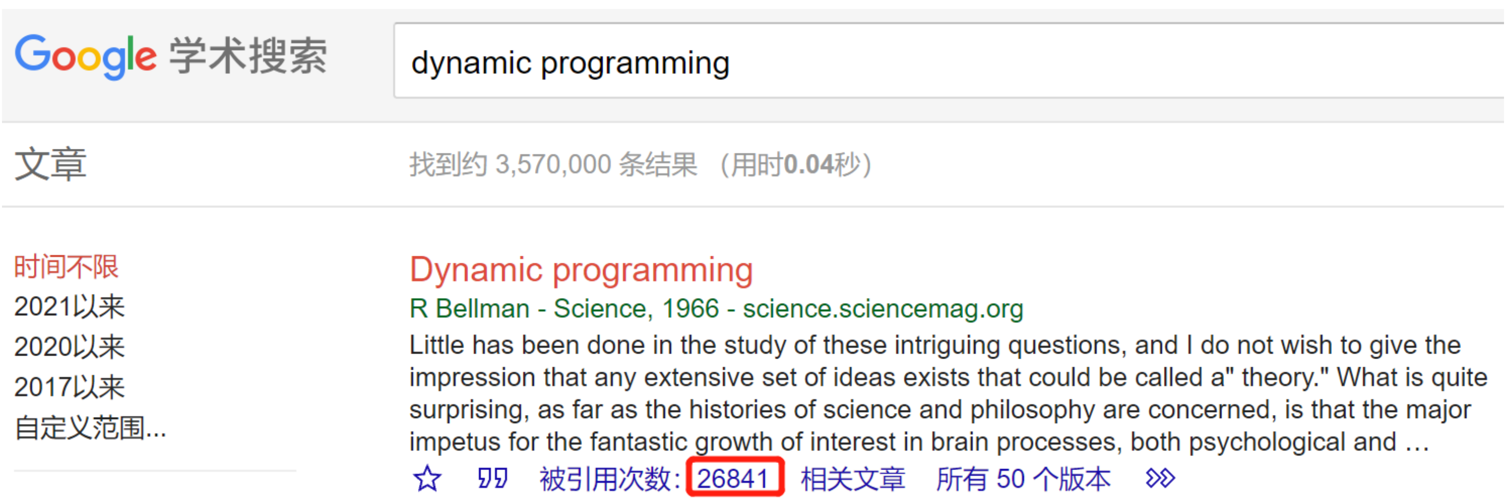
15 Dynamic Programming

Richard Bellman.
Dynamic Programming.
Princeton University Press, 1957.



¥411.00

预订 Dynamic Programming 原版新书



15 Dynamic Programming

- ✓ **Scheduling two automobile assembly lines**
- ✓ **Steel rod cutting (15.1)**
- ✓ **Matrix-chain multiplication (15.2)**
  矩阵链相乘，或矩阵连乘问题
- ✓ **Characteristics(Elements) of dynamic programming (15.3)**
- ✓ **Longest common subsequence (15.4)**
- ✓ **Optimal binary search trees (15.5)**

15.2 Matrix-chain multiplication (MCM)

- **Given a sequence (chain) $<A_1, A_2, \ldots, A_n>$ of n matrices to be multiplied, and we wish to compute the product** **n 个矩阵相乘，称为‘矩阵连乘’，如何求积？**

$$A_1A_2A_3A_4 \qquad (15.10)$$

$$(A_1(A_2(A_3A_4))),\ (A_1((A_2A_3)A_4)),\ ((A_1A_2)(A_3A_4)),\ \ldots\ldots$$

- **We can evaluate (15.10) using the <u>standard algorithm</u> for multiplying pairs of matrices as a subroutine once we have parenthesized it.**

- **A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Example, A_1, $(A_1((A_2A_3)A_4))$, $(A_1((A_2A_3)(A_4A_5)))$.**

矩阵连乘全括号：仅有一个矩阵，或者两个“矩阵连乘全括号”的乘积且外层包括一个括号，如： $(A_1((A_2A_3)A_4))$

这是嵌套的矩阵对，它给出了 矩阵连乘的一种求解顺序，也简称“矩阵全括号”。

Example: Multiplication of two matrices（矩阵相乘）

two $n \times n$ matrices A and B, Complexity(C=A×B) = ?

Standard method

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{pmatrix} \cdots\cdots\cdots \; . \; .. \\ \cdots\cdots\cdots c_{ij} \, .. \\ \cdots \\ \cdots\cdots\cdots \; . \; .. \end{pmatrix} = \begin{pmatrix} \cdots\cdots\cdots\cdots \\ \#\#\ldots\# \\ \cdots \\ \cdots\cdots\cdots\cdots \end{pmatrix} * \begin{pmatrix} \cdots\cdots\cdots\#.. \\ \cdots\cdots\cdots\#.. \\ \cdots \\ \cdots\cdots\cdots\#.. \end{pmatrix}$$

MATRIX-MULTIPLY(A, B)
for $i \leftarrow 1$ **to** n
  for $j \leftarrow 1$ **to** n
    $C[i, j] \leftarrow 0$
    for $k \leftarrow 1$ **to** n
      $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
return C

Complexity:
$O(n^3)$ multiplications and additions.
$T(n) = O(n^3)$.

15.2 Matrix-chain multiplication

- Given a sequence (chain) $\langle A_1, A_2, ..., A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 A_3 A_4 \,. \tag{15.10}$$

- Matrix multiplication is associative, so all parenthesizations yield the same product. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways:
矩阵连乘满足结合律，因此对所有加括号的方式，矩阵连乘的积相同。例如…
$(A_1 (A_2 (A_3 A_4)))$, $(A_1 ((A_2 A_3) A_4))$, $((A_1 A_2) (A_3 A_4))$,
$((A_1 (A_2 A_3)) A_4)$, $(((A_1 A_2) A_3) A_4)$.

15.2 Matrix-chain multiplication

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

采用不同的加括号方式，可导致差异极大的乘法开销

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

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

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

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

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

15.2 Matrix-chain multiplication

- **First, consider the cost of multiplying two matrices.**
- **Two matrices *A* and *B* can be multiplied only if they are compatible: columns of *A* = rows of *B*.** **（仅当矩阵*A*和*B*相容时，*A*和*B*能相乘）**
  - **If *A* is $p \times q$, *B* is $q \times r$, then *C* is $p \times r$.**
  - **The time to compute *C* is dominated by the number of scalar multiplications in line 7, which is *pqr*.**

$$A * B \Rightarrow C$$

$$\begin{pmatrix} \cdots\cdots\cdots \\ \#\#\ldots\# \\ \cdots\cdots\cdots \\ \cdots\cdots\cdots \end{pmatrix}_{p*q} * \begin{pmatrix} \cdots\cdots\#\ldots \\ \cdots\cdots\#\ldots \\ \cdots \\ \cdots\cdots\#\ldots \end{pmatrix}_{q*r} \Rightarrow \begin{pmatrix} \cdots\cdots\ .\ .. \\ \cdots\cdots c_{ij}\ldots \\ \cdots \\ \cdots\cdots\ .\ .. \end{pmatrix}_{p*r}$$

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then return "error: incompatible dimensions"
3  else for i ← 1 to rows[A]        // p is row[A]
4              for j ← 1 to columns[B]  // r is columns[B]
5                   C[i, j] ← 0
6                   for k ← 1 to columns[A]  // q is columns[A]
7                         C[i, j]←C[i, j]+A[i, k]·B[k, j]
8  return C
```

15.2 Matrix-chain multiplication

- For $A_{p\times q}$, $B_{q\times r}$, $C=AB$ is $p\times r$. The # of scalar multiplications is pqr .
- Consider the problem of a chain $<A_1, A_2, A_3>$,
  Suppose that A_1: 10×100; A_2: 100×5; A_3: 5×50
  - If $A = ((A_1A_2)A_3)$,
    a) $C = A_1A_2$, # of multiplications 10 ·100 · 5 = 5000, $C_{10\times 5}$
    b) $A = CA_3$, # of multiplications 10 · 5 · 50 = 2500, $A_{10\times 50}$
    then, # of scalar multiplications, for a total of 7500.
  - If $A = (A_1(A_2A_3))$,
    a) $C_{100\times 50} = A_2A_3$, # of multiplications 100 ·5·50 = 25,000 ,
    b) $A_{10\times 50} = A_1C$, # of multiplications 10·100·50 = 50,000 ,
    then, # of scalar multiplications, for a total of 75,000 .
  - The first case is 10 times faster than the second.

7,500 << 75,000

15.2 Matrix-chain multiplication

$A_1 A_2 A_3 A_4 A_5$: $(A_1 A_2 A_3)(A_4 A_5)$? $(A_1 A_2)(A_3 A_4 A_5)$? ……

$A_1 (A_2 A_3)$? $(A_1 A_2) A_3$?

- **Matrix-chain multiplication problem** : Given a chain $< A_1, A_2, \ldots, A_n >$, i=1, 2, ..., n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

- In the problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Counting the number of parenthesizations

$$A_1\, A_2\, A_3\, A_4\, A_5$$

$(\,\underline{(\,A_1\, A_2\, A_3\,)}\,(A_4\, A_5)\,)$? $(A_1\, A_2)\,(A_3\, A_4\, A_5)$? ……

$(\,A_1\,(A_2\, A_3)\,)$? $((A_1\, A_2)\, A_3\,)$?

$$(\;\underline{(\,A_i\,(A_{i+1}\ldots)\,(\ldots)\ldots A_k\,)}\;\underline{(\,A_{k+1}\ldots\;A_{j-1}\,A_j)}\;)$$

- **Brute force, exhaustively checking all possible parenthesizations.**
- **$P(n)$: the # of alternative parenthesizations of n matrices.** $P(n)$种全括号方式
  - **n=1, one matrix, one way to parenthesize the matrix product.**
  - **$n \geq 2$,**

$$P(n) = \begin{cases} 1 & , \quad \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{if } n \geq 2. \end{cases} \qquad (15.11)$$

- **The solution to (15.11) is $\Omega(2^n)$ (guess, then prove), a poor strategy.**

Counting the number of parenthesizations

$$A_1\,A_2\,A_3\,A_4\,A_5$$

$$(\,(\underline{\,(A_1\,A_2\,A_3)\,}\,(A_4\,A_5)\,)? \qquad (A_1\,A_2)\,(A_3\,A_4\,A_5)? \quad \ldots\ldots$$

$$(\,A_1\,(A_2\,A_3)\,)? \qquad ((A_1\,A_2)\,A_3\,)?$$

$$(\;\underline{(\,A_i\,(A_{i+1}\cdots)\,(\ldots)\ldots A_k\,)}\;\underline{(\,A_{k+1}\cdots\;A_{j-1}\,A_j)}\;)$$

$$P(n)=\begin{cases}1 & ,\ \text{if } n=1,\\ \displaystyle\sum_{k=1}^{n-1}P(k)P(n-k), & \text{if } n\ge 2.\end{cases} \qquad (15.11)$$

4 Recurrences

Algorithms analysis

recursion

h(n) = h(0)*h(n-1)
+ h(1)*h(n-2) + ...
+ h(n-1)h(0)

另类递归式:

h(n) = ((4*n-2)/(n+1))*h(n-1)

该递推关系的解为:

h(n) = C(2n, n)/(n+1)

其中，规定h(0) = 1

catalan数，卡特兰数，是一个常出现在各种计数问题中的数列，以比利时的数学家欧仁-查理-卡特兰命名

E Zexal的二叉树（签到）

时间限制: 1000ms 内存限制: 65536kb

通过率: 200/209 正确率: 200/596

题目

知识点: 树，数论，dp，递归（都可以做）

上学期我们学习了二叉树，也都知道3个结点的二叉树有5种，现给你二叉树的结点个数n，要你输出不同形态二叉树的种数。

输入

第一个数为一个整数n(n <= 30)

输出

对于每组数据，输出一行，不同形态二叉树的种数。

输入样例

输出样例

Dynamic Programming to solve MCM:

Four Steps

Step 1: The structure of an optimal parenthesization

$((\underline{A_1 A_2 A_3})(A_4 A_5))$? $(A_1 A_2)(A_3 A_4 A_5)$? ……

$(A_1 (A_2 A_3))$? $((A_1 A_2) A_3)$?

$$(\underline{(A_i (A_{i+1} \ldots)(\ldots)\ldots A_k)}\ \underline{(A_{k+1} \ldots\ A_{j-1} A_j)})$$

Find the optimal substructure

$A_{i..j}$ (where $i \le j$) : the product $A_i A_{i+1} \ldots A_k A_{k+1} \ldots A_j$

- **$i < j$, nontrivial, any parenthesization of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$.**
- **First compute $A_{i..k}$, and $A_{k+1..j}$, then $A_{i..k} \cdot A_{k+1..j} = A_{i..j}$**

The cost of this parenthesization $A_i A_{i+1} \ldots A_j$

= the cost of computing the matrix $A_{i..k}$ + the cost of computing $A_{k+1..j}$

+ the cost of multiplying $A_{i..k} \cdot A_{k+1..j}$

Step 1: The structure of an optimal parenthesization

The optimal substructure

- **Suppose that an optimal parenthesization of $A_iA_{i+1}\ldots A_j$ splits the product between A_k and A_{k+1}.**

$$\Big(\ \underline{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}\ \underline{(A_{k+1}\ldots\ A_{j-1}A_j)}\ \Big)$$

设矩阵连乘的最佳全括号将矩阵连乘分成 $A_{i..k}$ 和 $A_{k+1..j}$ 两部分之积

- **The parenthesization of the "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$?**

$A_{i..j}$ 的最佳全括号中的 $A_{i..k}$ 的全括号必定是 $A_{i..k}$ 的最佳全括号

$$\underbrace{\Big(\ \underbrace{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}_{M}\ (A_{k+1}\ldots\ A_{j-1}A_j)\ \Big)}_{X}$$

Proof

如果 X 最优，则 M 最优

Step 1: The structure of an optimal parenthesization

The optimal substructure

- **The parenthesization of the "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$?**

  Proof Optimal parenthesization $A_{i..j} = (A_i\ldots A_k)\,(A_{k+1}\ldots A_j) = M\cdot N$, parenthesization $M =(A_i\, A_{i+1}\ldots A_k)$ in $A_{i..j}$ above.

  If there were a less costly way to parenthesize $A_i\ldots A_k=P$, substituting M with P, that is $P\cdot N$ would produce another parenthesization of $A_{i..j}$ whose cost was lower than $M\cdot N$.

  A contradiction.

- **A similar observation holds for $A_{k+1}A_{k+2}\ldots A_j$**

Step 1: The structure of an optimal parenthesization

$$\underbrace{\Big(\underbrace{(A_i(A_{i+1}\dots)(\dots)\dots A_k)}_{M}(A_{k+1}\dots A_{j-1}A_j)\Big)}_{X}$$

- **Construct an optimal solution to the problem *X* from optimal solutions to subproblems *M* based on optimal substructure.**

from *M* to *X*

- **Any solution to the matrix-chain multiplication requires us to split the product, and any optimal solution contains within it optimal solutions to subproblem instances.**
  - **Split the problem into two subproblems (optimally parenthesizing $A_iA_{i+1}\dots A_k$ and $A_{k+1}A_{k+2}\dots A_j$);**
  - **find optimal solutions to subproblem *M*;**
  - **combine these optimal subproblem solutions (from *M* to *X*)**

Step 1: The structure of an optimal parenthesization

$$A_1 A_2 A_3 A_4 A_5$$

$((\underline{A_1 A_2 A_3})(A_4 A_5))$? $(A_1 A_2)(A_3 A_4 A_5)$? ……

$(A_1 (A_2 A_3))$? $((A_1 A_2) A_3)$?

$$(\underline{(A_i (A_{i+1}\ldots)(\ldots)\ldots A_k)}\ \underline{(A_{k+1}\ldots\ A_{j-1} A_j)})$$

We must consider all possible places so that we are sure of having examined the optimal one.

需要考虑所有分割位置以确保最优解是其中之一

Step 2: A recursive solution

$$\underline{(\ \underline{(A_i (A_{i+1} \ldots)(\ldots) \ldots A_k)} (A_{k+1} \ldots\ A_{j-1} A_j)\ )}$$

- **Define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.**
  根据子问题的最优解可以递归地定义原问题的最优解

- **Subproblems $A_{i..j}$: determining the minimum cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 \le i \le j \le n$.**

  Not $A_1 A_2 \ldots A_j$, Why?

Step 2: A recursive solution

$$\underbrace{\big(\underbrace{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}_{M}(A_{k+1}\ldots A_{j-1}A_j)\big)}_{X}$$

$m[i, j]$ = $|X|$:

the minimum # of scalar multiplications to compute $A_{i..j}$;

the cost of a cheapest way to compute $A_{1..n}$ is $m[1, n]$.

- **If $i = j$, one matrix $A_{i..i} = A_i$, no scalar multiplications. Thus, $m[i, i] = 0$ for $i = 1, 2, \ldots, n$.**
- **When $i < j$?**

Step 2: A recursive solution

$$\underline{\Big(\ \underline{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}\ (A_{k+1}\ldots\ A_{j-1}A_j)\ \Big)}$$

$m[i, j]$:

When $i<j$, assuming that the optimal parenthesization **splits** $A_i A_{i+1} \ldots A_j$ **between** A_k **and** A_{k+1}, where $i \le k < j$.

Then,

$$\boxed{m[i, j] = m[i, k]+m[k+1, j]+p_{i-1}p_kp_j}$$

$\Updownarrow$

$m[i,j]$ = the minimum cost for computing $A_{i..k}$
+ the minimum cost for computing $A_{k+1..j}$
+ the cost of multiplying $A_{i..k}$ and $A_{k+1..j}$

A_i has dimensions $p_{i-1}\times p_i$, then $A_{i..k}$ has $p_{i-1}\times p_k$, $A_{k+1..j}$ has $p_k\times p_j$.

Step 2: A recursive solution

$$\underline{\underline{(\ (A_i\,(A_{i+1}\ldots)\ (\ldots)\ldots A_k)}\ (A_{k+1}\ldots\ A_{j-1}\,A_j)\ )}$$

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

- **This recursive equation assumes that we know the value of *k*, which we actually do not know.**
- **Only *j*-*i* possible values for *k*, namely *k* = *i*, *i*+1, ... , *j*-1.**
- **Checking them all to find the best *k*, we have**

$$m[i, j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1, j] + p_{i-1}p_k p_j\}, & \quad \text{if } i < j. \end{cases} \qquad (15.12)$$

- **The $m[i, j]$ give the costs of optimal solutions to subproblems.**

Step 2: A recursive solution

$$\underline{\color{red}{(}\ \underline{\color{green}{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}}\ (A_{k+1}\ldots A_{j-1}A_j)\ \color{red}{)}}$$

$$m[i,j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \quad \text{if } i < j. \end{cases} \qquad (15.12)$$

Construct an optimal solution :

Define $s[i, j]$ to be a value of k at which we can split the product $A_i A_{i+1} \ldots A_j$ to obtain an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_k p_j$.

Step 3: Computing the optimal costs

$$m[i, j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\}, & \quad \text{if } i < j. \end{cases} \qquad (15.12)$$

recursive algorithm based on recurrence (15.12)?

***m*[1, *n*] for multiplying $A_1 A_2 \ldots A_n$.**

```
RE-MCM(p, i, j)
if i = j
    return 0
m[i, j] ← ∞
for k←i to j-1
    q ← RE-MCM(p, i, k) + RE-MCM(p, k+1, j) + p_{i-1} p_k p_j
    if q < m[i, j]
        m[i, j] ← q
return m[i, j]
```

Running time?

Step 3: Computing the optimal costs

$$m[i,j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \quad \text{if } i < j. \end{cases} \qquad (15.12)$$

RE-MCM(p, i, j)	$T(n)$
if $i = j$	1
return 0	1
$m[i, j] \leftarrow \infty$	1
for $k \leftarrow i$ **to** j-1	1
$q \leftarrow$ RE-MCM(p, i, k)	$T(k)$
+ RE-MCM(p, k+1, j)	$T(n$-$k)$
$+ p_{i-1}p_k p_j$	1
if $q < m[i, j]$	1
$m[i, j] \leftarrow q$	1
return $m[i, j]$	1

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

$$= 2\sum_{i=1}^{n-1} T(i) + n \ \ge \ 3^n$$

Proof **let** $T(i) \ge 3^i$ **,**

then $T(n) \ge 2\sum_{i=1}^{n-1} T(i) + n$

$$\ge 2 \times (3^1 + 3^2 + \cdots + 3^{n-1}) + n$$

$$\ge 2 \times 3\frac{3^{n-1} - 1}{3 - 1} + n$$

$$= 3^n - 3 + n \ge 3^n$$

This algorithm takes **exponential** time, which is no better than the brute-force method of checking each way of parenthesizing the product. Why?

$$P(n) = \begin{cases} 1 & , \quad \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \quad \text{if } n \ge 2. \end{cases} \qquad (15.11)$$

$\Omega(2^n)$

Step 3: Computing the optimal costs

$$(\ \underline{(\boldsymbol{A_i}(\boldsymbol{A_{i+1}}\ldots)(\ldots)\ldots\boldsymbol{A_k})}(\boldsymbol{A_{k+1}}\ldots\ \boldsymbol{A_{j-1}}\boldsymbol{A_j})\ )$$

$$m[i,j]=\begin{cases}0 & ,\quad \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}, & \text{if } i<j.\end{cases} \qquad (15.12)$$

Recursion, Extremely slow!

# of subproblems: one problem for each choice of *i* and *j* satisfying $1 \le i \le j \le n$?
（所有子问题个数为 ?）

$$C_n^2+n=\Theta(n^2)$$

$$1\le i=j\le n,\ C_n^1=n$$

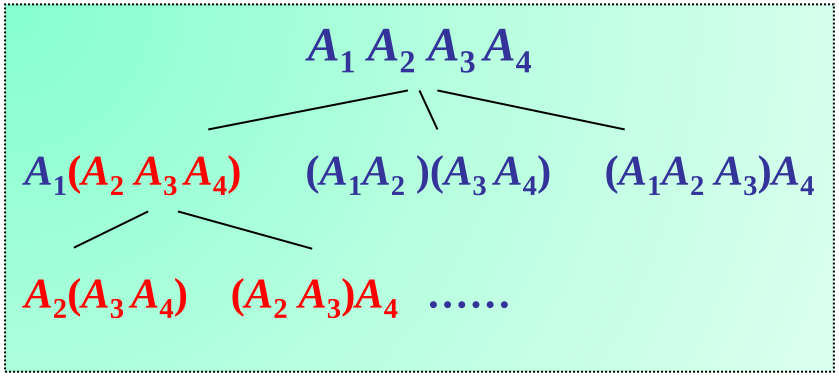
$$1\le i<j\le n,\ C_n^2=n(n-1)/2$$

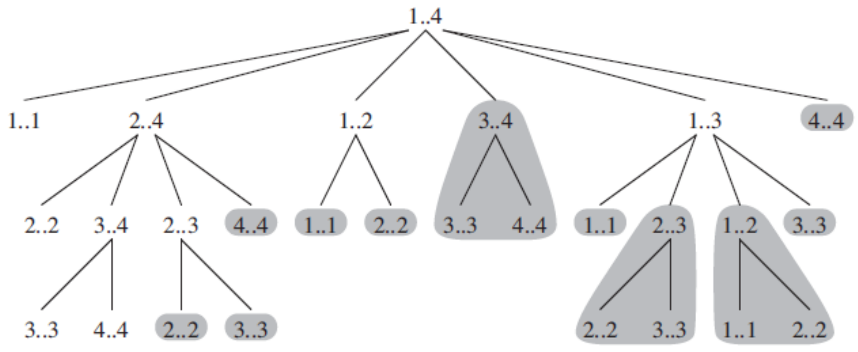
Step 3: Computing the optimal costs

$$\underline{\Big(\ \underline{\big(A_i\,(A_{i+1}\ldots)\,(\ldots)\ldots A_k\big)}\ \big(A_{k+1}\ldots\ A_{j-1}\,A_j\big)\ \Big)}$$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

- **A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree.**
- **Overlapping subproblems: the second hallmark of the applicability of dynamic programming.**



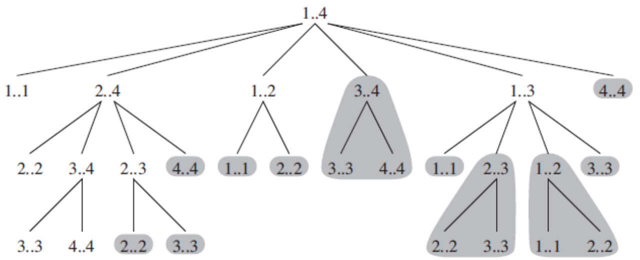


Step 3: Computing the optimal costs

$$\underline{\underline{(\ (A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}\ (A_{k+1}\ldots\ A_{j-1}A_j)\ )}$$

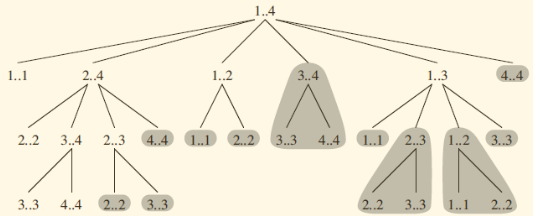
$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}, & \text{if } i<j.\end{cases} \qquad (15.12)$$

- **# of subproblems: one problem for each choice of *i* and *j* satisfying $1 \le i \le j \le n$, or $\binom{n}{2}+n=\Theta(n^2)$ in all.** （子问题个数为$\Theta(n^2)$）
- **Instead of recursive method, computing the optimal cost by using a tabular, bottom-up approach.**
  不用递归方法，而采用列表方式、自底向上的方法计算最优解



Step 3: Computing the optimal costs

$$\Big(\ \underline{(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k)}\ (A_{k+1}\ldots\ A_{j-1}A_j)\ \Big)$$



$$m[i,j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

A_i : dimensions $p_{i-1} \times p_i$

Input: $p = \langle p_0, p_1, \ldots, p_n \rangle$.

Procedure: Table $m[1..n, 1..n]$ storing the $m[i, j]$ costs;

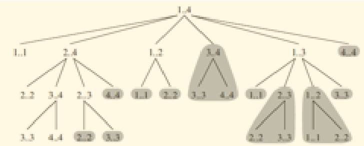
Auxiliary table $s[1..n, 1..n]$ recording which index of k achieved the optimal cost in computing $m[i, j]$.

计算 $m[i, j]$ 时，用辅助表项(table entry) $s[i, j]$ 来记录最佳位置 k 的值

```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
      m[i, i] ← 0
for l←2 to n                    // l is the chain length.
      for i ← 1 to n - l + 1
            j ← i + l – 1
            m[i, j] ← ∞
            for k ← i to j - 1
                  q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                  if q < m[i, j]
                        m[i, j] ← q
                        s[i, j] ← k
return m and s
```

Step 3: Computing the optimal costs

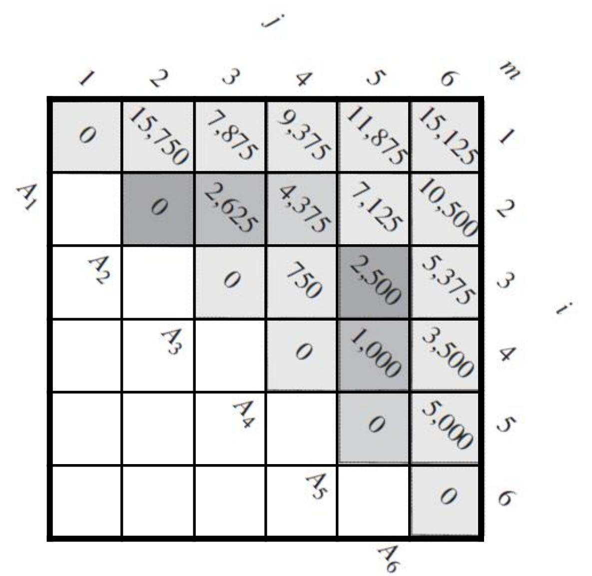
$$\left(\left(A_i (A_{i+1} \ldots) (\ldots) \ldots A_k \right) \left(A_{k+1} \ldots\ A_{j-1} A_j \right) \right)$$

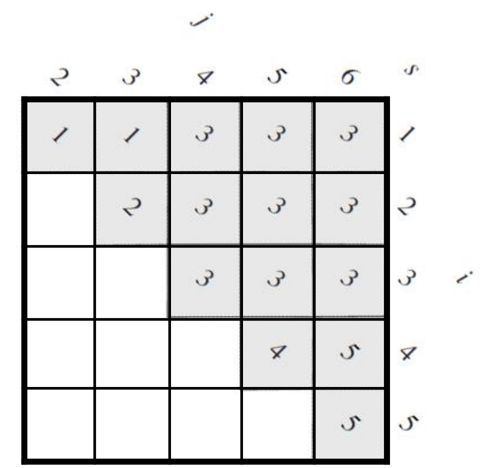


$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

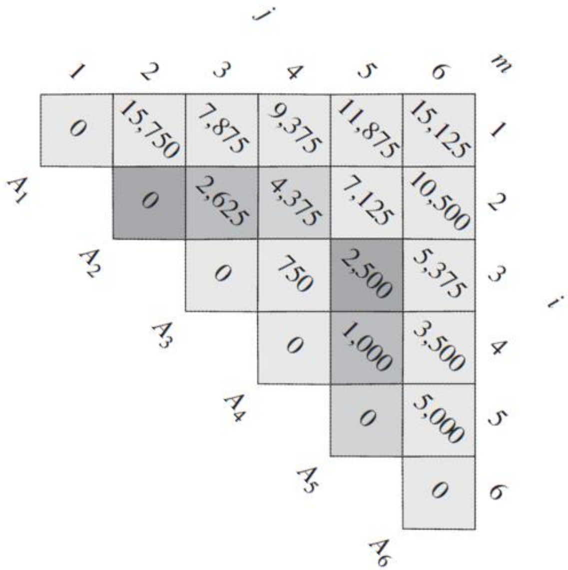
```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
      m[i, i] ← 0
for l←2 to n                    // l is the chain length.
      for i ← 1 to n - l + 1
            j ← i + l – 1
            m[i, j] ← ∞
            for k ← i to j - 1
                  q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                  if q < m[i, j]
                        m[i, j] ← q
                        s[i, j] ← k
return m and s
```

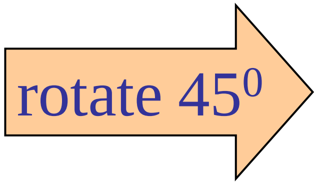
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

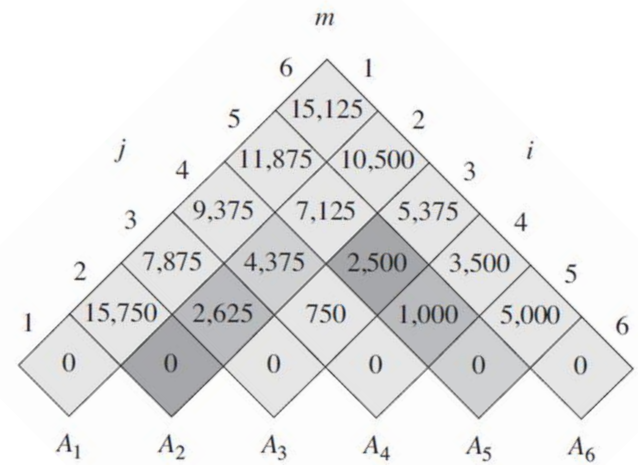


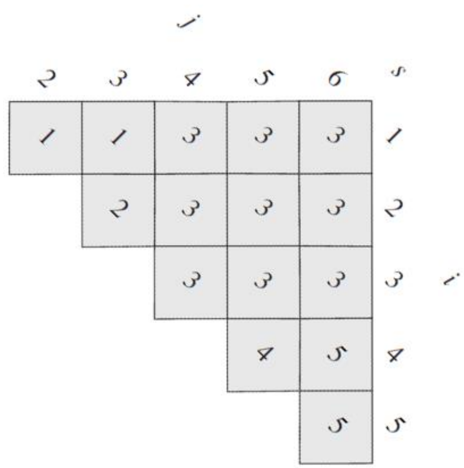


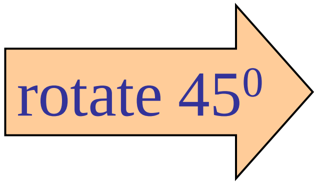
Step 3: Computing the optimal costs

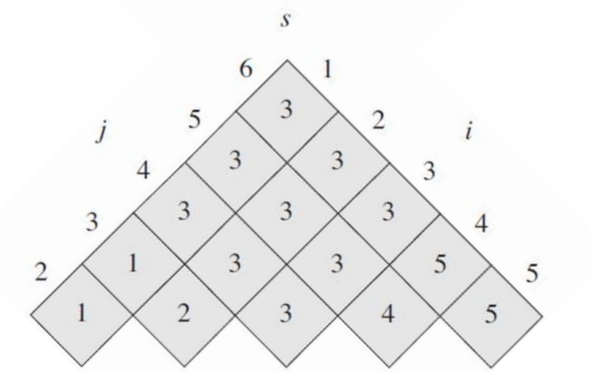






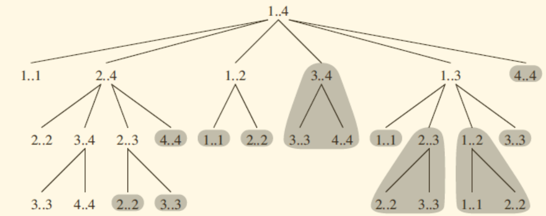






Step 3: Computing the optimal costs

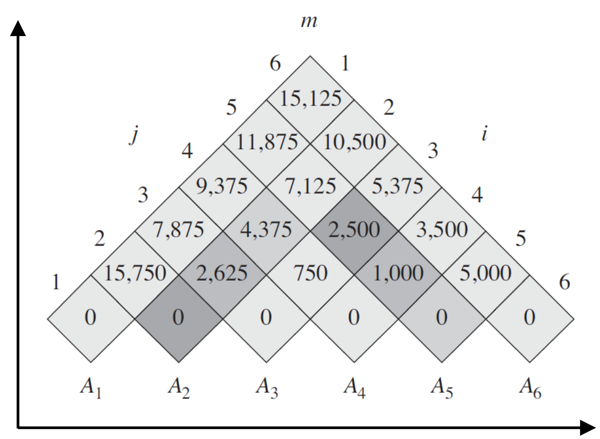
$$\Big(\underline{\big(A_i (A_{i+1}\ldots) (\ldots)\ldots A_k \big)} \big(A_{k+1}\ldots\ A_{j-1} A_j \big) \Big)$$

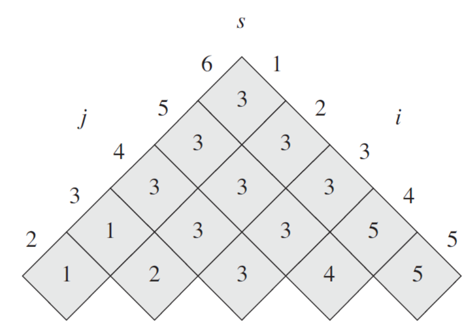


$$m[i,j] = \begin{cases} 0, & \text{if } i = j, \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
      m[i, i] ← 0
for l←2 to n          // l is the chain length.
      for i ← 1 to n - l + 1
            j ← i + l – 1
            m[i, j] ← ∞
            for k ← i to j - 1
                  q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                  if q < m[i, j]
                        m[i, j] ← q
                        s[i, j] ← k
return m and s
```

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





Step 3: Computing the optimal costs

```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
        m[i, i] ← 0
for l←2 to n          // l is the chain length.
        for i ← 1 to n - l + 1
                j ← i + l – 1
                m[i, j] ← ∞
                for k ← i to j - 1
                        q←m[i, k]+m[k+1, j]+p_{i-1} p_k p_j
                        if q < m[i, j]
                                m[i, j] ← q
                                s[i, j] ← k
return m and s
```

The running time?

Space requirement?

Time : $T(n) = O(n^3)$

Space: $S(n) = \Theta(n^2)$

Step 3: Computing the optimal costs

```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
      m[i, i] ← 0
for l←2 to n                        // l : n-1 times
      for i ← 1 to n - l + 1        // i : n-l+1 times
            j ← i + l – 1
            m[i, j] ← ∞
            for k ← i to j - 1      // k : j-i=l-1 times
                  q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
                  if q < m[i, j]
                        m[i, j] ← q
                        s[i, j] ← k
return m and s
```

Exercise:

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

The running time?

Step 3: Computing the optimal costs

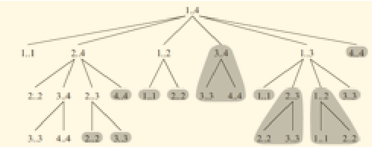
```
MCM-DP(p)
n ← length[p] – 1
for i ← 1 to n
    m[i, i] ← 0
for l←2 to n                         // l : n-1 times
    for i ← 1 to n - l + 1           // i : n-l+1 times
        j ← i + l – 1
        m[i, j] ← ∞
        for k ← i to j - 1           // k : j-i=l-1 times
            q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
            if q < m[i, j]
                m[i, j] ← q
                s[i, j] ← k
return m and s
```

```
int runCnt = 0;
void MCMdp(int n)
{
    int i,j,k,L;
    for(i=1; i<=n; i++)
        m[i][i] = 0;
    for(L=2; L<=n; L++)
    {
        for(i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = 1<<30;
            for(k=i; k<=j-1; k++)
            {
                int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if(q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
                runCnt++; // running times
            }
        }
    }
}
```

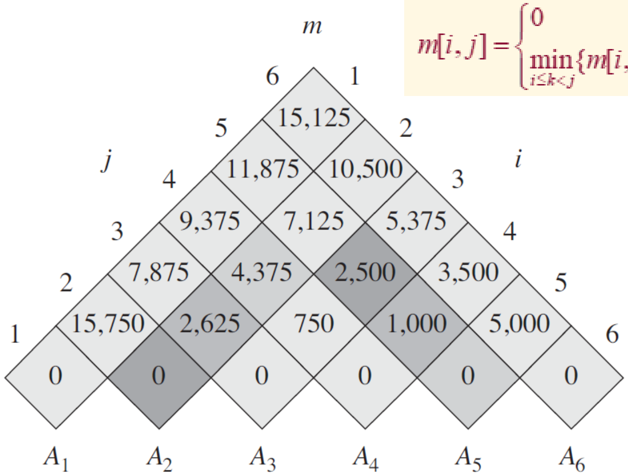
$$T(n) = \sum_{l=2}^{n} (n-l+1)(l-1) = \frac{(n-1)n(n+1)}{6} = \Theta(n^3)$$

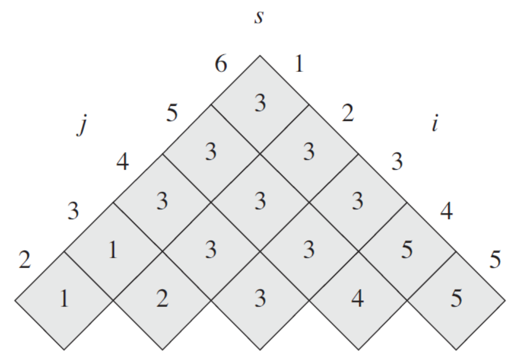
Step 4: Constructing an optimal solution

$$\left(\left(A_i(A_{i+1}\ldots)(\ldots)\ldots A_k\right)\left(A_{k+1}\ldots\ A_{j-1}A_j\right) \right)$$



$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}, & \text{if } i<j.\end{cases} \quad (15.12)$$



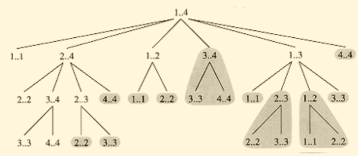


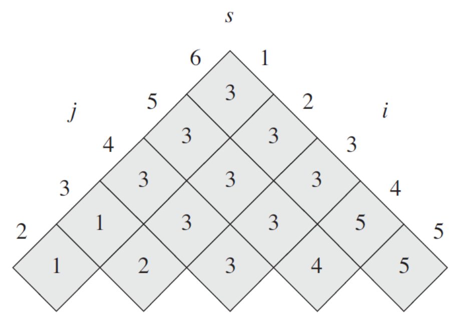
- MCM-DP **determines the optimal number *m*[*i*, *j*], but does not directly show how to multiply the matrices.**
  算法MCM-DP 给出了如何求解最佳乘法次数 *m*[*i*,*j*]，但对于按什么顺序来相乘各矩阵，没有给出具体方法
- **Constructing an optimal solution from table *s*[1.. *n*-1, 2.. *n*].**

Step 4: Constructing an optimal solution

$$(\; (A_i(A_{i+1}\cdots)\,(\ldots)\ldots A_k)\;(A_{k+1}\cdots\; A_{j-1}A_j) \;)$$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & \text{if } i < j. \end{cases} \qquad (15.12)$$





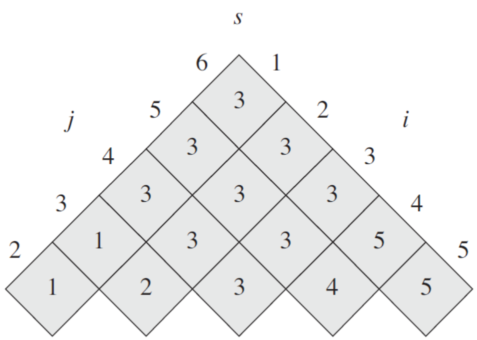
- **Each entry** $s[i, j]$ **records the value of** k **such that the optimal parenthesization of** $A_iA_{i+1}\cdots A_j$ **splits the product between** A_k **and** A_{k+1}**. Thus, the final matrix multiplication in computing** $A_{1..n}$ **optimally is** $A_{1..s[1,n]}A_{s[1,n]+1..n}$ **.**
  $s[i, j]$ 记录值 k，表示在矩阵连乘 $A_iA_{i+1}\cdots A_j$的最佳全括号中，分割点位于 A_k 和 A_{k+1} 之间。因此，矩阵连乘 $A_{1..n}$ 的最优分割方式为 $(A_1\, A_2 \ldots A_{s[1,n]})(A_{s[1,n]+1}\cdots A_n)$.
  - **The matrix multiplications can be computed recursively,**

    $s[1, s[1, n]] \rightarrow$ **splits** $A_{1..s[1,\, n]}$

    $s[s[1, n] + 1, n] \rightarrow$ **splits** $A_{s[1,\, n]+1..n}$
- **PRINT-OPTIMAL-PARENS**(s, i, j)

Step 4: Constructing an optimal solution

PRINT-OPTIMAL-PARENS(s, i, j) printing an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ recursively, given the s table. The initial call i=1, j=n .



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

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

How to work?

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

Step 4: Constructing an optimal solution

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

$s(1,6)$

$(\; s(1,3) \; s(4,6) \;)$

$(\; s(1,1) \; s(2,3) \;)$

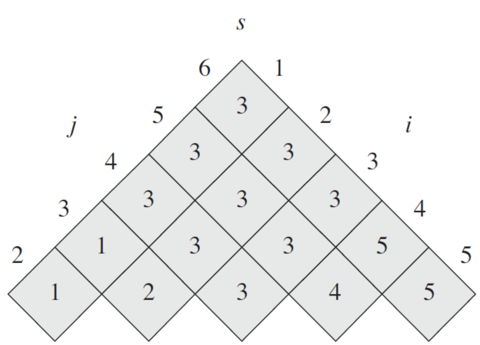
$(\; s(2,2) \; s(3,3) \;)$

$((A_1(A_2A_3))\ ((A_4A_5)A_6))$?

$A_1 A_2 A_3 A_4 A_5 A_6$: $((A_1 A_2 A_3)\ (A_4 A_5)\ A_6)$?

$A_1 (A_2 A_3)$? $(A_1 A_2) A_3$?

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



Exercise-1 (in class)

$A_1 A_2 A_3 A_4 A_5$: $(A_1 A_2 A_3)(A_4 A_5)$? $(A_1 A_2)(A_3 A_4 A_5)$? ……

$A_1 (A_2 A_3)$? $(A_1 A_2) A_3$?

$$(\underline{(A_i (A_{i+1} \ldots)(\ldots) \ldots A_k)} \; \underline{(A_{k+1} \ldots A_{j-1} A_j)})$$

- **Brute force: exhaustively checking all possible parenthesizations.**
- ***P*(*n*): the # of alternative parenthesizations of *n* matrices.**

  令$P(n)$表示 n 个矩阵连乘时所有可能的全括号方式的个数

- **What is the solution of *P*(*n*)?**

Solution of Exercise-1

Brute force: $P(n)$, the # of alternative parenthesizations of n matrices

$$P(n) = \begin{cases} 1 & , \quad \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{if } n \geq 2. \end{cases} \qquad (15.11)$$

substitution method: the solution to the recurrence (15.11) is $\Omega(2^n)$

Proof

we need $P(n) \geq c \cdot 2^n$

$P(k) \geq c \cdot 2^k \Rightarrow$

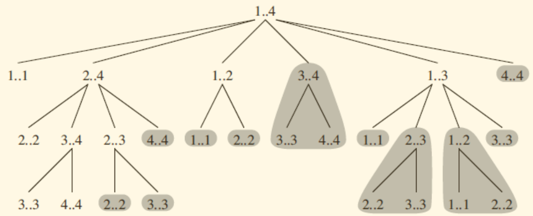
$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

$$\geq \sum_{k=1}^{n-1} c \cdot 2^k \cdot c \cdot 2^{n-k} = \sum_{k=1}^{n-1} c^2 \cdot 2^n = (n-1) \cdot c^2 \cdot 2^n$$

$$\geq c \cdot 2^n \quad (\text{ if } (n-1) \cdot c^2 \geq c, \text{ that is } n \geq \frac{1}{c} + 1)$$

Exercise-2 (in class)

$$\Big(\ \big(A_i\,(A_{i+1}\cdots)\,(\ldots)\ldots A_k\big)\,\big(A_{k+1}\cdots\ A_{j-1}\,A_j\big)\ \Big)$$



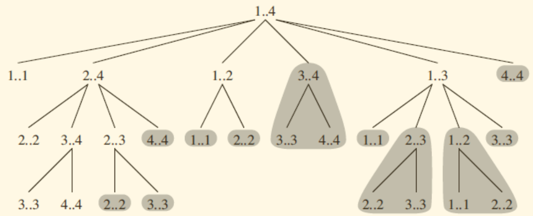
$$m[i,j] = \begin{cases} 0 & \text{if } i = j\,, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}\,, & \text{if } i < j\,. \end{cases} \qquad (15.12)$$

```
RE-MCM(p, i, j)
  if i = j
       return 0
  m[i, j] ← ∞
  for k←i  to  j-1
       q ← RE-MCM(p, i, k) + RE-MCM(p, k+1, j) + p_{i-1}p_k p_j
       if q < m[i, j]
            m[i, j] ← q
  return m[i, j]
```

Running time ?

Solution of Exercise-2

$$\Big(\; \underline{(A_i (A_{i+1}\ldots)(\ldots)\ldots A_k)} \; (A_{k+1}\ldots \; A_{j-1} A_j) \; \Big)$$



$$m[i,j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}, & \quad \text{if } i < j. \end{cases} \qquad (15.12)$$

RE-MCM(p, i, j)
if $i = j$
  return 0
$m[i, j] \leftarrow \infty$
for $k \leftarrow i$ **to** j-1
  $q \leftarrow$ RE-MCM(p, i, k) + RE-MCM(p, k+1, j) + $p_{i-1}p_k p_j$
  if $q < m[i, j]$
    $m[i, j] \leftarrow q$
return $m[i, j]$

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

$$= 2\sum_{i=1}^{n-1} T(i) + n \;\; \ge \;\; 3^n$$

Proof

let $T(i) \ge 3^i$,

then $T(n) \ge 2\sum_{i=1}^{n-1} T(i) + n$

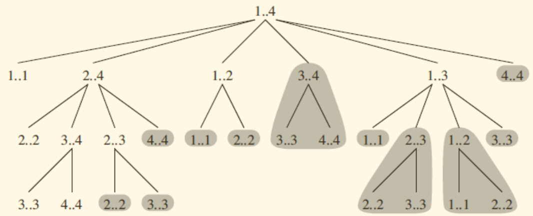
$$\ge \; 2 \times (3^1 + 3^2 + \cdots + 3^{n-1}) + n$$

$$\ge \; 2 \times 3 \frac{3^{n-1} - 1}{3 - 1} + n$$

$$= \; 3^n - 3 + n \ge \; 3^n$$

Exercise-3

$$\Big(\ \underline{(A_i (A_{i+1}\dots)(\dots)\dots A_k)}\ (A_{k+1}\dots\ A_{j-1} A_j)\ \Big)$$

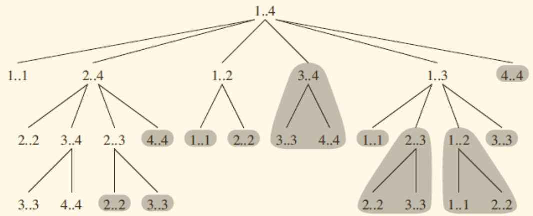


$$m[i,j]=\begin{cases}0 & \text{, if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\}, & \text{if } i<j.\end{cases} \qquad (15.12)$$

# of subproblems: one problem for each choice of *i* and *j* satisfying $1 \le i \le j \le n$?（所有子问题个数为 ?）

Exercise-3

$$(\ (A_i(A_{i+1}\ldots)\ (\ldots)\ldots A_k)\ (A_{k+1}\ldots\ A_{j-1}A_j)\ )$$



$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}, & \text{if } i<j.\end{cases} \quad (15.12)$$

# of subproblems: one problem for each choice of *i* and *j* satisfying $1 \le i \le j \le n$?（所有子问题个数为 ?）

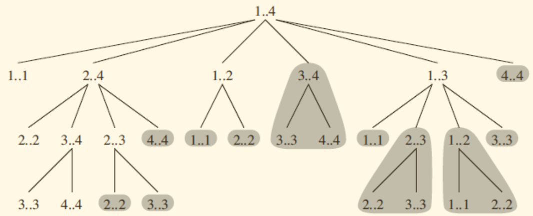
$$C_n^2+n=\Theta(n^2)$$

$$1\le i=j\le n,\ C_n^1=n$$

$$1\le i<j\le n,\ C_n^2=n(n-1)/2$$

Exercise-4

$$\color{red}{(}\ \underline{\color{green}{(\,A_i\,(A_{i+1}\ldots)\,(\ldots)\ldots A_k\,)}}\ \color{purple}{(\,A_{k+1}\ldots\ A_{j-1}\,A_j)}\ \color{red}{)}$$



$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

Dynamic Programming:

top-down with memoization?

15 Dynamic Programming

- ✓ **Scheduling two automobile assembly lines**
- ✓ **Steel rod cutting (15.1)**
- ✓ **Matrix-chain multiplication (15.2)**
- ✓ **Characteristics(Elements) of dynamic programming (15.3)**
- ✓ **Longest common subsequence (15.4)**
- ✓ **Optimal binary search trees (15.5)**

  最优二叉搜索树

15.5 Optimal binary search trees

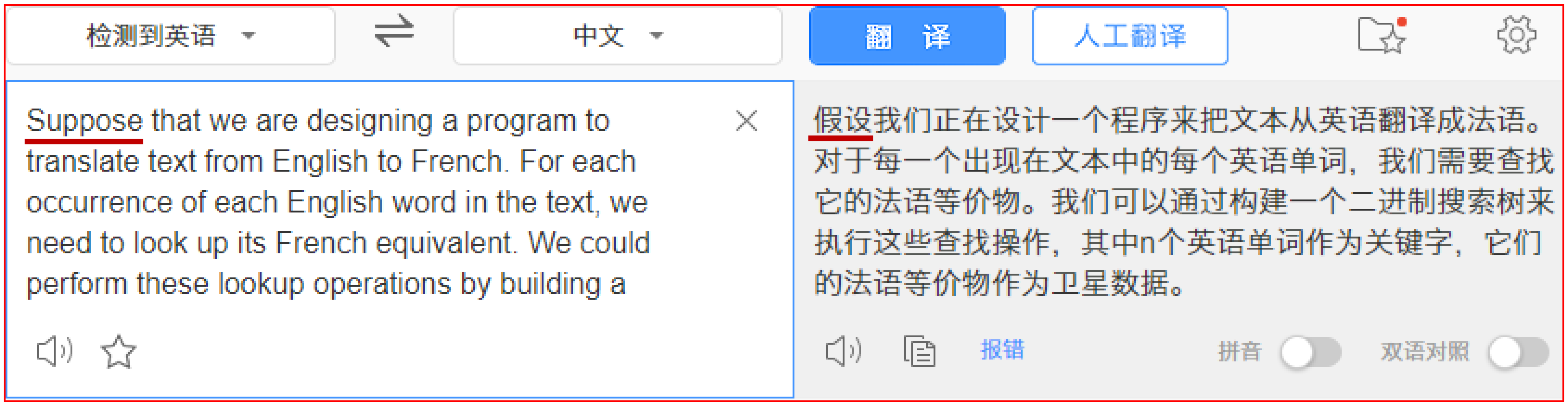
输入法的词库选择

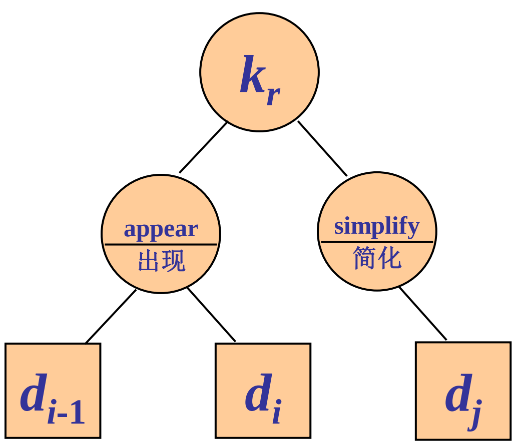
15.5 Optimal binary search trees

Design a program to translate text from English to Chinese （翻译软件）

15.5 Optimal binary search trees

Design a program to translate text from English to Chinese





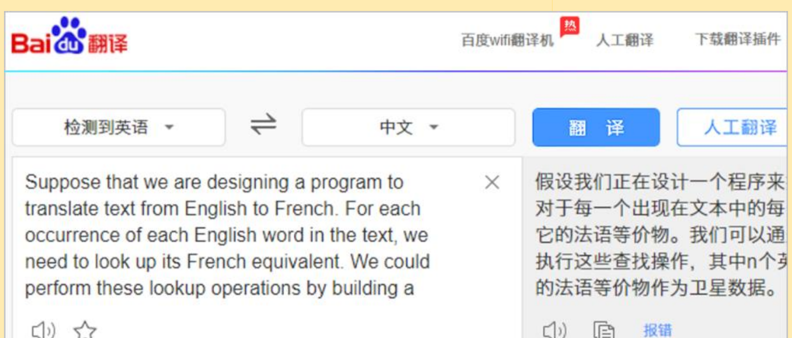
生词表	
字段1	字段2
aggregate	综合，总体
amortized	分摊，平摊
arbitrary	任意的，武断的
auxiliary	辅助的
binomial	二项的，二项式的
bog	沼泽，陷于泥沼
…	…
suppose	假设
…	…

15.5 Optimal binary search trees

- **Design a program to translate text from English to Chinese**
- **an $O(n)$ search time per occurrence by using any linear table operation**

生词表	
字段1	字段2
aggregate	综合，总体
amortized	分摊，平摊
arbitrary	任意的，武断的
auxiliary	辅助的
binomial	二项的，二项式的
bog	沼泽，陷于泥沼
…	…
suppose	假设
…	…

15.5 Optimal binary search trees

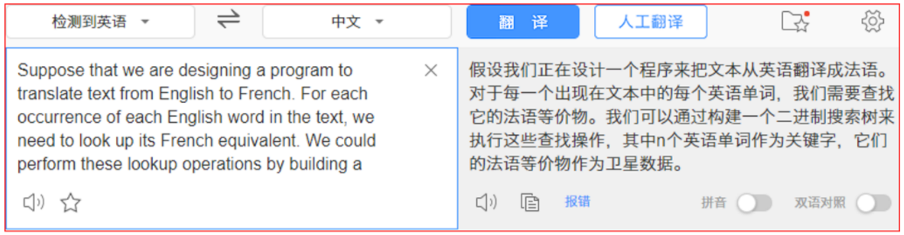


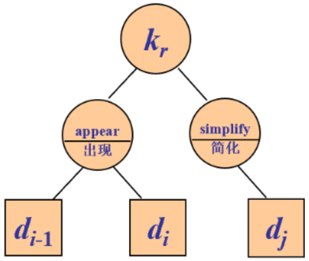
算法是软件的灵魂

算法是企业生产力

15.5 Optimal binary search trees

- **lookup operations:**
  build a binary search tress (BST) with
  - ***n* English words as keys**
  - **Chinses equivalents as satellite data 从属数据**

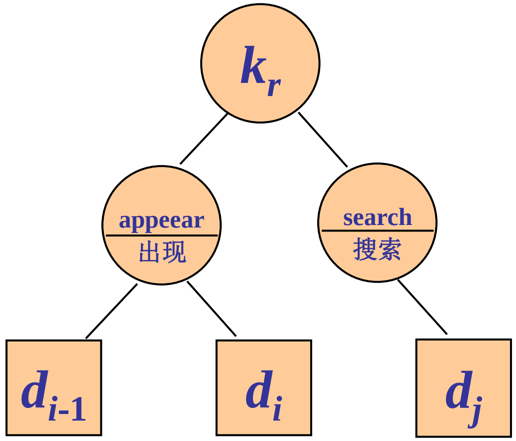




生词表	
字段1	字段2
aggregate	综合，总体
amortized	分摊，平摊
arbitrary	任意的，武断的
auxiliary	辅助的
binomial	二项的，二项式的
bog	沼泽，陷于泥沼
...	...
suppose	假设
...	...

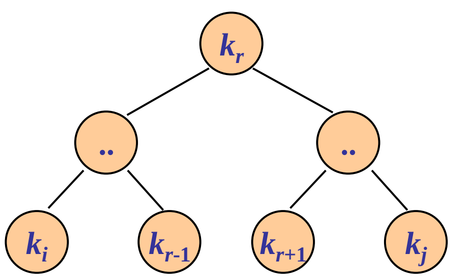
- **Because we will search the tree for each individual word in the text, we want the total time spent searching to be as low as possible.** 对于课文中出现的每个单词，都需要搜索该二叉树，如何使得总的搜索次数最少？
- **an $O(\lg n)$ search time per occurrence by using any balanced BST.** 对于任何一个单词的搜索，使用二分搜索法的时间为$O(\lg n)$.

15.5 Optimal binary search trees



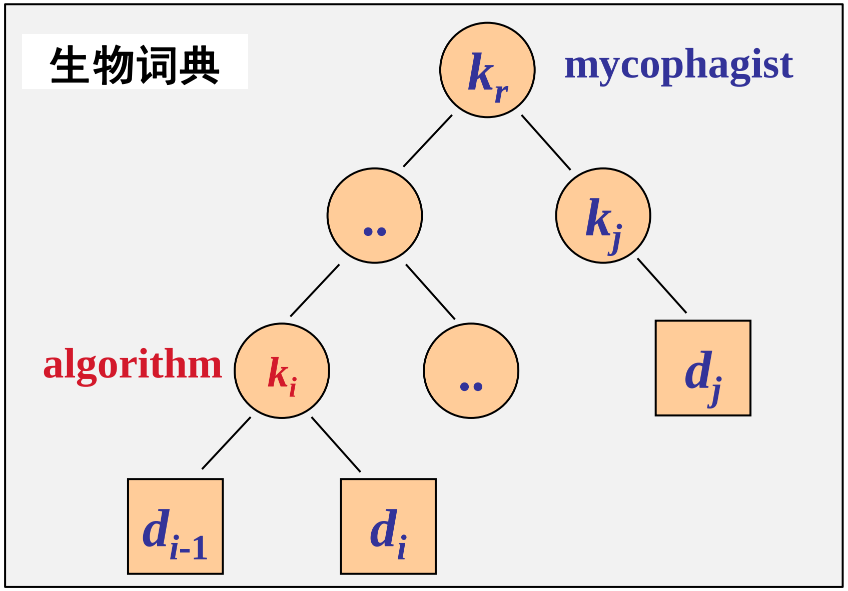
生词表	
字段1	字段2
aggregate	综合，总体
amortized	分摊，平摊
arbitrary	任意的，武断的
auxiliary	辅助的
binomial	二项的，二项式的
bog	沼泽，陷于泥沼
…	…

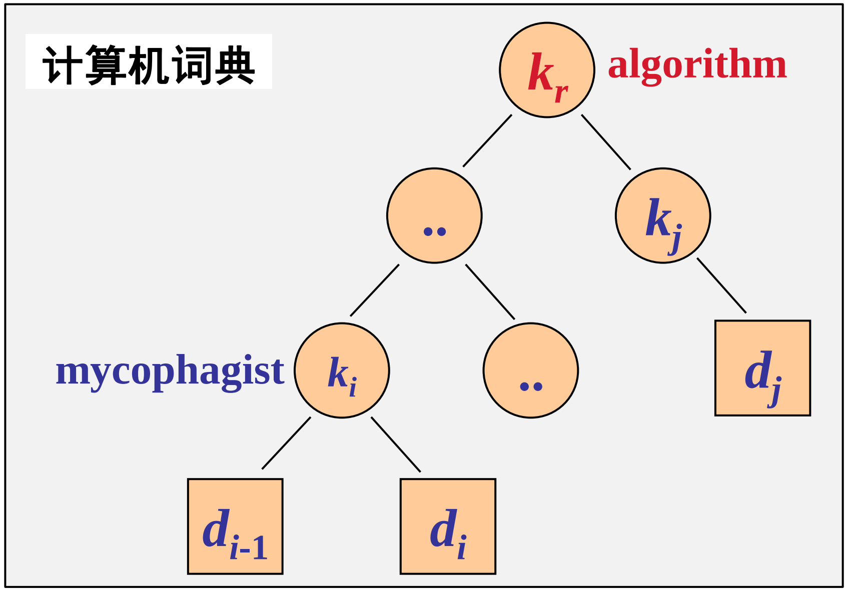
A balanced BST…



However, Words appear with different frequencies…?

15.5 Optimal binary search trees

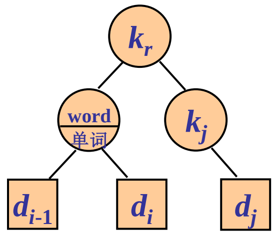




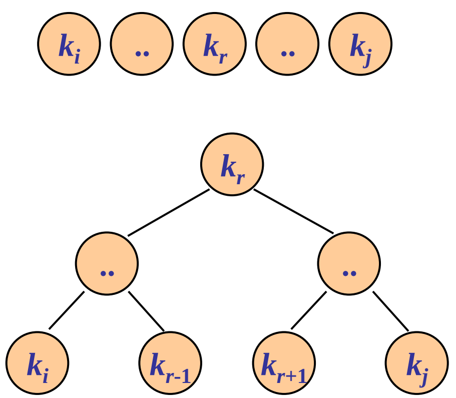
- **Words appear with different frequencies**
- **It may be: "algorithm" (frequently used) appears far from the root; "mycophagist" (rarely used, 食菌者) appears near the root**
- **Such an organization would slow down the translation, since # of nodes visited when searching for a key in a BST is 1 + depth**
- **We want words that occur frequently in the text to be placed nearer the root**

15.5 Optimal binary search trees

- Need the total time spent searching to be as low as possible.
- We want words that occur frequently in the text to be placed nearer the root.
- Moreover, there may be words in the text for which there is no Chinese translation, and such words might not appear in the BST at all.
  文中有些英语单词没有对应的汉语译文，即这些英语单词不出现在二叉搜索树"词典"中



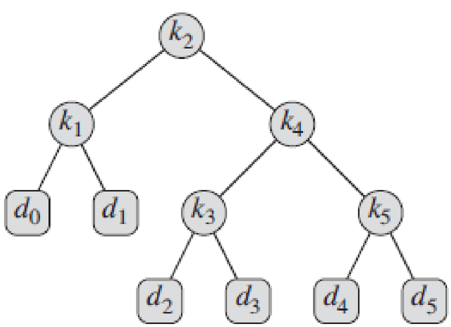
- How do we organize a BST so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?
  设已知每个单词出现的概率，如何组织一颗二叉搜索树，使得在所有搜索中，被访问的节点的总数最少？

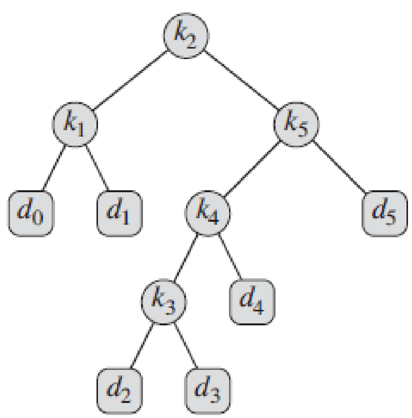


15.5 Optimal binary search trees

BST: Given a sequence $K = \langle k_1, k_2, ..., k_n \rangle$ of n distinct keys in sorted order ($k_1 < k_2 < \cdots < k_n$), how to build a BST?

- For key k_i, search probability p_i (it can also be the number I of occurrence of k_i in the text, whose number of total words is M, then $p_i = I / M$.)
- Some values not in K, n+1 "dummy keys" $d_0, d_1, ..., d_n$
- d_0 represents all values $< k_1$; d_n represents all values $> k_n$
- for $1 \le i \le n$-1, d_i : $k_i < d_i < k_{i+1}$, search probability q_i
- Fig, each key k_i, an internal node; d_i, a leaf





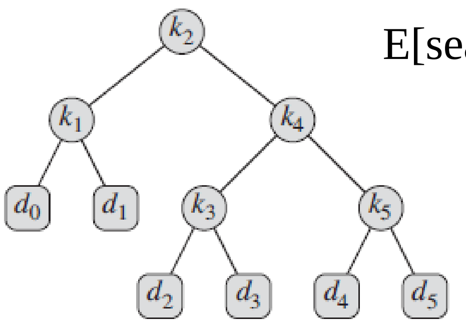
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

15.5 Optimal binary search trees

- **Every search is either successful or unsuccessful, we have**

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \qquad (15.15)$$

- **We have probabilities of searches for each (dummy) key, we can determine the expected cost of a search in a given BST *T*.**
- **Assume that the actual cost of a search is the number of nodes examined. Then the expected cost of a search in *T* is**



$$\begin{aligned}\text{E[search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i)\cdot q_i \ , \qquad (15.16)\end{aligned}$$

where depth_T denotes a node's depth in the tree *T*.（树根高度为0）

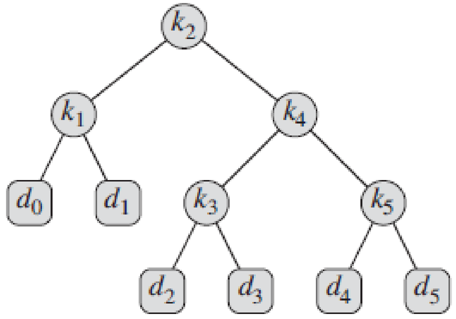
15.5 Optimal binary search trees

$$\text{E}[\text{search cost in } T]$$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

In Figure 15.7(a), we can calculate the expected search cost node by node:



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			**2.80**

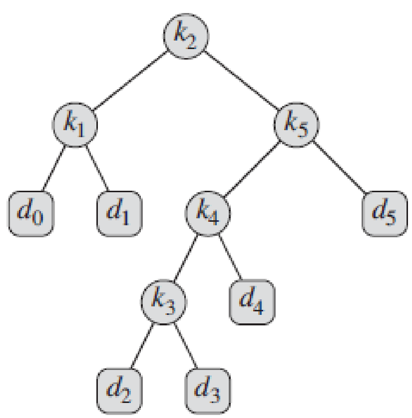
15.5 Optimal binary search trees

$$\text{E[search cost in } T]$$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

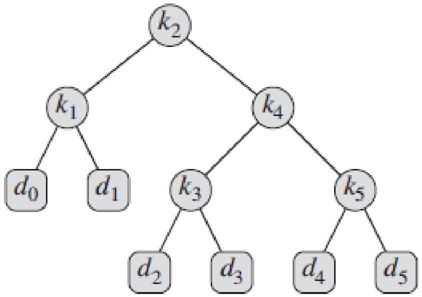
In Figure 15.7(b)

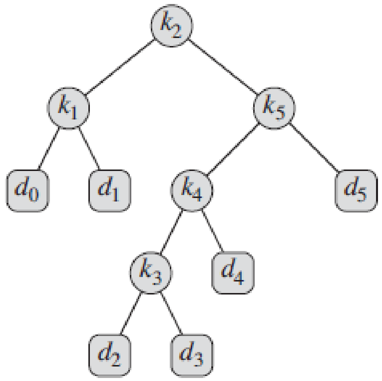


node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

15.5 Optimal binary search trees

- **Optimal BST** : for a given set of probabilities, our goal is to construct a BST whose expected search cost is the **smallest**.





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

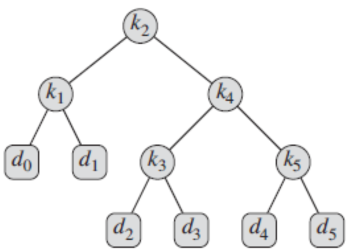
- **How to build an OBST?**

  Intuitively,

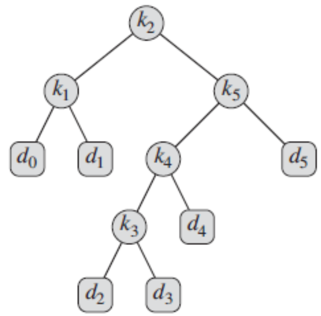
  - **the overall height is smallest**
  - **the key with the greatest probability at the root**

15.5 Optimal binary search trees

- **Optimal BST : for a given set of probabilities, our goal is to construct a BST whose expected search cost is the smallest.**



(a) cost: 2.80



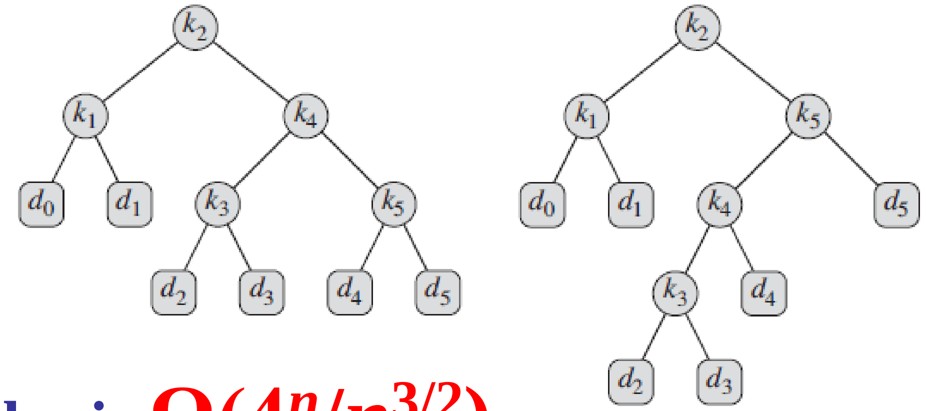
(b) cost: 2.75

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

- **Figure 15.7(b) shows an Optimal BST's expected cost is 2.75**
  - **An Optimal BST is not necessarily a tree whose overall height is smallest.**
    不一定要求树的高度最小
  - **Nor can we necessarily construct an Optimal BST by always putting the key with the greatest probability at the root. (The lowest expected cost of any BST with k_5 (the greatest probability) at the root is 2.85.)**
    不一定将概率最大的 key 放在树根，如…

15.5 Optimal binary search trees

- **Exhaustive checking of all possibilities fails to yield an efficient algorithm.**
  - **ALS, RodCut, MCM**

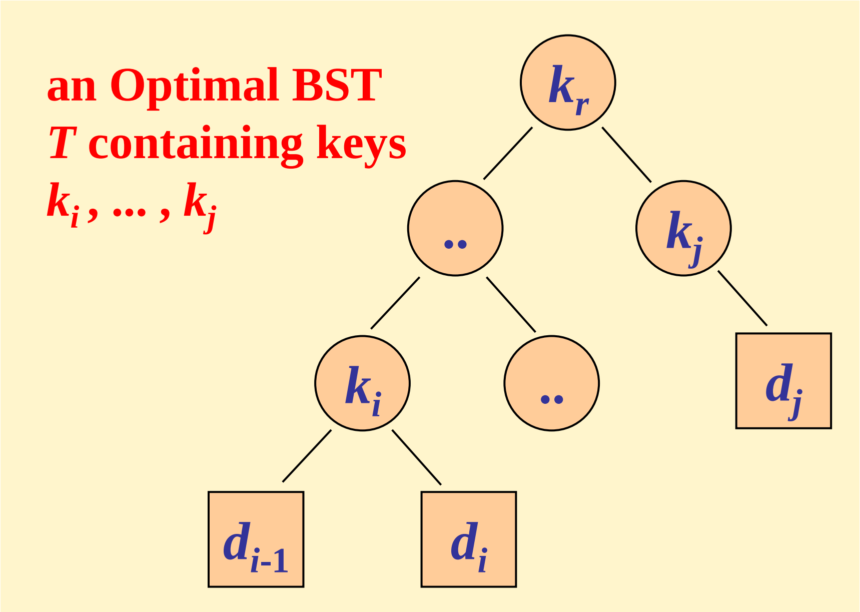


- **The # of BST with *n* nodes is $\Omega(4^n/n^{3/2})$ （Problem 12-4）.**

- **Not surprisingly, we will solve this problem with dynamic programming.**

Step 1: The structure of an Optimal BST

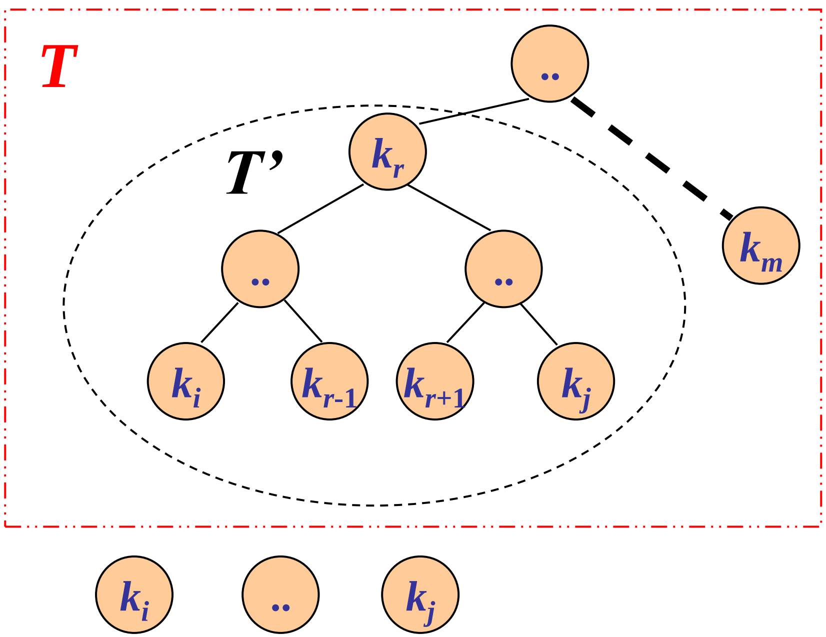
- **Start with an observation about subtrees.**
- **Consider any subtree of a BST**
  - **It must contain keys in a contiguous range k_i , ..., k_j , for some $1 \le i \le j \le n$.**
  - **In addition, the subtree must also have as its leaves the dummy keys d_{i-1} , ..., d_j .**
- **Optimal substructure?**



Step 1: The structure of an Optimal BST

Optimal substructure: If an Optimal BST T has a subtree T' containing keys k_i, ..., k_j , then this subtree T' must be optimal as well for the subproblem with keys k_i , ..., k_j and dummy keys d_{i-1} , ..., d_j .

设 T' 为最优BST T 的一个子树， T' 包含keys k_i, ..., k_j , 那么T' 是子问题〔关于keys k_i, ..., k_j 和dummy keys d_{i-1} , ..., d_j〕的最优BST



T : search tree of k_i , ... , k_m

T' : search tree of k_i , ... , k_j

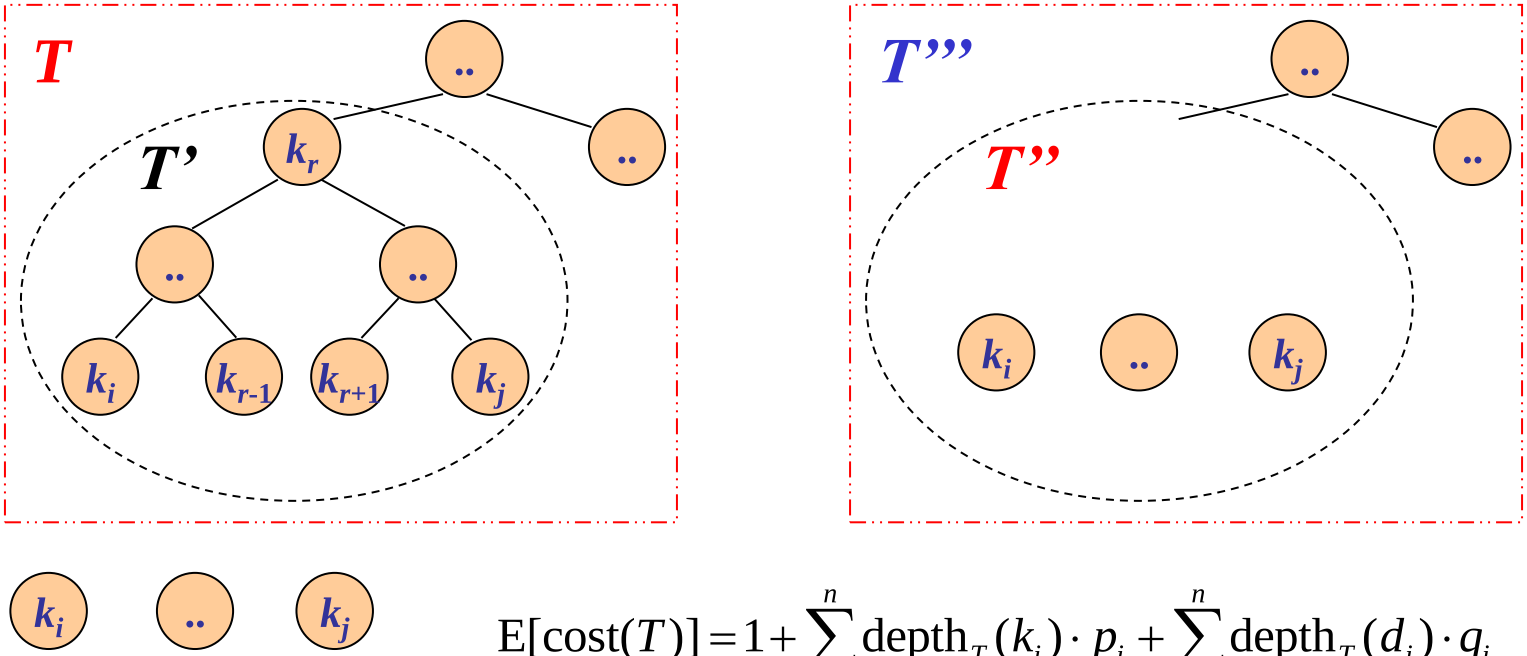
Optimal BST T

➔ Optimal BST T'

Step 1: The structure of an Optimal BST

Idea of Proof: Cut-and-paste argument applies.

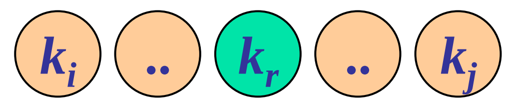
If there were a subtree *T"* whose expected cost is lower than that of *T'*, then we could cut *T'* out of *T* and paste in *T"*, resulting in a binary search tree of lower expected cost than *T*, thus contradicting the optimality of *T*.

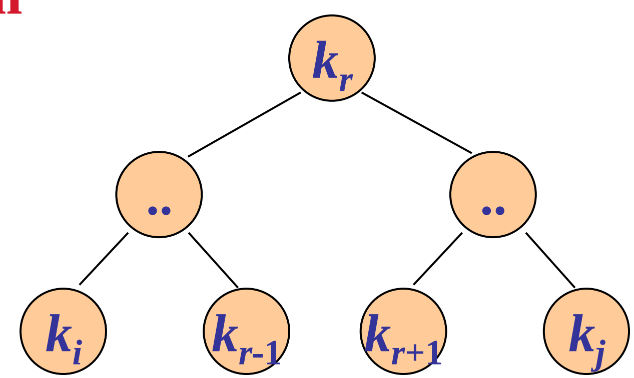


$$\mathrm{E}[\mathrm{cost}(T)] = 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i)\cdot q_i$$

Step 1: The structure of an Optimal BST

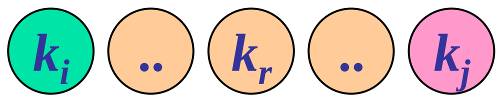
- **Using the optimal substructure, we can construct an optimal solution to the problem from optimal solutions to subproblems.**
- **Given keys k_i, ..., k_j, one of these keys, say k_r ($i \le r \le j$), will be the root of an optimal subtree.**
  - **The left subtree of the root k_r will contain the keys k_i, ..., k_{r-1} (and dummy keys d_{i-1}, ..., d_{r-1}); the right subtree will contain the keys k_{r+1}, ..., k_j (and dummy keys d_r, ..., d_j).**
- **As long as we examine all candidate roots k_r, where $i \le r \le j$, and we determine all optimal BST containing k_i, ..., k_{r-1} and those containing k_{r+1}, ..., k_j, we will find an Optimal BST.**

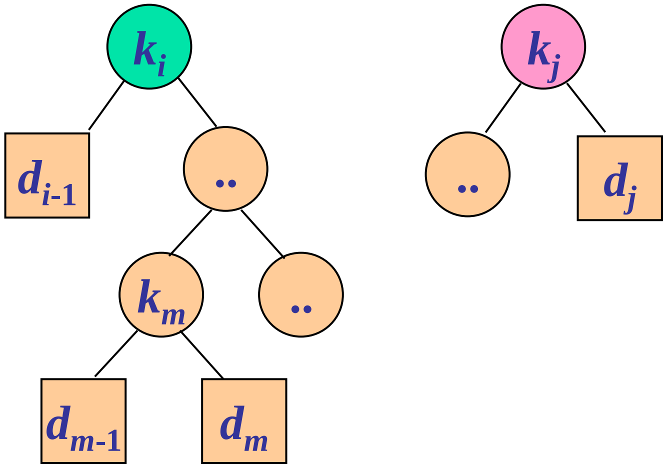




Step 1: The structure of an Optimal BST

- **A detail, "empty" subtrees**
- **Suppose that in a subtree with keys k_i , ..., k_j ,**
  - **We select k_i as the root, left subtree of k_i contains no keys. Bear in mind, however, that subtrees also contain dummy keys $d_{i\text{-}1}$.**
  - **Symmetrically, if we select k_j as the root, right subtree of k_j contains the keys k_{j+1} , ..., k_j ; this right subtree contains no actual keys, but it does contain the dummy key d_j .**

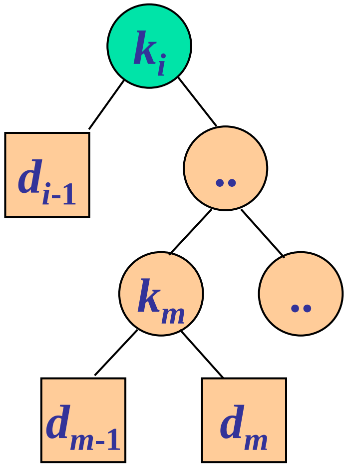


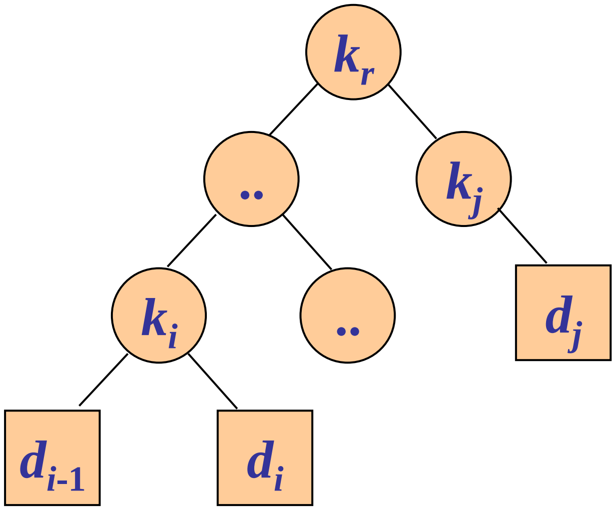


Step 2: A recursive solution

Subproblem : finding an Optimal BST containing the keys k_i , ..., k_j , where $i \geq 1$, $j \leq n$, and $j \geq i$-1. (when $j = i$-1, there are no actual keys, we have just the dummy key $d_{i\text{-}1}$.)

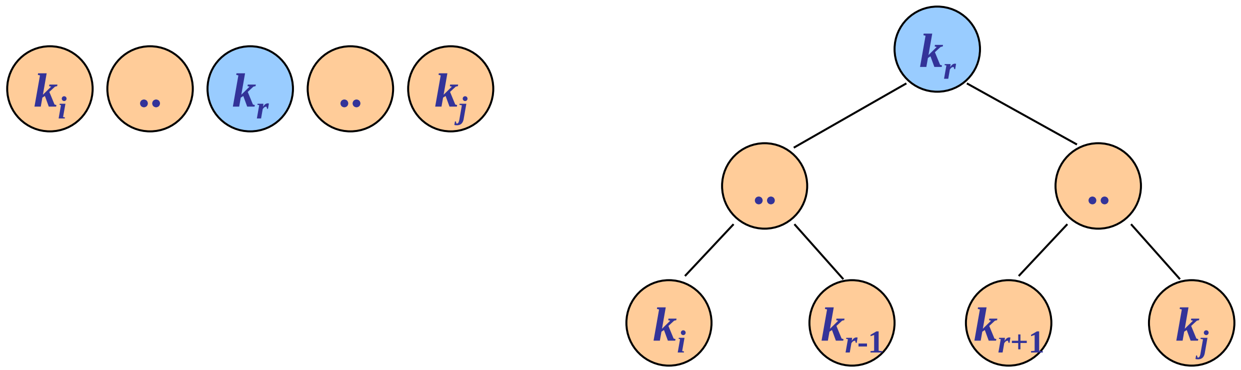
- **$e[i, j]$: the expected cost of searching an Optimal BST containing the keys k_i , ..., k_j .**
- Ultimately, wish to compute $e[1, n]$.
- when $j = i$-1, only $d_{i\text{-}1}$, $e[i, i\text{-}1] = q_{i\text{-}1}$.
- When $j \geq i$?





Step 2: A recursive solution

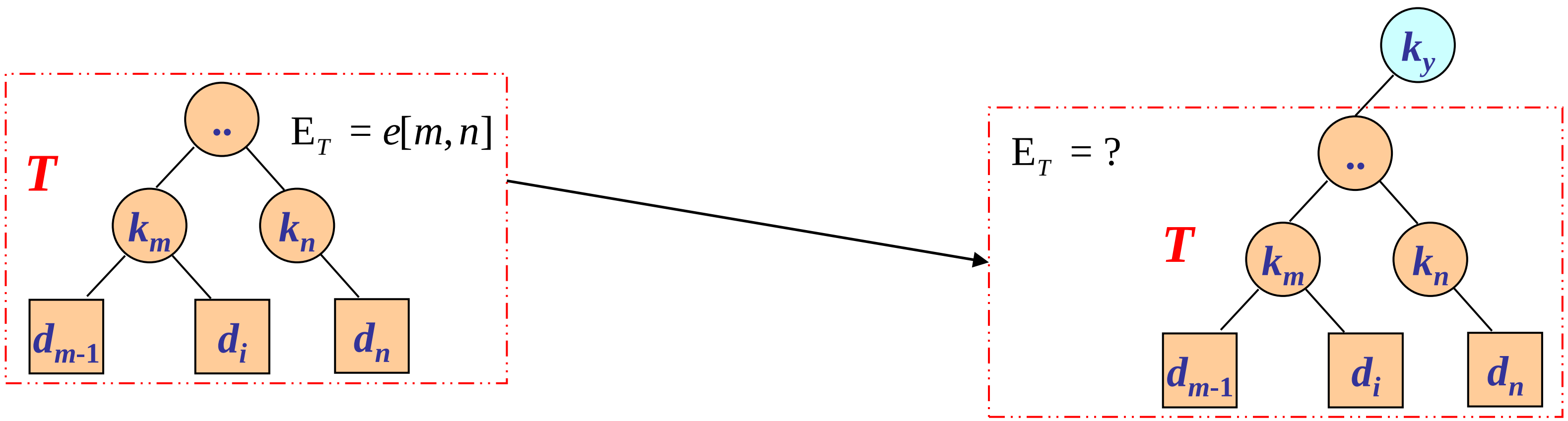
When $j \geq i$, select a root k_r from among k_i , ..., k_j , and then make an Optimal BST with keys k_i , ..., k_{r-1} its left subtree and an Optimal BST with keys k_{r+1} , ..., k_j its right subtree.



Step 2: A recursive solution

What happens to the expected search cost of a subtree *T* when it becomes a subtree of a node?

- **The depth of each node in the subtree increases by 1, the expected search cost of this subtree increases by the sum of all the probabilities in the subtree.**



the sum of all the probabilities:

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \qquad (15.17)$$

$$E_T = \sum_{x=m}^{n} (\text{depth}(k_x)+1+1)\cdot p_i + \sum_{x=m\text{-}1}^{n} (\text{depth}(d_x)+1+1)\cdot q_x$$

$$= \sum_{x=m}^{n} (\text{depth}(k_x)+1)\cdot p_i + \sum_{x=m-1}^{n} (\text{depth}(d_x)+1)\cdot q_x + \sum_{x=m}^{n} p_i + \sum_{x=m-1}^{n} q_x$$

$$= e[m,n] + w[m,n]$$

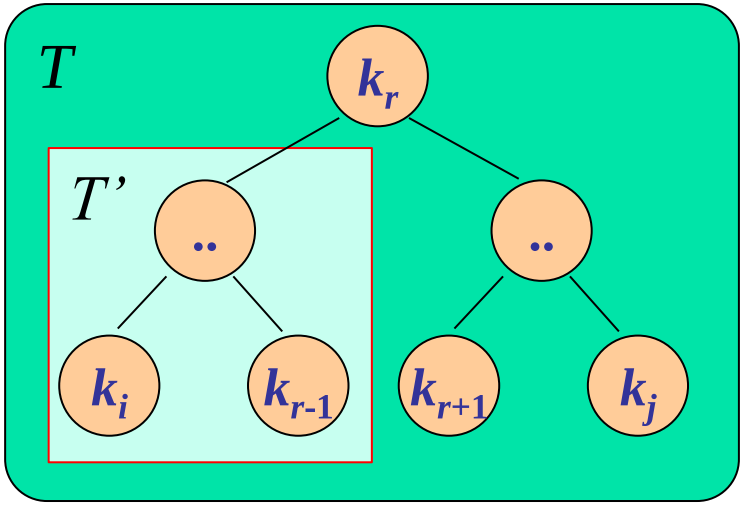
增量为 $w[m, n]$

Step 2: A recursive solution

OBST *T* 与 OBS-subTree *T'* 的关系:

if k_r is the root of an OBST containing keys k_i , ... , k_j , we have

$$\boxed{e[i, j\,]} = p_r + (\boxed{e[i, r\text{-}1]} + w[i, r\text{-}1]) + (e[r+1, j] + w[r+1, j])\ ?$$



Noting that $w[i, j] = w[i, r\text{-}1] + p_r + w[r+1, j]$

$$\left(w[i, r-1] = \sum_{l=i}^{r-1} p_l + \sum_{l=i-1}^{r-1} q_l \quad , \quad w[r+1, j] = \sum_{l=r+1}^{j} p_l + \sum_{l=r}^{j} q_l \right)$$

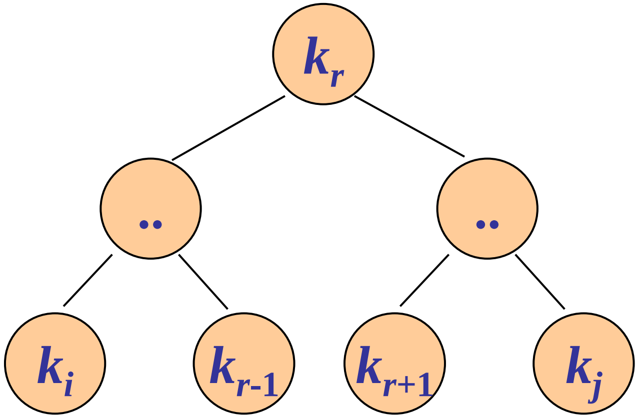
We rewrite $e[i, j]$ as

$$\mathbf{e[i, j\,] = e[i, r\text{-}1] + e[r+1, j] + w[i, j]} \qquad \mathbf{(15.18)}$$

The recursive equation (15.18) assumes that we know which node k_r to use as the root, which we do not know.

Step 2: A recursive solution

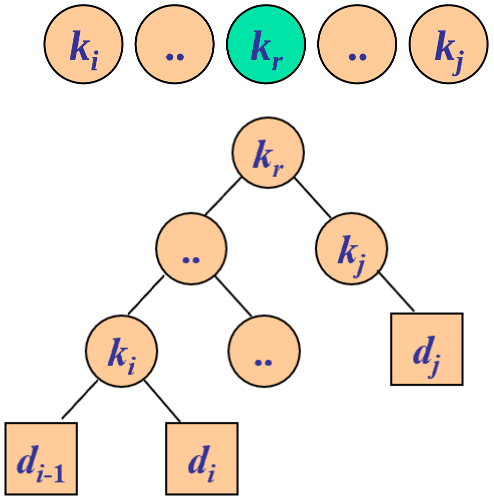
- **Choose k_r as the root that gives the lowest expected search cost, giving us the final recursive formulation of an OBST cost $e[i, j]$:**



$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

- **To help us keep track of the structure of Optimal BST, define $root[i, j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an Optimal BST containing keys k_i , ..., k_j .**

Step 3: Computing the expected search cost



$$A_i \ldots A_k A_{k+1} \ldots A_j$$

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

- Similarity: OBST and matrix-chain multiplication.
- A direct, recursive implementation would be as inefficient ?
- Store the $e[i, j]$ values in a table $e[1..\ n+1,\ 0..\ n]$.
  - The first index runs to $n+1$, in order to have a subtree containing only d_n , need to compute and store $e[n+1, n]$.
  - The second index starts from 0, in order to have a subtree containing only d_0 , need to compute and store $e[1, 0]$.
- *root*$[i, j]$, recording the root of the subtree containing keys k_i , ..., k_j .

Step 3: Computing the expected search cost

- Other table for efficiency.

$$e[i, j] = e[i, r\text{-}1] + e[r+1, j] + w[i, j] \qquad (15.18)$$

- Rather than compute the value of $w[i, j]$ every time we are computing $e[i, j]$—which would take $\Theta(j\text{-}i)$ additions—we store these values in a table $w[1..\ n+1, 0..\ n]$.

（无需每次计算 $e[i, j]$ 时都计算 $w[i, j]$，…）

- For the base case, we compute $w[i, i\text{-}1] = q_{i\text{-}1}$ for $1 \le i \le n$.
- For $j \ge i$,

$$w[i, j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i, j-1] + p_j + q_j \qquad (15.20)$$

- Thus, compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.
- Inputs: the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n

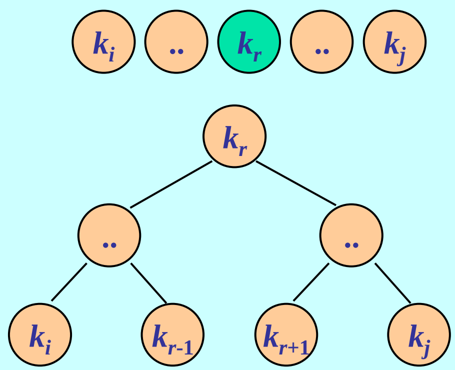
Step 3: Computing the expected search cost

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i,j-1] + p_j + q_j \qquad (15.20)$$

OBST(p, q, n)

```
for i←1 to n+1
    e[i, i-1]←q_{i-1}
    w[i, i-1]←q_{i-1}
for l←1 to n
    for i←1 to n-l+1
        j←i+l-1
        e[i, j]←∞
        w[i, j]←w[i, j-1]+p_j+q_j
        for r←i to j
            t←e[i, r-1]+e[r+1, j]+w[i, j]
            if t<e[i, j]
                e[i, j]←t
                root[i, j]←r
return e and root
```



VS

$$(\ (A_i (A_{i+1} \ldots)(\ldots)\ldots A_k)\ (A_{k+1} \ldots\ A_{j-1} A_j)\ )$$

$$m[i,j] = \begin{cases} 0 & , \quad \text{if } i = j, \\ \min\limits_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}, & \text{if } i < j. \end{cases} \qquad (15.12)$$

MCM-DP(p)

```
n ← length[p] – 1
for i ← 1 to n
    m[i, i] ← 0
for l←2 to n              // l is the chain length.
    for i ← 1 to n - l + 1
        j ← i + l – 1
        m[i, j] ← ∞
        for k ← i to j - 1
            q←m[i, k]+m[k+1, j]+p_{i-1}p_k p_j
            if q < m[i, j]
                m[i, j] ← q
                s[i, j] ← k
return m and s
```

Step 3: Computing the expected search cost

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i,j-1] + p_j + q_j \qquad (15.20)$$

```
OBST(p, q, n)
for i←1 to n+1
    e[i, i-1]←q_{i-1}
    w[i, i-1]←q_{i-1}
for l←1 to n
    for i←1 to n-l+1  // ?1
        j←i+l-1       // ?2
        e[i, j]←∞
        w[i, j]←w[i, j-1]+p_j+q_j
        for r←i to j
            t←e[i, r-1]+e[r+1, j]+w[i, j]
            if t<e[i, j]
                e[i, j]←t
                root[i, j]←r
return e and root
```

?1

***e*[*i*, *j*]:**

l **个元素的 Opti-BST 的 cost**

$i = 1,\ j = l,$

$i = 2,\ j = l+1,$

…

$i = x,\ j = n,$

$n - x + 1 = l \Rightarrow x = n - l + 1$

?2

$j - i + 1 = l \implies j = i + l - 1$

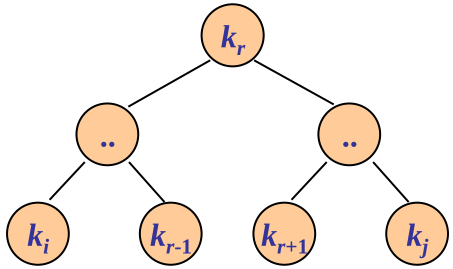
Step 3: Computing the expected search cost

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

$$w(i,j)=\sum_{l=i}^{j}p_l+\sum_{l=i-1}^{j}q_l = w(i,j-1)+p_j+q_j$$

```
OBST(p, q, n)
1  for i←1 to n+1
2      e[i, i-1]←q_{i-1}
3      w[i, i-1]←q_{i-1}
4  for l←1 to n          // 求l 个元素的Opti-BST
5      for i←1 to n-l+1
6          j←i+l-1
7          e[i, j]←∞
8          w[i, j]←w[i, j-1]+p_j+q_j
9          for r←i to j
10             t←e[i, r-1]+e[r+1, j]+w[i, j]
11             if t<e[i, j]
12                 e[i, j]←t
13                 root[i, j]←r
14 return e and root
```

Innermost for loop, in lines 9–13, tries each candidate index r to determine which key k_r to use as the root of an OBST containing keys k_i , ..., k_j . 对包含 k_i , ..., k_j 的最优 BST，遍历每一个 k_r 作为树根，…

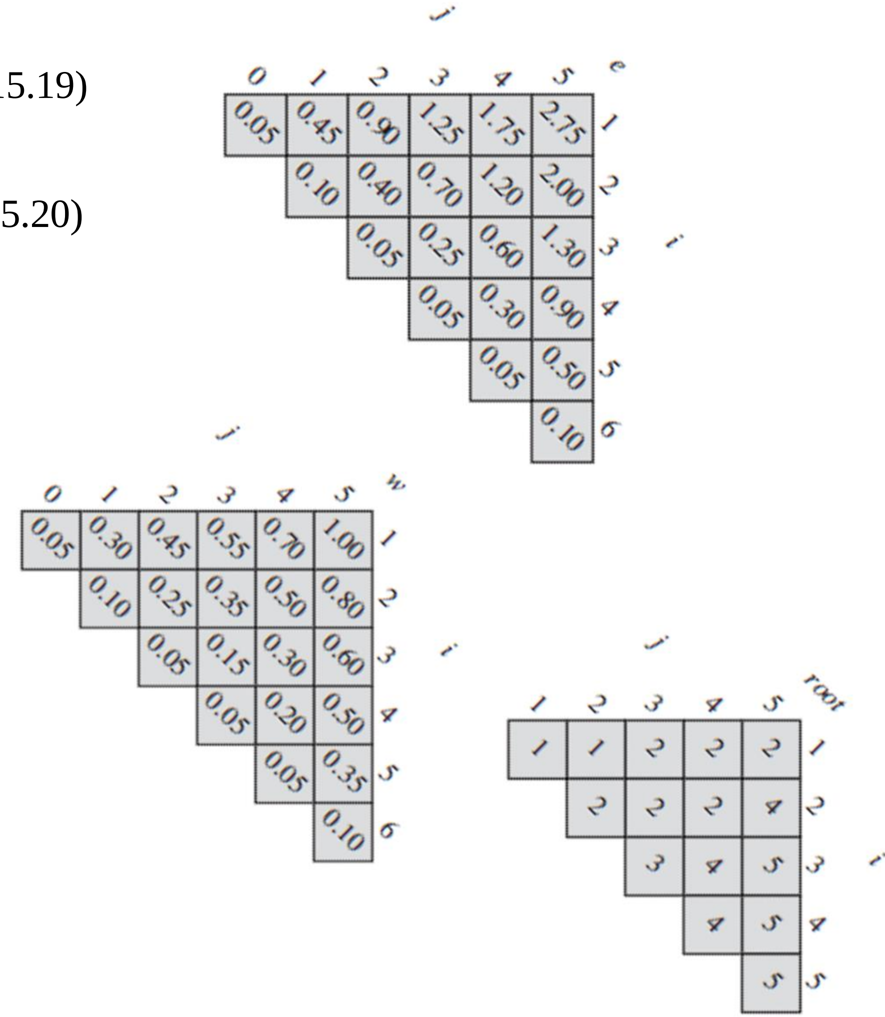


Step 3: Computing the expected search cost

$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1, \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\} & \text{if } i\le j. \end{cases} \qquad (15.19)$$

$$w[i,j]=\sum_{l=i}^{j} p_l+\sum_{l=i-1}^{j} q_l = w[i,j-1]+p_j+q_j \qquad (15.20)$$

```
OBST(p, q, n)
for i←1 to n+1
    e[i, i-1]←q_{i-1}
    w[i, i-1]←q_{i-1}
for l←1 to n        // 求l 个元素的Opti-BST
    for i←1 to n-l+1
        j←i+l-1
        e[i, j]←∞
        w[i, j]←w[i, j-1]+p_j+q_j
        for r←i to j
            t←e[i, r-1]+e[r+1, j]+w[i, j]
            if t<e[i, j]
                e[i, j]←t
                root[i, j]←r
return e and root
```

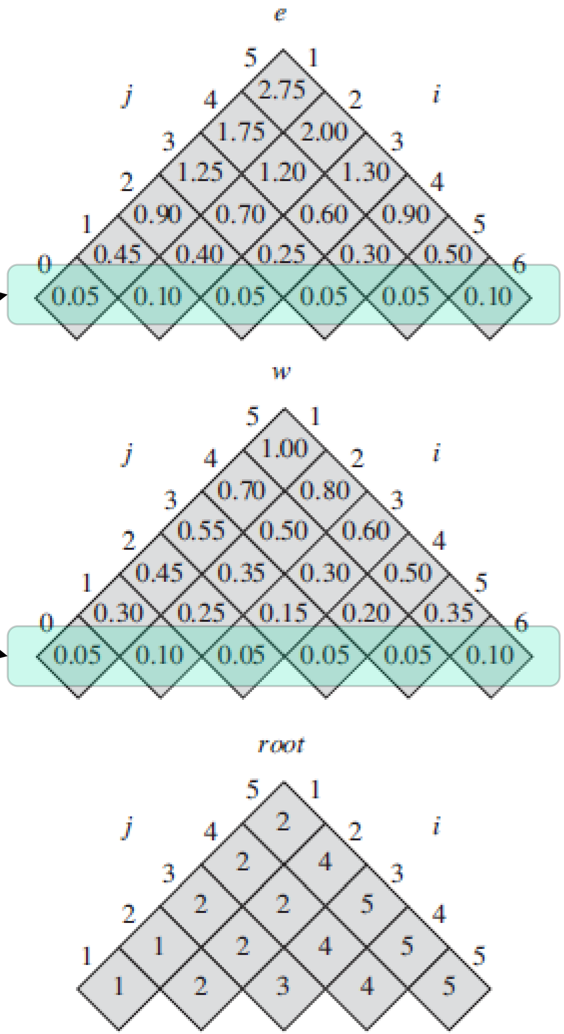


Step 3: Computing the expected search cost

$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1, \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\} & \text{if } i\le j. \end{cases} \qquad (15.19)$$

$$w[i,j]=\sum_{l=i}^{j} p_l+\sum_{l=i-1}^{j} q_l = w[i,j-1]+p_j+q_j \qquad (15.20)$$

```
OBST(p, q, n)
for i←1 to n+1
      e[i, i-1]←q_{i-1}
      w[i, i-1]←q_{i-1}
for l←1 to n          // 求l 个元素的Opti-BST
      for i←1 to n-l+1
            j←i+l-1
            e[i, j]←∞
            w[i, j]←w[i, j-1]+p_j+q_j
            for r←i to j
                  t←e[i, r-1]+e[r+1, j]+w[i, j]
                  if t<e[i, j]
                        e[i, j]←t
                        root[i, j]←r
return e and root
```



Step 3: Computing the expected search cost

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i,j-1] + p_j + q_j \qquad (15.20)$$

```
OBST(p, q, n)
for i←1 to n+1
    e[i, i-1]←q_{i-1}
    w[i, i-1]←q_{i-1}
for l←1 to n
    for i←1 to n-l+1   // n-l+1 times
        j←i+l-1
        e[i, j]←∞
        w[i, j]←w[i, j-1]+p_j+q_j
        for r←i to j      // j-i+1=i+l-1-i+1=l
            t←e[i, r-1]+e[r+1, j]+w[i, j]
            if t<e[i, j]
                e[i, j]←t
                root[i, j]←r
return e and root
```

Running times ?

Step 3: Computing the expected search cost

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i,j-1] + p_j + q_j \qquad (15.20)$$

```
OBST(p, q, n)
for i←1 to n+1
    e[i, i-1]←q_{i-1}
    w[i, i-1]←q_{i-1}
for l←1 to n
    for i←1 to n-l+1    // n-l+1 times
        j←i+l-1
        e[i, j]←∞
        w[i, j]←w[i, j-1]+p_j+q_j
        for r←i to j    // j-i+1=i+l-1-i+1=l
            t←e[i, r-1]+e[r+1, j]+w[i, j]
            if t<e[i, j]
                e[i, j]←t
                root[i, j]←r
return e and root
```

Running times: $\Theta(n^3)$

Proof:

$O(n^3)$: for loops are nested three deep and each loop index takes on at most n values.

$\Omega(n^3)$:

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

$$= \frac{(n+1)(n+1)n}{2} - \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{n(n+1)(n+2)}{6} = \Omega(n^3)$$

Exercise-5 (in class)

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } i \le j. \end{cases} \qquad (15.19)$$

$$w[i,j] = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l = w[i,j-1] + p_j + q_j \qquad (15.20)$$

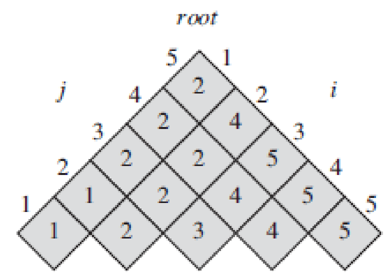
Based on equations (15.19) and (15.20) from the OBST,

What is the raw Recursive algorithm?

What is the Recursive algorithm with Memoization?

Running time?

Exercise-6



15.5-1 **Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST(root) which, given the table root, outputs the structure of an optimal binary search tree. For the example in Figure 15.8, your procedure should print out the**

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

Solution of Exercise-6

15.5-1

```
A(r, i, j)
if i=1 && j=n
    r[i, j] is the root
if i=j
    print "d"i-1 "is the left child of k"i
    print "d"i "is the right child of k"i
else if r[i, j]=i
        print "d"i-1 "is the left child of k"i
        print "k"r[i+1, j] "is the right child of k"r[i, j]
        A(r, i+1, j)
    else if r[i, j]=j
            print "k"r[i, j-1] is the left child of k"r[i, j]
            A(r, i, j-1)
            print "d"j "is the right child of k"j
        else // i<r[i, j]<j
            print "k"r[i, r[i, j]-1] "is the left child of k"r[i, j]
            A(r, i, r[i,j]-1)
            print "k"r[r[i, j]+1, j] "is the right child of k"r[i, j]
            A(r, r[i, j]+1, j)
```

Big Exercises

根据一本专业书籍（如《算法导论》），建设一个翻译软件中计算机类词库（字典）的OBST。说明：只考虑英语单词作为关键字。

求解思路：

1. 统计书籍里有多少个单词 M
2. 按字母序，第 i $(1 \le i \le n)$ 个单词在书中出现了 k_i 次 $(k_1 + k_2 + \ldots + k_n = M)$，其词频 $p_i = k_i / M$
3. 根据词频表，构建OBST

思考：对比一般的平衡搜索树 BBST（用中间点作为树根），比较 BBST 与 OBST 的搜索代价。