3.3 DP and Optimization Problems

- **P can be solved using DP if the principle of optimality applies in P.**
- **Def.: The principle of optimality is said to apply in a problem if an optimal solution to an instance of a problem always contains optimal solutions to all subinstances.**
- **Ex) *shortest path problem***
  - **If the path from v_i to v_j is optimal, then the subpaths from v_i to v_k and from v_k to v_j must be optimal.**

$$v_i \Rightarrow v_k \Rightarrow v_j$$

Counterexample

- **Ex)** ***longest path problem***
  - **See Fig. 3.6 (consider simple path only)**

$$\underbrace{v_1 \to v_3}_{\text{NOT optimal}} \to v_2 \to v_4 \quad \text{longest}$$

$$(v_1 \to v_2 \to v_3 \quad \text{optimal})$$

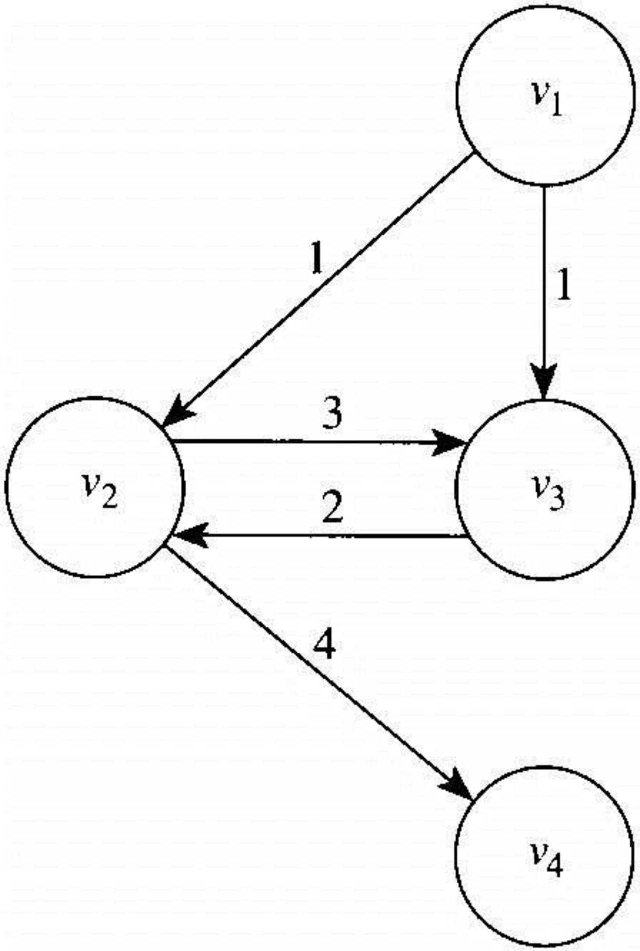


Figure 3.6 A weighted, directed graph with a cycle.

3.4 Chained Matrix Multiplication

- **ijk multiplication** $A_{i \times j} \bullet B_{j \times k} = C_{i \times k}$

Ex) **A x B x C x D**

20x2 2x30 30x12 12x8

((AB)C)D 20x2x30+20x30x12+20x12x8 = 10,320

A((BC)D) 2x30x12+2x12x8+20x2x8 = 1,232

- **Goal: determine the optimal order for multiplying *n* matrices**

Basic Observation

- **Brute-force algorithm**
  - **Let t_n be the # of different orders in which we can multiply n matrices**

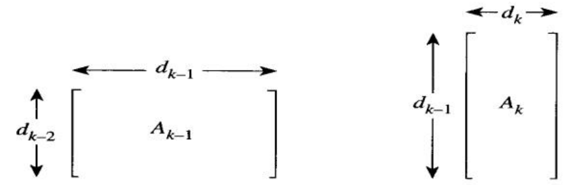
$$A_1\underbrace{(A_2 \cdots A_n)}_{t_{n-1}} \qquad \underbrace{(A_1 \cdots A_{n-1})}_{t_{n-1}}A_n$$

$$t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$$

$$t_2 = 1 \qquad t_n \geq 2^{n-2}$$

- **Principle of optimality applies?**
  - **If $A_1((((A_2A_3)A_4)A_5)A_6)$ is optimal**
  - **$(A_2A_3)A_4$ is also optimal**
  - **Proof by contradiction**

Figure 3.7 The number of columns in A_{k-1} is the same as the number of rows in A_k.



Example

$$\begin{array}{ccccc} A_1 & A_2 & \cdots & A_k & \cdots & A_n \\ d_0 \times d_1 & d_1 \times d_2 & & d_{k-1} \times d_k & & d_{n-1} \times d_n \end{array}$$

Let $M[i][j] = \text{min \# of multiplications needed}$

$\text{to multiply } A_i \text{ through } A_j \quad 1 \le i \le j \le n$

$M[i][i] = 0 \qquad \text{Want} : M[1][n]$

Ex. 3.5

$$\begin{array}{ccccccc} & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\ d_0 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 \\ 5 & 2 & 3 & 4 & 6 & 7 & 8 \end{array}$$

$$\#(A_4 A_5) A_6 = \underset{4}{d_3} \times \underset{6}{d_4} \times \underset{7}{d_5} + \underset{4}{d_3} \times \underset{7}{d_5} \times \underset{8}{d_6} = 392$$

$$\# A_4 (A_5 A_6) = \underset{6}{d_4} \times \underset{7}{d_5} \times \underset{8}{d_6} + \underset{4}{d_3} \times \underset{6}{d_4} \times \underset{8}{d_6} = 528$$

$$M[4][6] = \min(392,\ 528) = 392$$

Factorization (1/2)

- **Multiplying 6 matrices**

1. $(A_1)(A_2A_3A_4A_5A_6)$
2. $(A_1A_2)(A_3A_4A_5A_6)$
3. $(A_1A_2A_3)(A_4A_5A_6)$
4. $(A_1A_2A_3A_4)(A_5A_6)$
5. $(A_1A_2A_3A_4A_5)(A_6)$

- **One of these is optimal**

$$M[1][6] = \min_{1 \le k \le 5} (M[1][k] + M[k+1][6] + d_0 d_k d_6)$$

- **In general**

$$\underset{\substack{d_{i-1} \times d_k \\ M[i][k]}}{(A_i \cdots A_k)} \; \underset{\substack{d_k \times d_j \\ M[k+1][j]}}{(A_{k+1} \cdots A_j)}$$

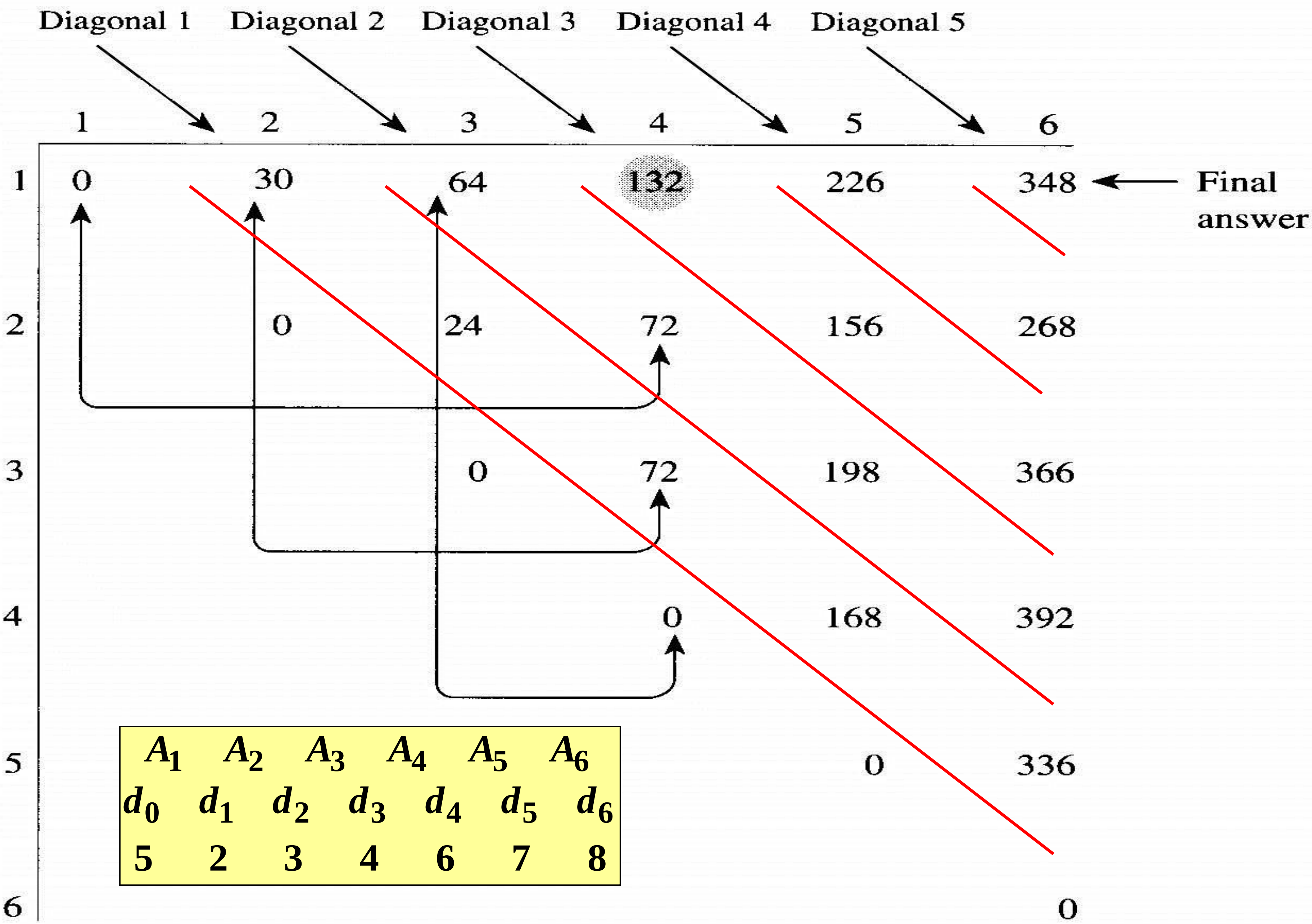
$$M[i][j] = \min_{i \le k \le j\text{-}1} (M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) \quad \text{if} \quad i < j$$

$$M[i][i] = 0$$

Factorization (2/2)

- **Computing order**
  - **From main diagonal ($j - i = 0$), diagonal 1($j - i = 1$), diagonal 2 ($j - i = 2$), etc.**
- **See Ex. 3.6 and Fig. 3.8**
- **See Alg. 3.6**

Figure 3.8 The array M developed in Example 3.6. $M[1][4]$, which is circled, is computed from the pairs of entries indicated.



Example 3.6 (1/2)

A_1		A_2		A_3		A_4		A_5		A_6	
d_0	d_1	d_2	d_3	d_4	d_5	d_6					
5	2	3	4	6	7	8					

- **Compute diagonal 0:**

$$M[i][i] = 0 \text{ for } 1 \le i \le 6$$

- **Compute diagonal 1:**

$$\begin{aligned} M[1][2] &= \min_{1 \le k \le 1} (M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30 \end{aligned}$$

$$M[2][3],\ M[3][4],\ M[4][5], \text{ and } M[5][6]$$

- **Compute diagonal 2:**

$$\begin{aligned} M[1][3] &= \min_{1 \le k \le 2} (M[1][k] + M[k+1][3] + d_0 d_k d_3) \\ &= \min(M[1][1] + M[2][3] + d_0 d_1 d_3, \\ &\qquad M[1][2] + M[3][3] + d_0 d_2 d_3 \\ &= \min(0 + 24 + 5 \times 2 \times 4,\ 30 + 0 + 5 \times 3 \times 4) = 64 \end{aligned}$$

$$M[2][4],\ M[3][5], \text{ and } M[4][6]$$

Example 3.6 (2/2)

A_1	A_2	A_3	A_4	A_5	A_6	
d_0	d_1	d_2	d_3	d_4	d_5	d_6
5	2	3	4	6	7	8

- **Compute diagonal 3:**

$$M[1][4] = \min_{1 \le k \le 3} (M[1][k] + M[k+1][4] + d_0 d_k d_4)$$
$$= \min(M[1][1] + M[2][4] + d_0 d_1 d_4,$$
$$M[1][2] + M[3][4] + d_0 d_2 d_4,$$
$$M[1][3] + M[4][4] + d_0 d_3 d_4,$$
$$= \min(0 + 72 + 5 \times 2 \times 6,\ 30 + 72 + 5 \times 3 \times 6,$$
$$64 + 0 + 5 \times 4 \times 6) = 132$$

$M[2][5]$, and $M[3][6]$

- **Compute diagonal 4:**

$M[1][5]$, and $M[2][6]$

- **Compute diagonal 5:**

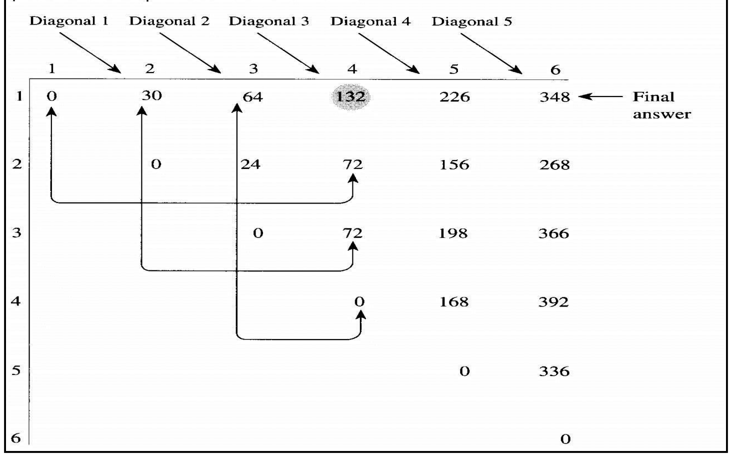
$M[1][6] = 348$

Algorithm 3.6 Minimum Multiplications (1/2)

- <u>Problem</u>: Determining the minimum number of elementary multiplications needed to multiply n matrices and an order that produces that minimum number.
- <u>Inputs</u>: the number of matrices n, and an array of integers d, indexed from 0 to n, where $d[i-1] \times d[i]$ is the dimension of the i-th matrix.
- <u>Outputs</u>: *minmult*, the minimum number of elementary multiplications needed to multiply the n matrices; a 2D array P from which the optimal order can be obtained. P has its rows indexed from 1 to $n-1$ and its columns indexed from 1 to n. $P[i][j]$ is the point where matrices i through j are split in an optimal order for multiplying the matrices.

Algorithm 3.6 Minimum Multiplications (2/2)

Figure 3.8 The array M developed in Example 3.6. $M[1][4]$, which is circled, is computed from the pairs of entries indicated.




```
int minmult(int n, const int d[ ], index P[ ] [ ])
{
   index i, j, k, diagonal;
   int M[1..n][1..n];
   for (i = 1; i <= n; i++)
      M[i][i] = 0;
   for (diagonal = 1; diagonal <= n-1; diagonal ++)
                  // diagonal-1 is just above the main diagonal.
      for (i = 1; i <= n - diagonal; i++){
         j = i + diagonal;   // remember that "# of alternatives = diagonal"
         M[i][j] = minimum        (M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j]);
                   i ≤ k ≤ j–1
         P[i][j] = a value of k that gave the minimum;
      }
   return M[1][n];
}
```

$$M[i][j] = \min_{i \le k \le j\text{-}1} (M[i][k] + M[k+1][j] + d_{i-1}d_k d_j) \text{ if } i < j$$

$$M[i][i] = 0$$

T(n) of Alg. 3.6

- **Basic operation: instruction executed for each value of k and (min) comparison**
- **Input size: n = (# of matrices)**
- **Given diagonal**
  - **for i-loop n - diagonal**
  - **for k-loop j–1 – i + 1 = i + diagonal – i = diagonal**

$$T(n) = \sum_{d=1}^{n-1} (n-d) \cdot d = n \cdot \frac{n(n-1)}{2} - \frac{(n-1)n(2n-1)}{6}$$

$$= \frac{(n-1)n(n+1)}{6} \in \Theta(n^3)$$

- **Note: Yao(1982) $\Theta(n^2)$ Hu & Shing(1982,84) $\Theta(n \log n)$**

Algorithm 3.7 Print Optimal Order (1/2)

- Problem: Print the optimal order for multiplying *n* matrices.
- Inputs: positive integer *n*, and the array *P* produced by Alg. 3.6. *P*[*i*][*j*] is the point where matrices *i* through *j* are split in an optimal order for multiplying those matrices. (*P*[i][j] = a value of k that minimize M[i][j])
- Outputs: the optimal order for multiplying the matrices.

Algorithm 3.7 Print Optimal Order (2/2)

Figure 3.9 The array *P* produced when Algorithm 3.6 Example 3.5.

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

- **P[2][5] = 4 → $(A_2A_3A_4)A_5$**
- **P[1][6] = ?**

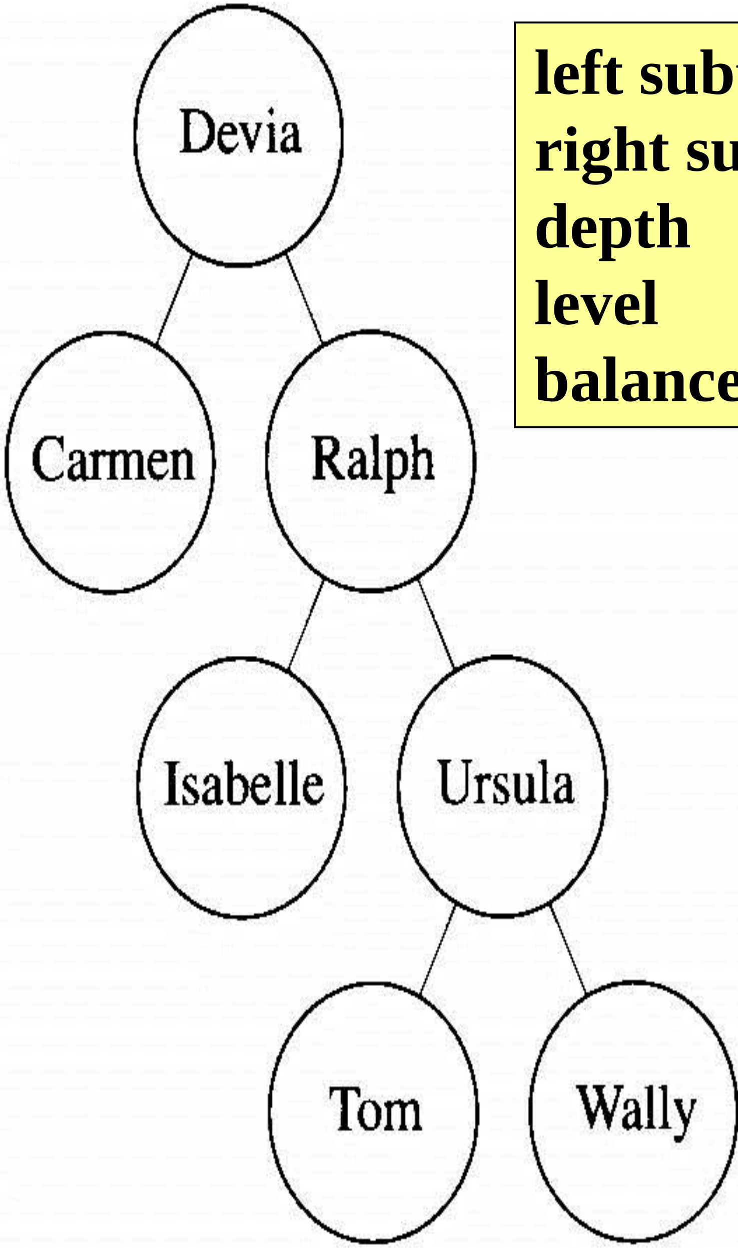
```
void order(index i, index j)
{
   if (i == j)
      cout <<"A" << i;
   else {
      k = P[i][j];
      cout <<"(";
      order(i, k);
      order(k + 1, j);
      cout <<")";
   }
}
```

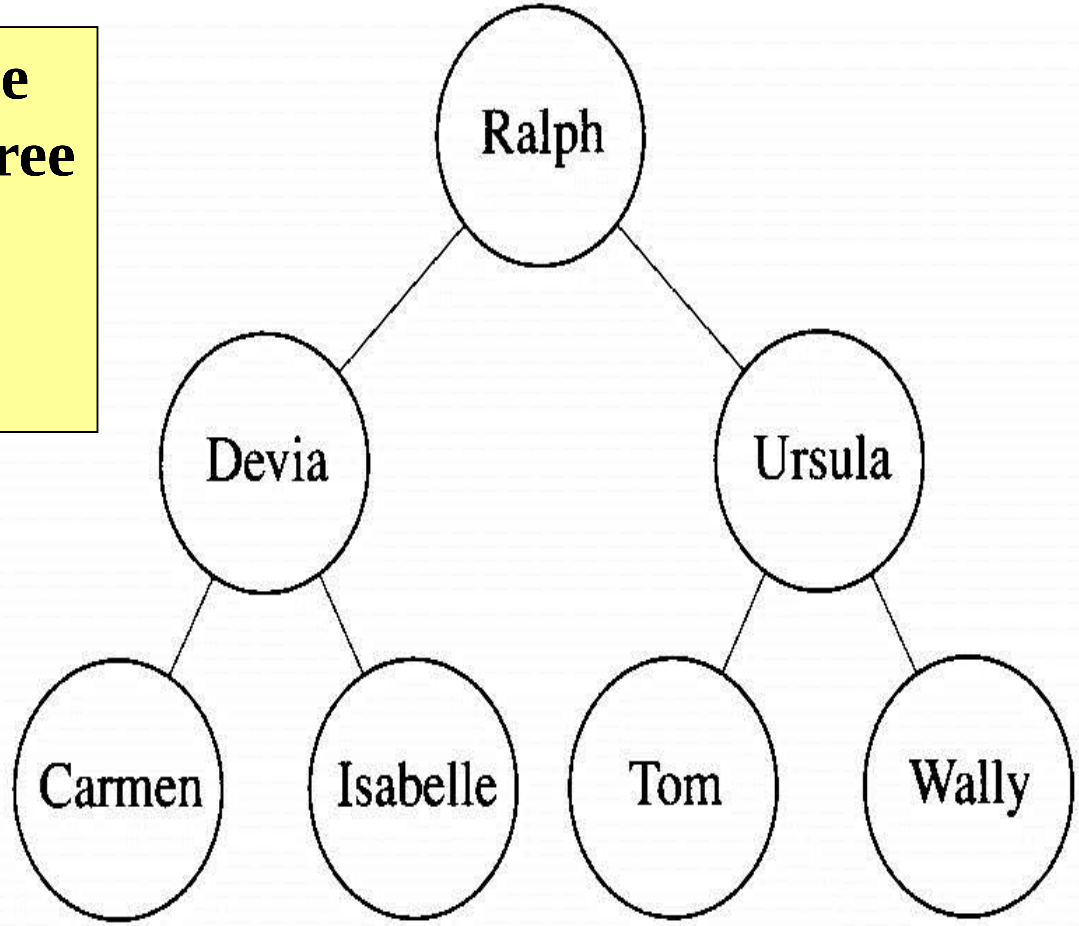
3.5 Optimal Binary Search Trees

- **Def.: A BST is a binary tree of items (ordinary called keys), that come from an ordered set, such that**

1. **Each node contains one key.**
2. **The keys in the left subtree of a given node are less than or equal to the key in that node.**
3. **The keys in the right subtree of a given node are greater than or equal to the key in that node.**

left subtree
right subtree
depth
level
balanced





Goal: to organize the keys in a BST so that the average time it takes to locate a key is minimized (Optimal BST)

Figure 3.10 Two binary search trees.

Data Type Used in Algorithm

```
struct nodetype
{
    keytype key;
    nodetype* left;
    nodetype* right;
};
typedef nodetype* node_pointer;
```

Algorithm 3.8 Search Binary Tree

- **Problem: Determine the node containing a key in a BST. It is assumed that the key is in the tree.**
- **Inputs: a pointer *tree* to a BST and a key *keyin*.**
- **Outputs: a pointer *p* to the node containing the key.**

```
void search(node_pointer tree, keytype keyin, node_pointer& p)
{
   bool found;
   p = tree;
   found = false;
   while (!found)
      if(p->key ==keyin)
         found = true;
      else if(keyin < p->key)
         p = p->left;     // Advance to left child.
      else
         p = p->right;  // Advance to right child.
}
```

Search time for a given key
***depth*(*key*) + 1**

Average Search Time

- Let $K_1, K_2, \ldots, K_n$ be the n keys in order
  - Let p_i be the probability that K_i be the search key.
  - Let c_i be the number of comparisons needed to find K_i.

$$\text{minimize} \quad \sum_{i=1}^{n} c_i p_i \qquad \text{average search time}$$

Example 3.7

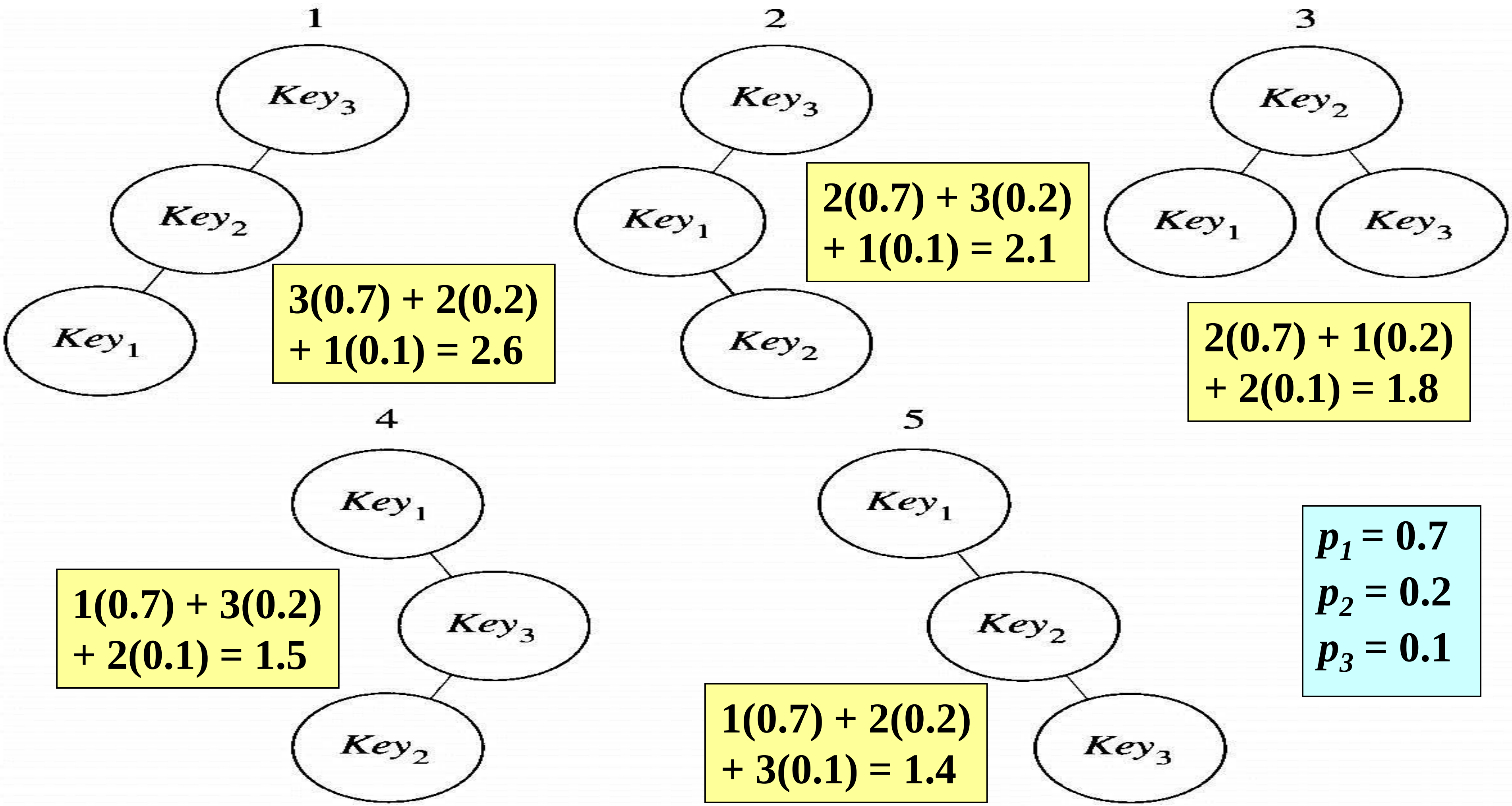


Figure 3.11 The possible binary search trees when there are three keys.

Principle of Optimality

- **Principle of optimality applies?**
  - **Any subtree of OBST is optimal.**
  - **Proof by contradiction.**

Formulation

- **Suppose that K_i through K_j are arranged in a tree that minimize** $\sum_{m=i}^{j} c_m p_m$
  - **We will call such a tree optimal**
- **Let A[i][j] denote the value of the OBST with $K_i, \ldots, K_j$**
  - **A[i][i] = p_i**

Example 3.8

- $p_1 = 0.7$, $p_2 = 0.2$, $p_3 = 0.1$.
- **To determine A[2][3], we must consider the two trees**
  1. $1(p_2)+2(p_3) = 1(0.2)+2(0.1) = 0.4$
  2. $2(p_2)+1(p_3) = 2(0.2)+1(0.1) = 0.5$
- **The first tree is optimal, and A[2][3] = 0.4.**

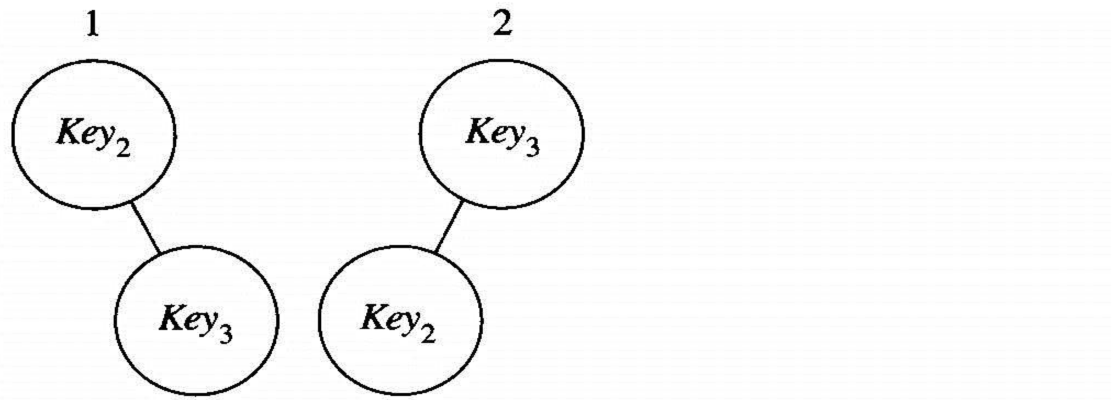
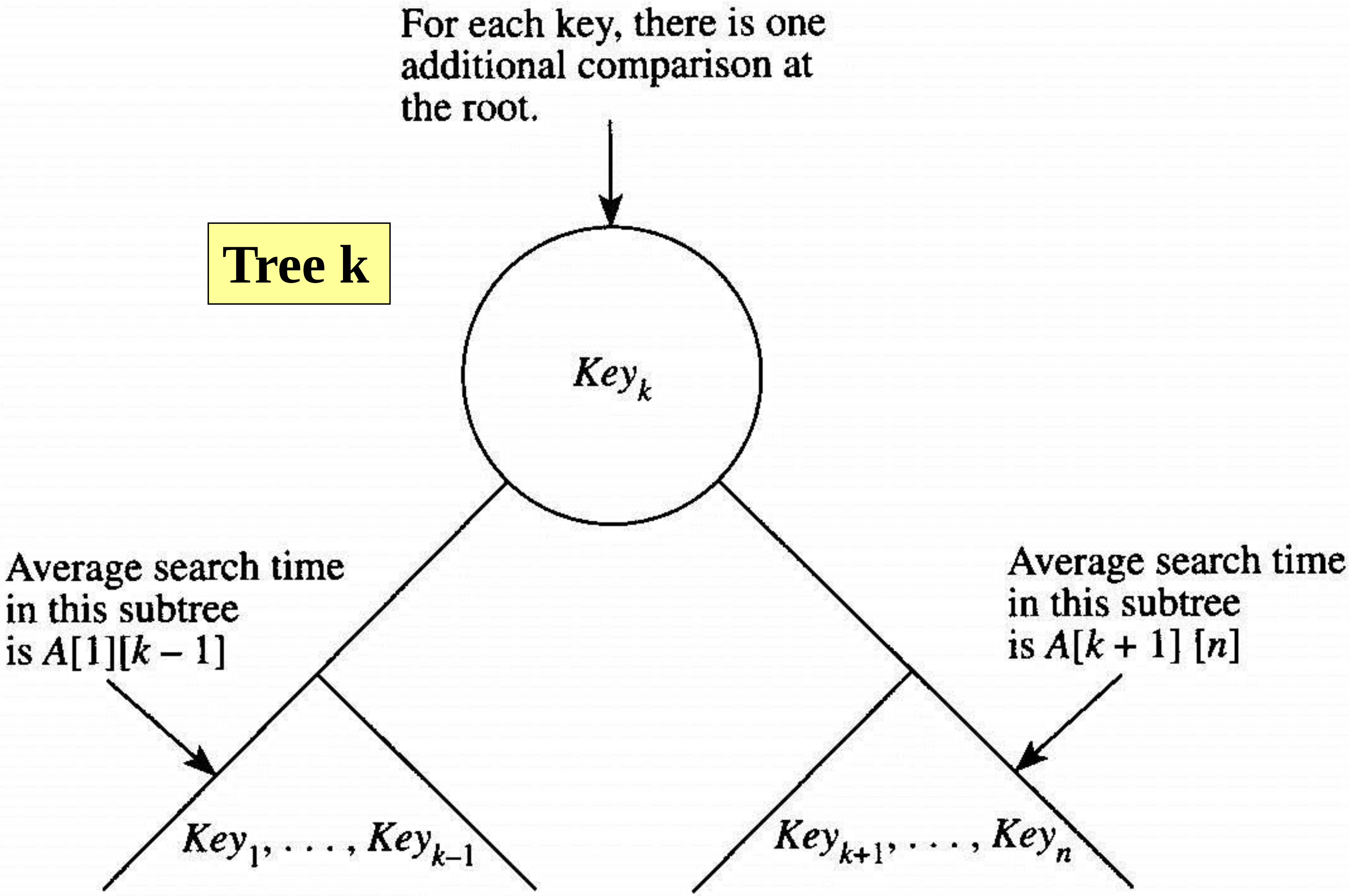


Figure 3.12 The binary search trees composed of Key_2 and Key_3.

Figure 3.13 Optimal binary search tree given that Key_k is at the root.



Average search time for tree

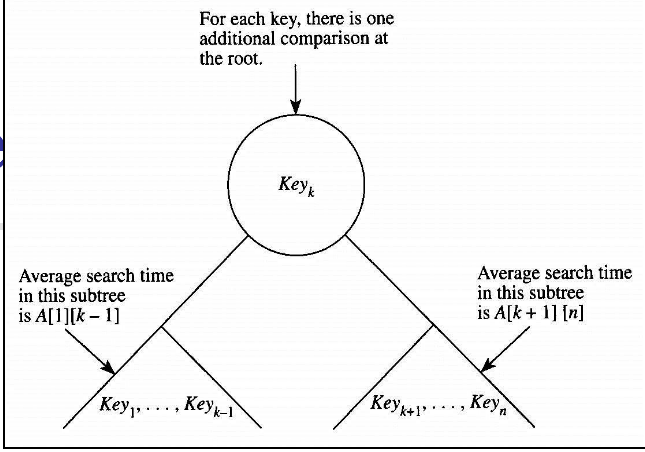
- $A[1][k-1] + (p_1 + \ldots + p_{k-1}) + p_k + A[k+1][n] + (p_{k+1} + \ldots + p_n)$
  - $A[1][k-1]$: average time in left subtree
  - $(p_1 + \ldots + p_{k-1})$: additional time comparing at root
  - p_k: average time searching for root
  - $A[k+1][n]$: average time in right subtree
  - $(p_{k+1} + \ldots + p_n)$: additional time comparing at root

$$= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

$$A[1][n] = \min_{1 \le k \le n} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

$$A[1][0] = A[n+1][n] = 0$$

Figure 3.13 Optimal binary search tree given that Key_k is at the root.



Generalization

$$A[i][j] = \min_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad ; \quad i < j$$

$$A[i][i] = p_i$$

$$A[i][i-1] = A[j+1][j] = 0$$

- **Compute smaller j - i first**
- **See Alg. 3.9**
  - **R[i][j] has the root of OBST**

$$W_{ij} = \sum_{m=i}^{j} p_m \qquad W_{ij} = W_{i,j-1} + p_j$$

Algorithm 3.9 Optimal Binary Search Tree (1/3)

- **<u>Problem</u>: Determine an OBST for a set of keys, each with a given probability of being the search key.**
- **<u>Inputs</u>: *n*, the number of keys, and an array of real numbers *p* indexed from 1 to *n*, where *p*[*i*] is the probability of searching for the *i*-th key.**
- **<u>Outputs</u>: a variable *minavg*, whose value is the average search time for an OBST; and a 2D array *R* from which an optimal tree can be constructed. *R* has its rows indexed from 1 to *n* + 1 and its columns indexed from 0 to *n*. *R*[*i*][*j*] is the index of the key in the root of an optimal tree containing the *i*-th through the *j*-th keys.**

Algorithm 3.9 Optimal Binary Search Tree (2/3)

```
void optsearchtree(int n, const float p[ ], float& minavg, index R[ ][ ])
{
   index i, j, k, diagonal;
   float A[1..n+1][0..n];
   for (i = 1; i <= n; i++){
      A[i][i-1] = 0;
      A[i][i] = P[i];
      R[i][i] = i;
      R[i][i-1] = 0;
   }
```

$$\mathrm{A[i][i]} = p_i$$

$$\mathrm{A[i][i-1]} = \mathrm{A[j+1][j]} = 0$$

Algorithm 3.9 Optimal Binary Search Tree (3/3)

$$A[i][j] = \min_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad ; \quad i < j$$

$$A[i][i-1] = A[j+1][j] = 0$$

```
A[n+1][n] = 0;
R[n+1][n] = 0;
for (diagonal = 1; diagonal <= n - 1; diagonal ++)
        // diagonal-1 is just above the main diagonal.
    for (i = 1; i <= n - diagonal; i++){
        j = i + diagonal; // remember that "# of alternatives = diagonal+1"
        A[i][j] = minimum_{i≤k≤j} (A[i][k-1] + A[k+1][j]) + Σ_{m=i}^{j} p_m ;
        R[i][j] = a value of k that gave the minimum;
    }
    minavg = A[1][n];
}
```

T(n) of Alg. 3.9

- **Basic operation: instruction executed for each value of *k* and (min) comparison**
- **Input size: *n* = (# of keys)**

$$T(n) = \frac{(n-1)n(n+4)}{6} \quad \in \quad \Theta(n^3) \ \text{ similarly as Alg. 3.6}$$

- **Note: Yao(1982)** $\Theta(n^2)$

Algorithm 3.10 Build OBST (1/2)

- <u>**Problem**</u>**: build an OBST.**
- <u>**Inputs**</u>**: *n*, the number of keys, an array *Key* containing the *n* keys in order, and the array *R* produced by Algorithm 3.9. *R*[*i*][*j*] is the index of the key in the root of an optimal tree containing the *i*-th through the *j*-th keys.**
- <u>**Outputs**</u>**: a pointer *tree* to an OBST containing the *n* keys.**

Algorithm 3.10 Build OBST (2/2)

```
node_pointer tree(index i, j )
{
    index k;
    node_pointer p;
    k = R[i][j];
    if (k == 0)
      return NULL;
    else {
        p = new nodetype;
        p->key = Key[k];
        p->left = tree(i, k-1);
        p->right = tree(k+1, j);
        return p;
    }
}
```

Example 3.9

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

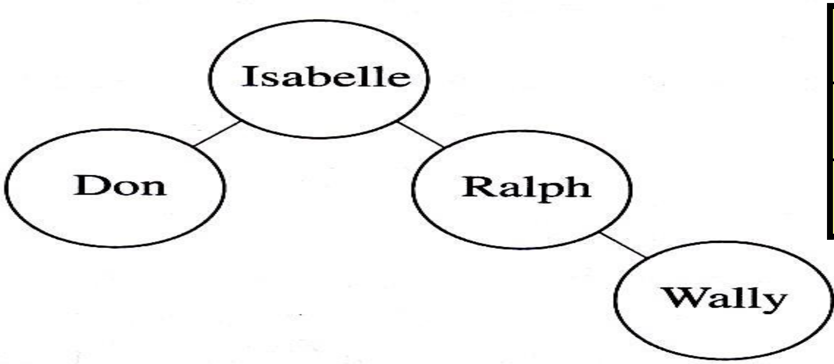
A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R

Figure 3.14 The arrays *A* and *R*, produced when Algorithm 3.9 is applied to the instance in Example 3.9.

Figure 3.15 The tree produced when Algorithms 3.9 and 3.10 are applied to the instance in Example 3.9.



Key[i]	*Key*[1]	*Key*[2]	*Key*[3]	*Key*[4]
Instance	**Don**	**Isabelle**	**Ralph**	**Wally**
p_i	**3/8**	**3/8**	**1/8**	**1/8**

Example 3.7

A[1][1] = 0.7, A[2][2] = 0.2, A[3][3]=0.1

A[1][2] = min(A[1][0]+A[2][2], A[1][1]+A[3][2])+(0.7+0.2)

= 1.1

R[1][2] = 1

A[2][3] = min(A[2][1]+A[3][3], A[2][2]+A[4][3])+(0.2+0.1)

= 0.4

R[2][3] = 2

A[1][3] = min(A[1][0]+A[2][3], A[1][1]+A[3][3],

A[1][2]+A[4][3])+(0.7+0.2+0.1)

= 1.4

R[1][3] = 1