

Chapter 3



Dynamic Programming

Cause of inefficiency for divide-and-conquer

- After division ...
 - Smaller instances are unrelated, e.g., *mergesort*
 - Smaller instances are related, e.g., *fibonacci*
 - repeatedly solve common instances
- Dynamic programming
 - bottom-up approach
 - use an array (table) to save solutions to small instances

Recall Algorithm 1.7

- Algorithm 1.7: n th Fibonacci Term (Iterative)
 - Problem: Determine the n th term in the Fibonacci sequence.
 - Inputs: a nonnegative integer n .
 - Outputs : $fib2$, the n th term in the Fibonacci sequence.

```
int fib2 (int n) {  
    index i;  
    int f[0 .. n];  
    f[ 0 ] = 0;  
    if (n > 0){  
        f[ 1 ] = 1;  
        for (i = 2; i <= n; i++)  
            f[ i ] = f[i - 1] + f [i -2 ]; }  
    return f[ n ];  
}
```

Steps

- Establish a recursive property that gives the solution to an instance of the problem
- Solve an instance of the problem in a bottom-up fashion by solving smaller instances first

The binomial coefficient

□ Definition

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } 0 \leq k \leq n$$

□ Recursive definition

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

The algorithm

□ Algorithm 3.1: Binomial Coefficient Using Divide-and-Conquer

- Problem: Compute the binomial coefficient.
- Inputs: nonnegative integers n and k , where $k \leq n$.
- Outputs: bin , the binomial coefficient .

```
int bin (int  $n$ , int  $k$ ) {  
    if (  $k == 0$  ||  $n == k$ )  
        return 1;  
    else  
        return bin ( $n-1$ ,  $k - 1$ )+bin ( $n - 1$ ,  $k$ );  
}
```

Using dynamic programming

- Using an array B to store coefficients
- Steps:
 - Establish a recursive property:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ or } j = i \end{cases}$$

- Solve an instance of the problem in a bottom-up fashion by computing the rows in B in sequence starting with the first row

Compute sequence of rows

	0	1	2	3	4		j	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
i								
n								

$B[i-1][j-1]$ $B[i-1][j]$
↓ ↓
→ $B[i][j]$

□ Let's compute $B[4][2]$

The algorithm

□ Algorithm 3.2: Binomial Coefficient Using Dynamic Programming

- Problem: Compute the binomial coefficient.
- Inputs: nonnegative integers n and k , where $k \leq n$.
- Outputs: *bin 2*, the binomial coefficient

```
int bin2 (int n, int k) {  
    index i, j;  
    int B[0..n] [0..k];  
    for (i = 0; i <= n; i++)  
        for (j = 0; j <= minimum (i, k); j++)  
            if (j == 0 || j == i)  
                B[i][j] = 1;  
            else  
                B[i][j] = B[i - 1][j - 1] + B[i - 1][j];  
    return B[n][k];  
}
```

Time complexity

- The number of passes through the for- j loop for each value of i

i	0	1	2	3	...	k	$k+1$...	n
Number of passes	1	2	3	4	...	$k+1$	$k+1$...	$k+1$

$$1 + 2 + 3 + 4 + \dots + k + \underbrace{(k+1) + (k+1) + \dots + (k+1)}_{n-k+1 \text{ times}} =$$

$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$$

Floyd's algorithm for shortest paths

A common problem encountered by air travellers is the determination of the shortest way to fly from one city to another.

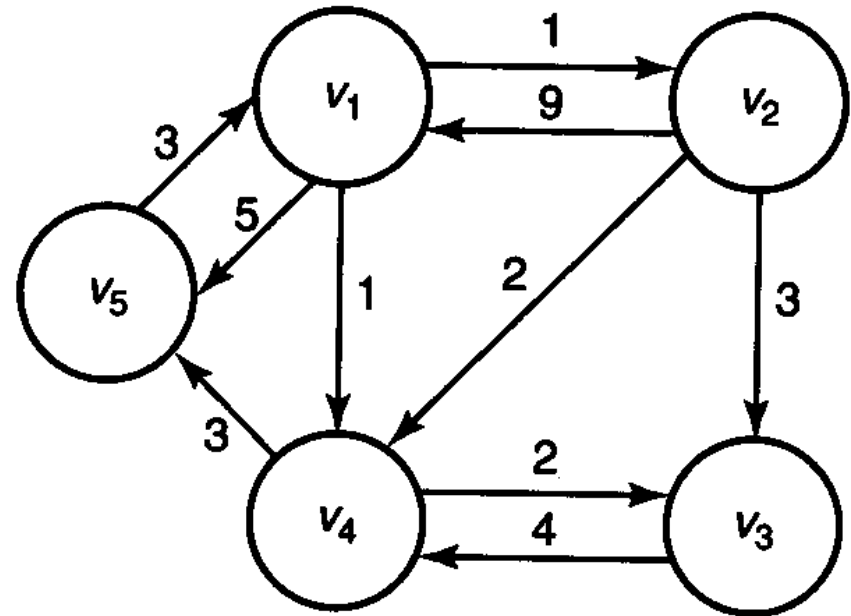


Figure 3.2 ● A weighted, directed graph.

Terminology

- If each edge has a direction associated with it, the graph is called a *directed graph* or ***digraph***.
- If the edges have values associated with them, the values are called ***weights*** and the graph is called a ***weighted graph***.
- In a directed graph, a ***path*** is a sequence of vertices such that there is an edge from each vertex to its successor.

Terminology (2)

- A path from a vertex to itself is called a **cycle**. The path $[v_1, v_4, v_5, v_1]$ is a cycle.
- A path is called **simple** if it never passes through the same vertex twice. The path $[v_1, v_2, v_3]$ is simple, but the path $[v_1, v_4, v_5, v_1, v_2]$ is not simple.
- The **length** of a path in a weighted graph is the sum of the weights on the path.

Shortest path problem

- Optimization problem
 - there can be more than one *candidate solution*
 - each candidate solution has a *value*
 - a *solution* is any candidate solution that has an *optimal value*
- A subset of all the paths from one vertex to another vertex is the set of the all those paths that start at the first vertex, end at the other vertex, and pass through all the other vertices.
 $(n - 2)(n - 3) \dots 1 = (n - 2)!$

Representation of the graph

□ Adjacency matrix

$$W[i][j] = \begin{cases} \text{weight on edge if there is an edge from } v_i \text{ to } v_j \\ \infty \text{ if there is no edge from } v_i \text{ to } v_j \\ 0 \text{ if } i = j \end{cases}$$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Figure 3.3 • *W* represents the graph in Figure 3.2 and *D* contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in *D* from those in *W*.

The algorithm

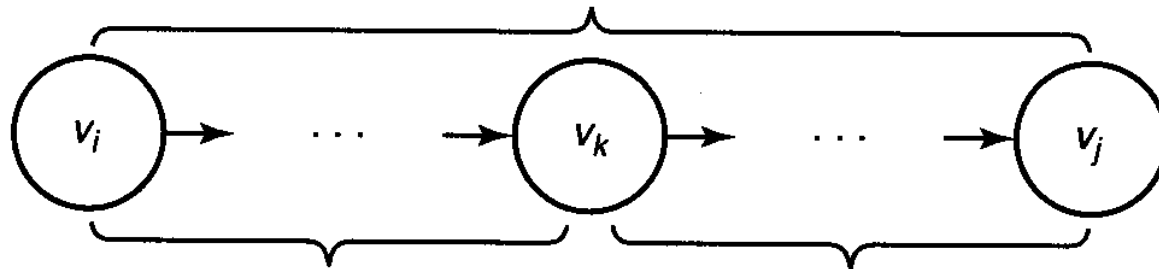
- Create a sequence of $n+1$ arrays $D^{(k)}$, where $0 \leq k \leq n$ and where
$$D^{(k)}[i][j] = \text{length of a shortest path from } v_i \text{ to } v_j \text{ using only vertices in the set } \{v_1, v_2, \dots, v_k\} \text{ as intermediate vertices}$$
$$D^{(0)} = W \text{ and } D^{(n)} = D$$
- Try to determine $D[2][5]$

To compute $D^{(k)}$ from $D^{(k-1)}$

- Case 1. At least one shortest path from v_i to v_j , using only vertices in $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices, does not use v_k . Then $D^{(k)}[i][j] = D^{(k-1)}[i][j]$
- Case 2. All shortest paths from v_i to v_j , using only vertices in $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices, do use v_k . Then $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$

Put together

A shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$



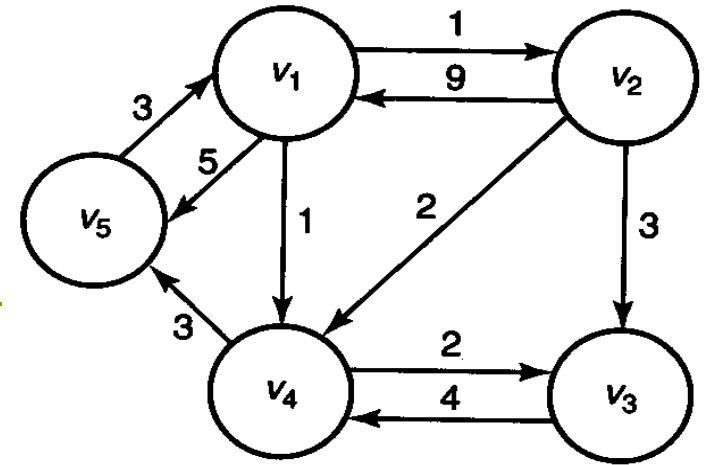
A shortest path from v_i to v_k using only vertices in $\{v_1, v_2, \dots, v_k\}$

A shortest path from v_k to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$

Figure 3.4 • The shortest path uses v_k .

- $D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$
- Try to compute $D^{(2)}[5][4]$

$$D^0 = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \hline \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$



$$D^1[1][2] = \text{Min}\{D^0[1][2], D^0[1][1] + D^0[1][2]\} = 1$$

$$D^1[1][3] = \text{Min}\{D^0[1][3], D^0[1][1] + D^0[1][3]\} = \infty$$

$$D^1[1][4] = \text{Min}\{D^0[1][4], D^0[1][1] + D^0[1][4]\} = 1$$

$$D^1[1][5] = \text{Min}\{D^0[1][5], D^0[1][1] + D^0[1][5]\} = 5$$

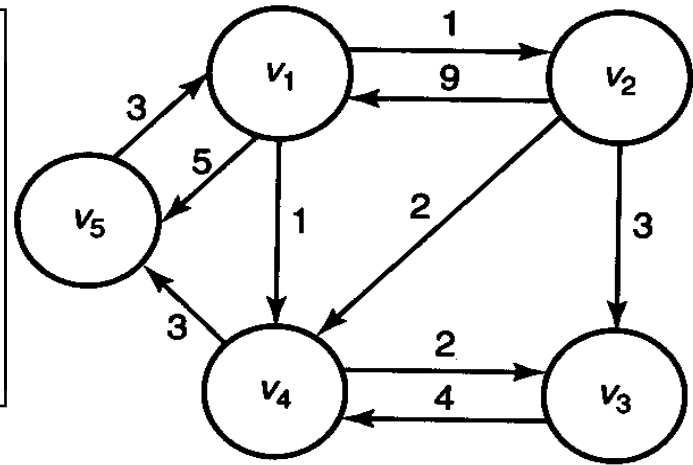
$$D^1[2][1] = \text{Min}\{D^0[2][1], D^0[2][1] + D^0[1][1]\} = \text{Min}\{9, 9 + 0\} = 9$$

$$D^1[2][3] = \text{Min}\{D^0[2][3], D^0[2][1] + D^0[1][3]\} = \text{Min}\{3, 9 + \infty\} = 3$$

$$D^1[2][4] = \text{Min}\{D^0[2][4], D^0[2][1] + D^0[1][4]\} = \text{Min}\{2, 9 + 1\} = 2$$

$$D^1[2][5] = \text{Min}\{D^0[2][5], D^0[2][1] + D^0[1][5]\} = \text{Min}\{\infty, 9 + 5\} = 14$$

$$D^0 = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \quad D^1 = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \underline{14} \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \underline{4} & \infty & \underline{4} & 0 \end{bmatrix}$$



$$D^2 = \begin{bmatrix} 0 & 1 & \underline{4} & 1 & 5 \\ 9 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 4 & \underline{7} & 4 & 0 \end{bmatrix} \quad D^3 = \begin{bmatrix} 0 & 1 & 4 & 1 & 5 \\ 9 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 4 & 7 & 4 & 0 \end{bmatrix}$$

$$D^4 = \begin{bmatrix} 0 & 1 & 3 & 1 & \underline{4} \\ 9 & 0 & 3 & 2 & \underline{5} \\ \infty & \infty & 0 & 4 & \underline{7} \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 4 & \underline{6} & 4 & 0 \end{bmatrix} \quad D^5 = \begin{bmatrix} 0 & 1 & 3 & 1 & 4 \\ \underline{8} & 0 & 3 & 2 & 5 \\ \underline{10} & \underline{11} & 0 & 4 & 7 \\ \underline{6} & \underline{7} & 2 & 0 & 3 \\ 3 & 4 & 6 & 4 & 0 \end{bmatrix}$$

The algorithm

- ❑ Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices.
- ❑ Inputs: A weighted, directed graph and n , the number of vertices in the graph.
- ❑ Outputs: A two-dimensional array D , which has both its rows and columns indexed from 1 to n , where $D[i][j]$ is the length of a shortest path from the i th vertex to the j th vertex.

```
void floyd (int  $n$ , const number  $W[][]$ , number  $D[][]$ )  
{  
  index  $i, j, k$ ;  
   $D = W$ ;  
  for ( $k = 1$ ;  $k \leq n$ ;  $k++$ )  
    for ( $i = 1$ ;  $i \leq n$ ;  $i++$ )  
      for ( $j = 1$ ;  $j \leq n$ ;  $j++$ )  
         $D[i][j] = \text{minimum}(D[i][j], D[i][k] + D[k][j]);$   
}
```

Every-case time complexity

- Basic operation: The instruction (if) in the for- j loop
- Input size: n , the number of vertices in the graph
- Time complexity:
$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

Floyd's algorithm for shortest paths

```
void floyd2 (int n, const number W[][i], number D[][j],  
             index P[][k])  
{  
    index, i, j, k;  
    for(i = 1; i <= n; i++)  
        for (j = 1; j <= n; j++)  
            P[i][j] = 0;  
  
    D = W  
    for (k = 1; k <= n; k++)  
        for(i = 1; i <= n; i++)  
            for(j = 1; j <= n; j++)  
                if (D[i][k] + D[k][j] < D[i][j]) {  
                    P[i][j] = k;  
                    D[i][j] = D[i][k] + D[k][j]; }  
}
```

Sample output

- The array P created when the algorithm is applied to the graph in Figure 3.2

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

Figure 3.5 • The array P produced when Algorithm 3.4 is applied to the graph in Figure 3.2.

Print shortest path

```
void path (index q, r)  
{
```

```
    if ( $P[q][r] \neq 0$ ) {  
        path (q,  $P[q][r]$ );
```

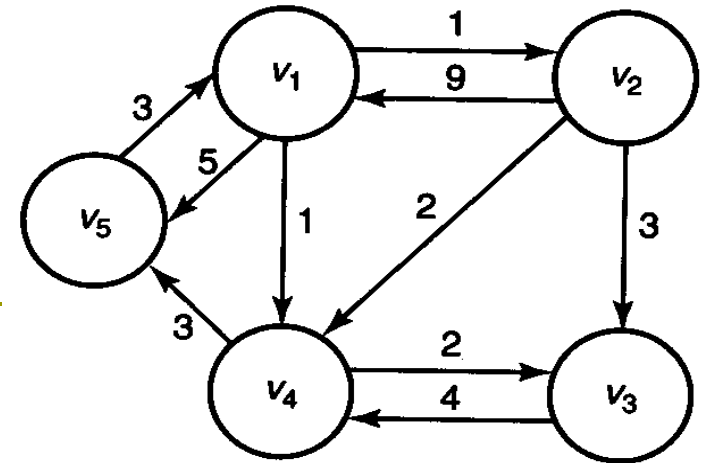
```
        cout << "v" <<  $P[q][r]$ ;
```

```
        path ( $P[q][r]$ , r);
```

```
    }
```

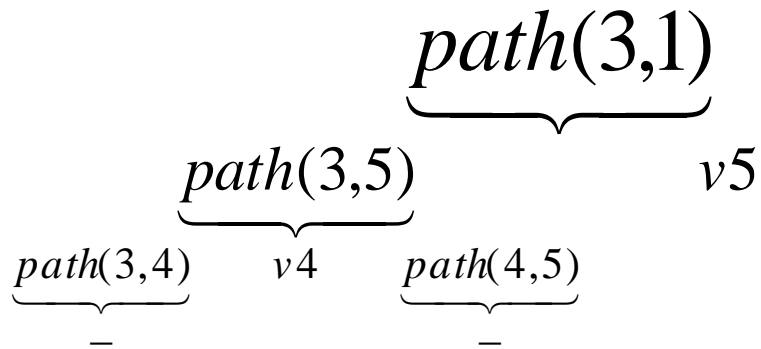
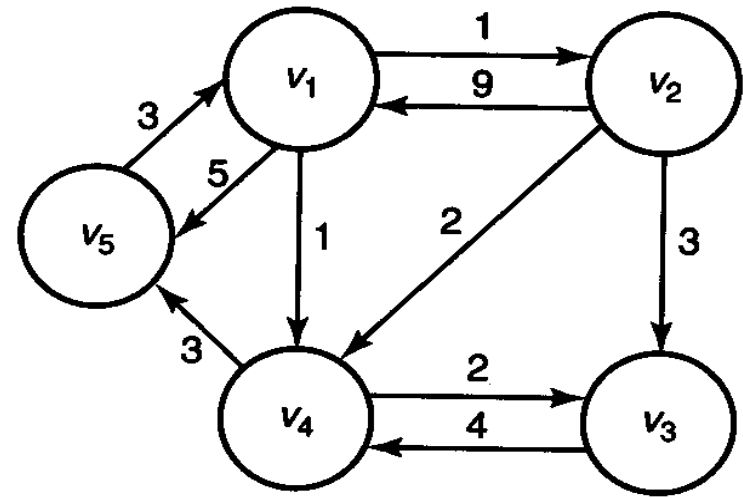
```
}
```

Try to determine path(5,3)



```
void path (index q, r)
```

```
{
  if (P[q][r] != 0)
  {
    path (q, P[q][r]);
    cout << "v" << P[q][r];
    path (P[q][r], r);
  }
}
```



$path(5,1)$

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

Dynamic programming and optimization problems

□ Steps to follow:

- *Establish* a recursive property that gives the optimal solution to an instance of the problem
- Compute the value of an optimal solution in a *bottom-up* fashion
- Construct an optimal solution in a *bottom-up* fashion

Principle of optimality

The optimal solution to the instance contains optimal solutions to all subinstances

Principle of optimality (Example)

if v_k is a vertex on an optimal path from v_i to v_j , then the subpaths from v_i to v_k and from v_k to v_j must also be optimal.

Principle of optimality

It is necessary to show that the principle applies before using dynamic programming to obtain the solution

Longest simple paths problem

- The longest path from v_1 to v_4 is $[v_1, v_3, v_2, v_4]$. However, the subpath $[v_1, v_3]$ is not optimal

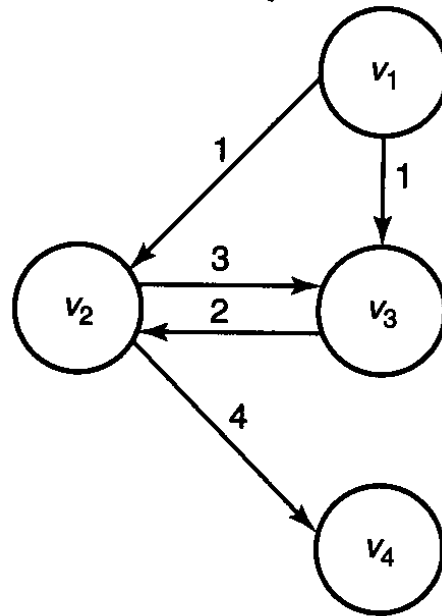


Figure 3.6 • A weighted, directed graph with a cycle.

Chained matrix multiplication

- To multiply an $i \times j$ matrix times a $j \times k$ matrix, the number of elementary multiplications is: $i \times j \times k$ (why?)
- Matrix multiplication is associative, which means there are different orders in which we can multiply multiple matrix, resulting in different number of elementary multiplications

Example

$$A \times B \times C \times D$$

$$20 \times 2 \quad 2 \times 30 \quad 30 \times 12 \quad 12 \times 8$$

$$A(B(CD)) \Rightarrow \text{The number of multiplications} = 3680$$

$$A((BC)D) \quad 1232$$

$$(AB)(CD) \quad 8880$$

$$(A(BC))D \quad 3120$$

$$((AB)C)D \quad 10320$$

Using dynamic programming

- Principle of optimality applies in this problem (example?)
- Some properties
 - the number of columns in A_{k-1} equals the number of rows in A_k
 - We can let d_0 be the number of rows in A_1 and d_k be the number of columns in A_k for $1 \leq k \leq n$, the dimension of A_k is $d_{k-1} \times d_k$

Introducing a sequence of arrays

- $M[i][j]$ = minimum number of multiplications needed to multiply A_i through A_j , if $i < j$.
- $M[i][i] = 0$
- Example

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

What is $M[4][6]$?

To multiply 6 matrices

□ Possible factorizations

- $A_1 (A_2 A_3 A_4 A_5 A_6)$
- $(A_1 A_2) (A_3 A_4 A_5 A_6)$
- $(A_1 A_2 A_3) (A_4 A_5 A_6)$
- $(A_1 A_2 A_3 A_4) (A_5 A_6)$
- $(A_1 A_2 A_3 A_4 A_5) A_6$

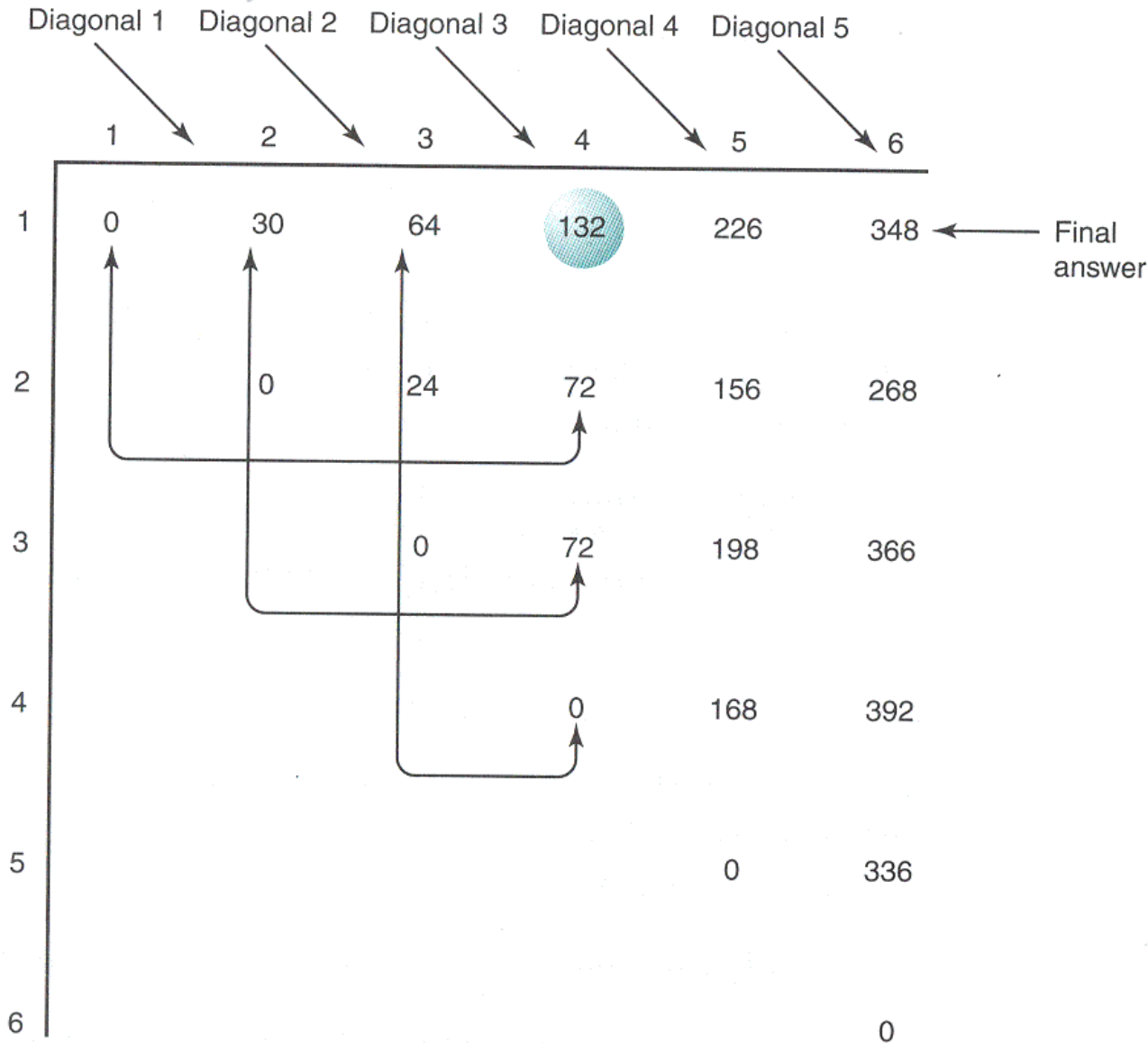
□ The number of multiplications for the k th factorization:

$$M[1][k] + M[k+1][6] + d_0 d_k d_6$$

□ Therefore (Recursive Equation)

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

$$\left\{ \begin{array}{l} M[i][j] = \min_{i \leq k \leq j-1} (M[i][k] + M[k+1][j] + d_{i-1}d_kd_j), \quad \text{if } i < j \\ M[i][i] = 0 \end{array} \right.$$



Example

$$\begin{aligned}M[1][3] &= \text{Min}\{M[1][1] + M[2][3] + d_0d_1d_3, M[1][2] + M[3][3] + d_0d_2d_3\} \\ &= \text{Min}\{0 + 24 + 40, 30 + 0 + 60\} = 64\end{aligned}$$

$$\begin{aligned}M[2][4] &= \text{Min}\{M[2][2] + M[3][4] + d_1d_2d_4, M[2][3] + M[4][4] + d_1d_3d_4\} \\ &= \text{Min}\{0 + 72 + 36, 24 + 0 + 48\} = 72\end{aligned}$$

$$\begin{aligned}M[1][4] &= \text{Min}\{M[1][1] + M[2][4] + d_0d_1d_4, M[1][2] + M[3][4] + d_0d_2d_4 \\ &M[1][3] + M[4][4] + d_0d_3d_4\} \\ &= \text{Min}\{0 + 72 + 60, 30 + 72 + 90, 64 + 0 + 120\} = 132\end{aligned}$$

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

The algorithm

Problem: Determining the minimum number of elementary multiplications needed to multiply n matrices and an order that produces that minimum number.

```
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
        for (i = 1; i <= n - diagonal; i++) { // just above the
            j = i + diagonal; // main 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];
}
```

Every-case time complexity

- Basic operation: instructions executed for each value of k
- Input size: n , the number of matrices to be multiplied
- Time complexity

$$\sum_{diagonal \neq 1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

To obtain an optimal order from array P

	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

Figure 3.9 • The array P produced when Algorithm 3.6 is applied to the dimensions in Example 3.5.

□ What is the optimal order?

Print optimal order

- Problem: Print the optimal order for multiplying n matrices.

```
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 << ")";
    }
}
```

Optimal Binary Search Trees

□ Binary search tree

Definition

A *binary search tree* is a binary tree of items (ordinarily 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.

Examples

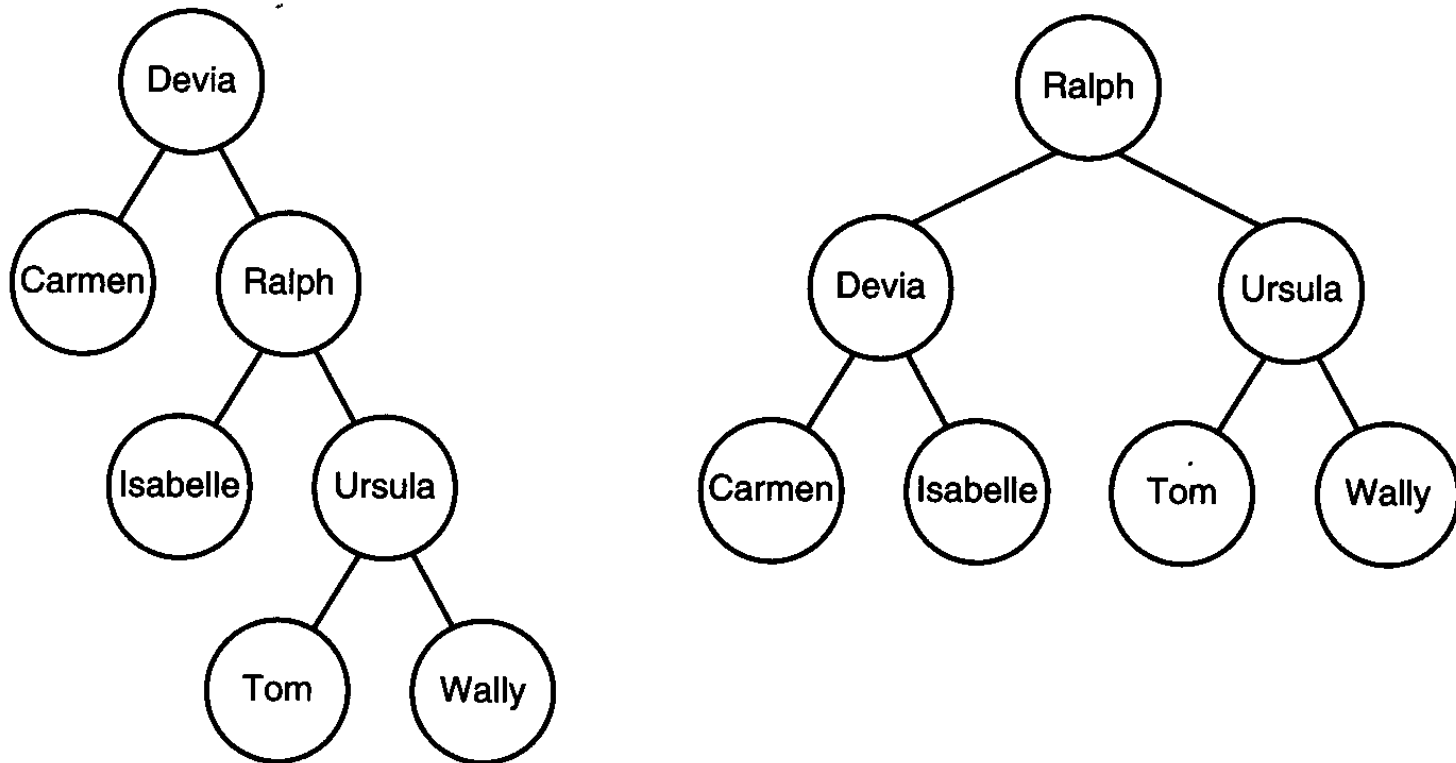


Figure 3.10 • Two binary search trees.

Balanced tree

- Depth (level) of a node: the number of edges in the unique path from the root to the node.
- Depth of a tree: maximum depth of all nodes in the tree.
- Balanced binary tree: the depth of the two subtrees of every node never differ by more than 1.
- Optimal binary search tree: the average time it takes to locate a key is minimized.

Data structure

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

```
typedef nodetype* node_pointer
```

The search algorithm

□ Algorithm 3.8: Search Binary Tree

Problem: Determine the node containing a key in a binary search tree. It is assumed that the key is in the tree.

```
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.  
}
```

The average search time

- Search time: the number of comparisons done to locate a key
- Search time for *key* is: $depth(key) + 1$
- The average search time:

$$\sum_{i=1}^n c_i p_i$$

Where n is the number of keys, p_i the probability that Key_i is the search key, c_i the number of comparisons needed to find Key_i

Find out the average search time

- $P_1 = 0.7,$
 $P_2 = 0.2,$
 $P_3 = 0.1$

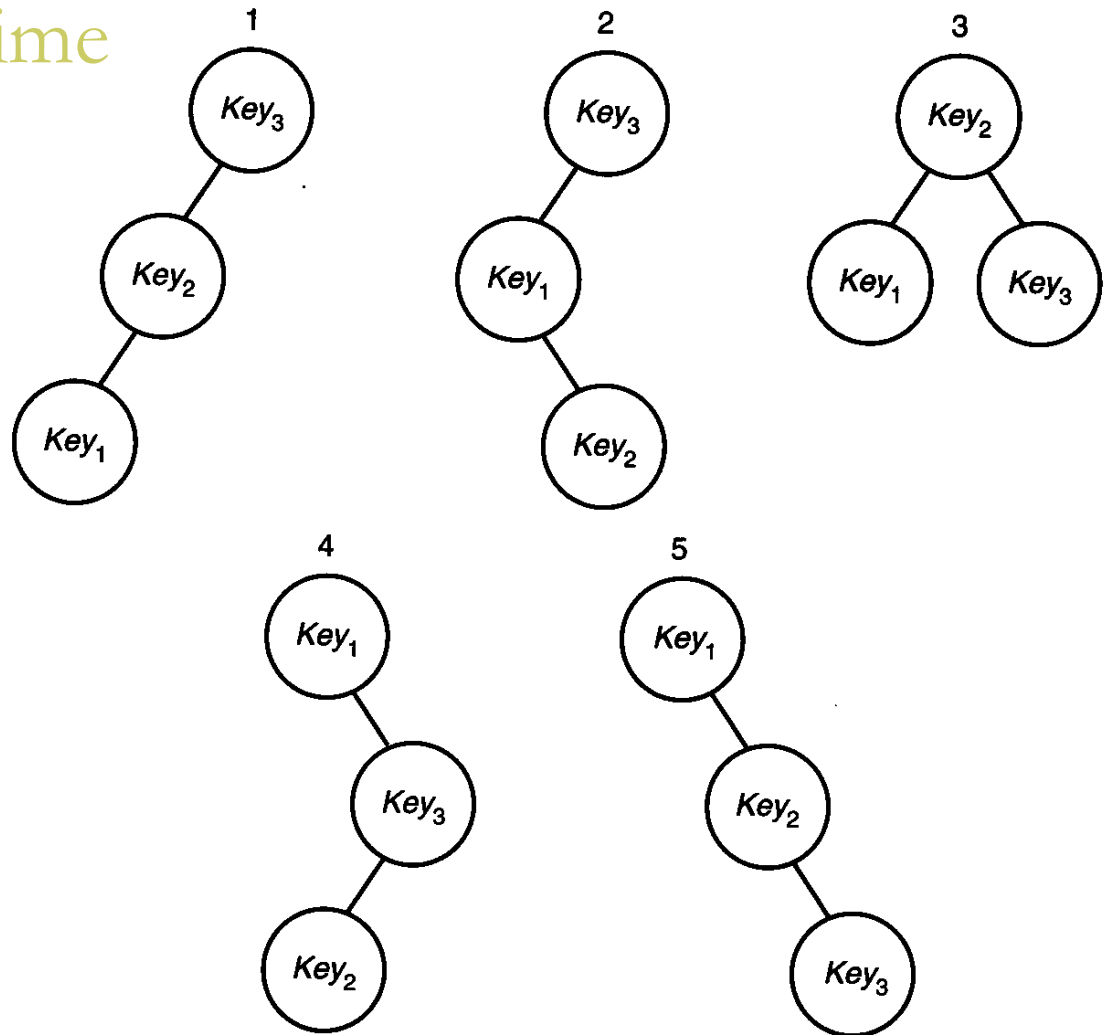


Figure 3.11 • The possible binary search trees when there are three keys.

Brute force search complexity to find optimal solution

- Brute force search is at least exponential (The number of different binary search trees with a depth of $n - 1$ is 2^{n-1})



To develop an efficient algorithm

□ Principle of optimality applies

□ Let $A[i][j]$ = minimum value of $\sum_{m=i}^j c_m p_m$

□ $A[i][i] = p_i$

To search a key in Tree k

- Let Tree k be the optimal tree that Key_k is at the root

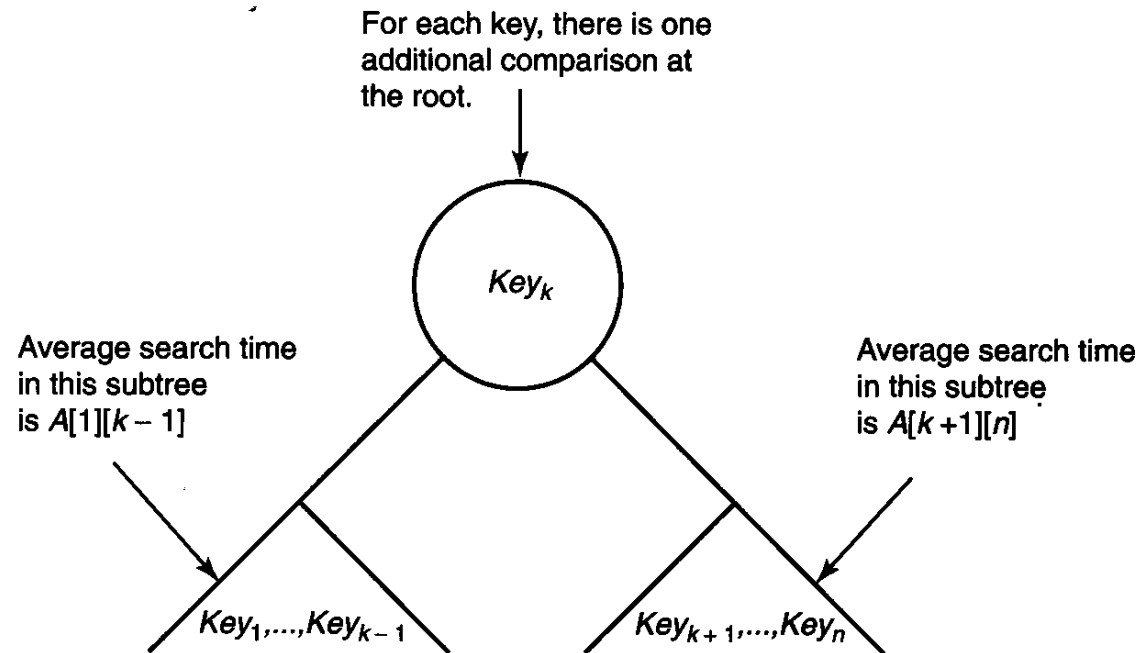


Figure 3.13 • Optimal binary search tree given that Key_k is at the root.

The average search time

$$\underbrace{A[1][k-1]}_{\text{Average time in left subtree}} + \underbrace{p_1 + \dots + p_{k-1}}_{\text{Additional time comparing at root}} + \underbrace{p_k}_{\text{Average time searching for root}} + \underbrace{A[k+1][n]}_{\text{Average time in right subtree}} + \underbrace{p_{k+1} + \dots + p_n}_{\text{Additional time comparing at root}},$$

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



$$A[1][n] = \underset{1 \leq k \leq n}{\text{Minimum}}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^n p_m$$

$$A[i][j] = \underset{i \leq k \leq j}{\text{minimum}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^j p_m \quad i < j$$

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

$A[i][i-1]$ and $A[j+1][j]$ are defined to be 0.

The algorithm(Matrix initialization)

```
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;  
}  
A[n + 1][n] = 0;  
R[n + 1][n] = 0;
```

The algorithm (cont'd)

```
for (diagonal = 1; diagonal <= n - 1; diagonal ++)  
  for (i = 1; i <= n - diagonal; i ++)  
  {  
    j = i + diagonal;  
     $A[i][j] = \text{minimum} (A[i][k - 1] + A[k + 1][j]) + \sum_{m=i}^j P_m$   
     $R[i][j] = \text{a value of } k \text{ that gave the minimum;}$   
     $i \leq k \leq j$   
  }  
minavg  $A[1][n]$ ;
```

Every-case time complexity

- Basic operation: The instructions executed for each value of k
- Input size: n , the number of keys
- Time complexity:

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

Build optimal binary search tree

□ Algorithm 3.10: Build Optimal Binary Search Tree

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*;

 }

}

$$A[1][n] = \underset{1 \leq k \leq n}{\text{Minimum}}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^n p_m$$

An example

- Keys: Don Isabelle Ralph Wally
 Key[1] Key[2] Key[3] Key[4]
 $p_1=3/8$ $p_2=3/8$ $p_3=1/8$ $p_4=1/8$
- Arrays produced

	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.

The resultant tree

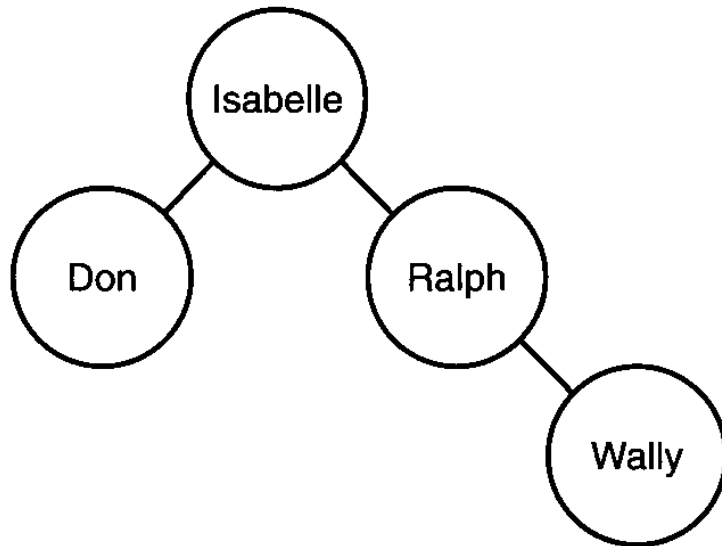


Figure 3.15 • The tree produced when Algorithms 3.9 and 3.10 are applied to the instance in Example 3.9.

The traveling sales person problem

- Tour (Hamilton circuit): a path from a vertex to itself that passes through each of the other vertices exactly once
- Optimal tour: such a path of minimum length
- Brute force algorithm is $(n-1)(n-2)\cdots 1 = (n-1)!$
- Principle of optimality applies

Representation of the graph

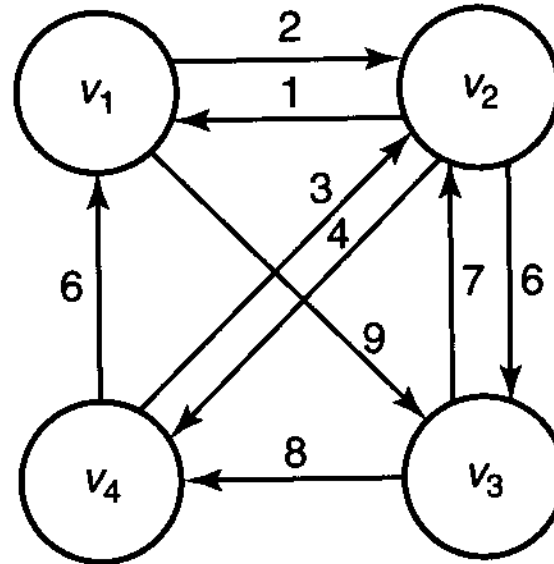


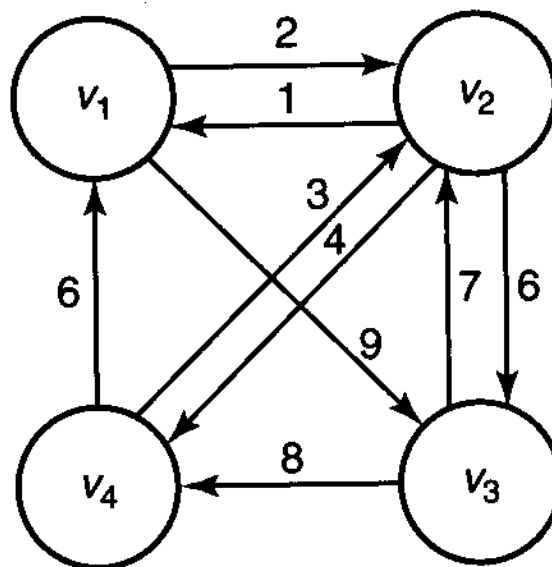
Figure 3.16 • The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Figure 3.17 • The adjacency matrix representation W of the graph in Figure 3.16.

Preparation

- Let $D[v_i][A]$ = length of a shortest path from v_i to v_1 passing through each vertex in A exactly once
- Compute $D[v_2][A]$ when $A = \{v_3\}$ and $A = \{v_3, v_4\}$



The algorithm

- Length of an optimal tour =

$$\mathit{Minimum}_{2 \leq j \leq n} (W[1][j] + D[v_j][V - \{v_1, v_j\}])$$

- In general for $i \neq 1$ and v_i not in A

$$D[v_i][A] = \mathit{Minimum}_{j: v_j \in A} (W[i][j] + D[v_j][A - \{v_j\}]) \quad \text{if } A \neq \phi$$

$$D[v_i][\phi] = W[i][1]$$

Compute the optimal tour

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Figure 3.17 • The adjacency matrix representation W of the graph in Figure 3.16.

The Algorithm

- Algorithm 3.11: The Dynamic Programming Algorithm for the Traveling Salesperson Problem

void *travel* (**int** *n*, **const number** *W*[], **index** *P*[], **number&** *minlength*)

```
{
  index i, j, k;
  number D[1..n][subset of  $V - \{v_1\}$ ];
  for (i = 2; i <= n; i++) D[i][ $\emptyset$ ] = W[i][1];
  for (k = 1; k <= n - 2; k++)
    for (all subsets  $A \in V - \{v_1\}$  containing k vertices)
      for (i such that  $i \neq 1$  and  $v_i$  is not in  $A$ ) {
         $D[i][A] = \text{minimum}_{j: v_j \in A} (W[i][j] + D[j][A - \{v_j\}]);$ 
        P[i][ $A$ ] = value of j that gave the minimum;
      }
   $D[1][V - \{v_1\}] = \text{minimum}_{2 \leq j \leq n} (W[1][j] + D[j][V - \{v_1, v_j\}]);$ 
  P[1][ $V - \{v_1\}$ ] = value of j that gave the minimum;
  minlength = D[1][ $V - \{v_1\}$ ];
}
```

Time complexity

$$\sum_{k=1}^n k \binom{n}{k} = n 2^{n-1}$$

the total number of times the basic operation is done:

$$T(n) = \sum_{k=1}^{n-2} (n-1-k) k \binom{n-1}{k}.$$

$$\left| (n-1-k) \binom{n-1}{k} \right| = (n-1) \binom{n-2}{k}.$$

$$T(n) = (n-1) \sum_{k=1}^{n-2} k \binom{n-2}{k}.$$

Every-case time complexity

- Basic operation: the instructions executed for each value of v_j
- n , the number of vertices in the graph
- Time complexity
$$T(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$
- Memory complexity:
$$M(n) = 2 \times n2^{n-1} = n2^n \in \Theta(n2^n)$$

Exercises

- 4
- 7,9
- 12,14
- 19,26
- 30,34

The End of Chapter 3