Chapter 3:
Dynamic Programming

Contents

Paradigm

- Divide-and conquer
  - Ex) Alg. 1.6 (recursive Alg. for computing *n*th Fibonacci number)
    - Top-down approach
    - Smaller instances are related - compute the same term more than once - # of terms computed: $\theta(2^n)$
  - Ex) mergesort - smaller instances are not related
- Dynamic Programming (DP)
  - Solve smaller instances first
  - Store the results
  - Whenever we need a result, look it up instead of recomputing it.
  - Ex) Alg 1.7(iterative Alg. for computing *n*th Fibonacci term)
    - Bottom-up approach

Steps for DP

- **Establish a recursive property that gives the solution to an instance of the problem**

$$f_n = f_{n-1} + f_{n-2}$$

- **Solve an instance of the problem in a bottom-up fashion by solving smaller instance first.**

$$f_0, \ f_1, \ f_2, \ \cdots, \ f_n$$

3.1 Binomial Coefficient

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

- **Recursive relation**

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

- **(Proof)**

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{(k)!(n-k-1)!}$$

$$= \frac{k(n-1)!}{(k)!(n-k)!} + \frac{(n-k)(n-1)!}{(k)!(n-k)!}$$

$$= \frac{n!}{k!(n-k)!} = \binom{n}{k}$$

Algorithm 3.1 Divide-and-Conquer

- **<u>Problem</u>: Compute the binomial coefficient.**
- **<u>Inputs</u>: nonnegative integer n and k, where $k \le n$.**
- **<u>Outputs</u>: *bin*, the binomial coefficient (n k)**

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

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

Alg. 3.1 D&C (recursion)

- computes $2\binom{n}{k}-1$ terms

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

- (Proof by induction)
  - Induction basis $n=0,\ k=0,\ 2\binom{0}{0}-1=1$
  - Induction hypothesis
    - Assume it is true for m < n
    - Show that it is true for n

$$\underbrace{2\binom{n-1}{k-1}-1}_{bin(n-1,k-1)}+\underbrace{2\binom{n-1}{k}-1}_{bin(n-1,k)}+1=2\binom{n}{k}-1$$

DP for Binomial Coefficient

- **Let** $B[i][j]$ **contain** $\binom{i}{j}$

  1. Establish a recursive property

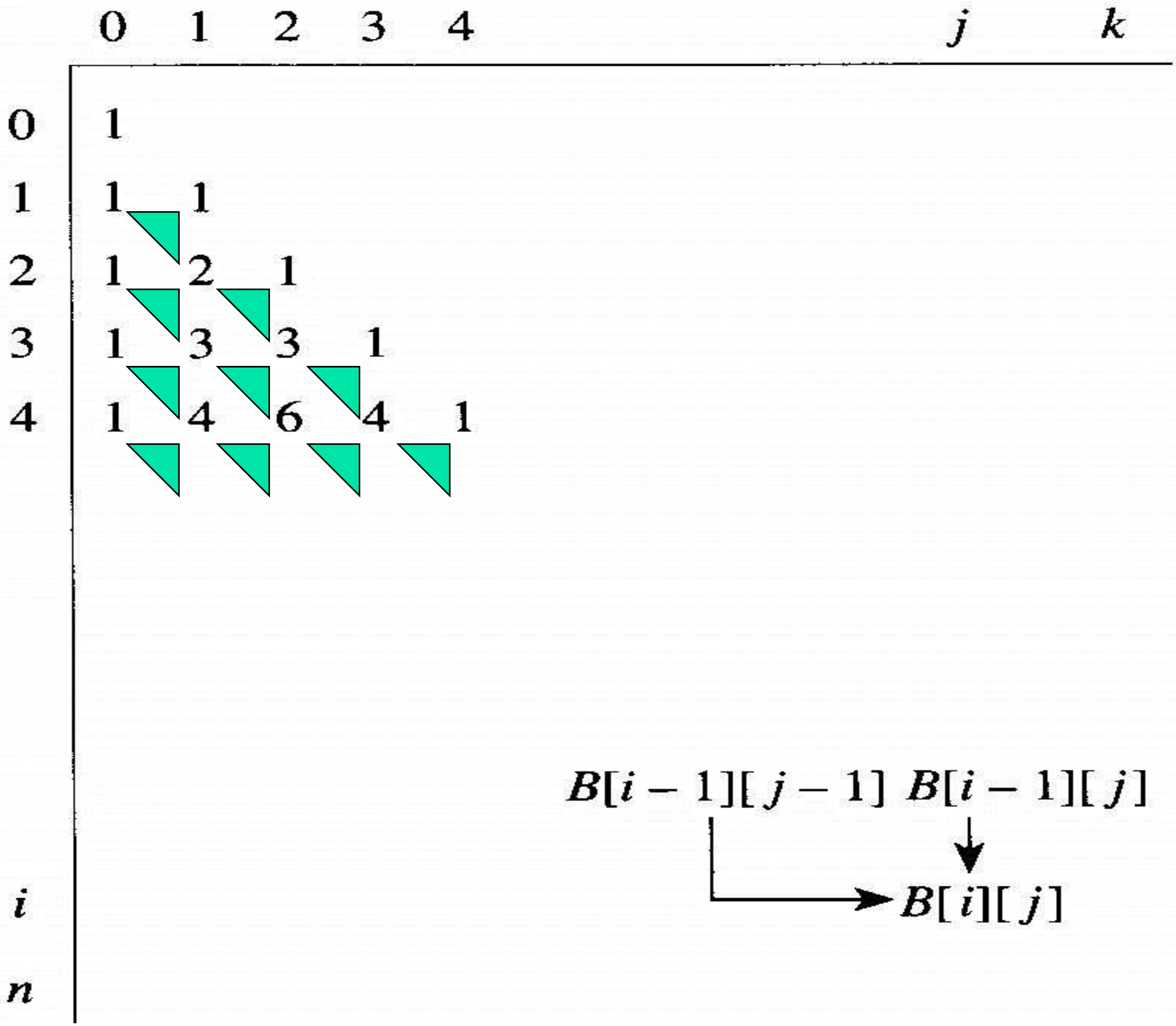
$$\mathrm{B[i][j]} = \begin{cases} \mathrm{B[i-1][j-1]} + \mathrm{B[i-1][j]} & 0 < j < i \\ 1 & j = 0 \text{ or } j = i \end{cases}$$

  2. Solve in bottom-up fashion starting with first row

  - **See Fig. 3.1**

- **Ex) 3.1**

Figure 3.1 The array B used to compute the binomial coefficient.



Algorithm 3.2 Binomial Coefficient Using DP

- **<u>Problem</u>: Compute the binomial coefficient.**
- **<u>Inputs</u>: nonnegative integers *n* and *k*, where $k \le n$.**
- **<u>Outputs</u>: *bin2*, the binomial coefficient (n k)**

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

```
int bin2(int n, int k)
{
   index i, k;
   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];
}
```

Analysis of Algorithm 3.2

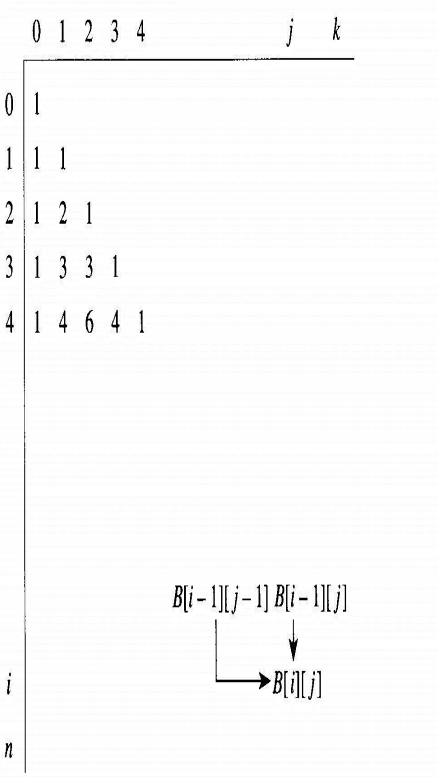
- **Complexity**
  - **Use n and k (note that n and k are not input size)**

i	0	1	$\cdots$	k - 1	k	$\cdots$	n
# of passes	1	2	$\cdots$	k	k + 1	$\cdots$	k + 1
					$\mapsto$	$n-k+1$ times	$\hookleftarrow$

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

- **Improvements**
  - **Rewrite using 1D array indexed from 0 to k. (Exercise 4)**
  - **Take advantage of the fact that** $\binom{n}{k} = \binom{n}{n-k}$

Figure 3.1 The array B used to compute the binomial coefficient.



3.2 Floyd's Algorithm for Shortest Paths

- **Let $G = (V, E)$ be a weighted directed graph**

$$\underset{\text{cost (weight) matrix}}{W[i][j]} = \begin{cases} \text{weight on edge} & \text{if } \langle i, j \rangle \in E \\ \infty & \text{if no edge} \\ 0 & \text{if } i = j \end{cases}$$

vertex (node)
edge (arc)
directed graph (digraph)
weighted graph
weight
path
length
cycle
cyclic
acyclic
simple

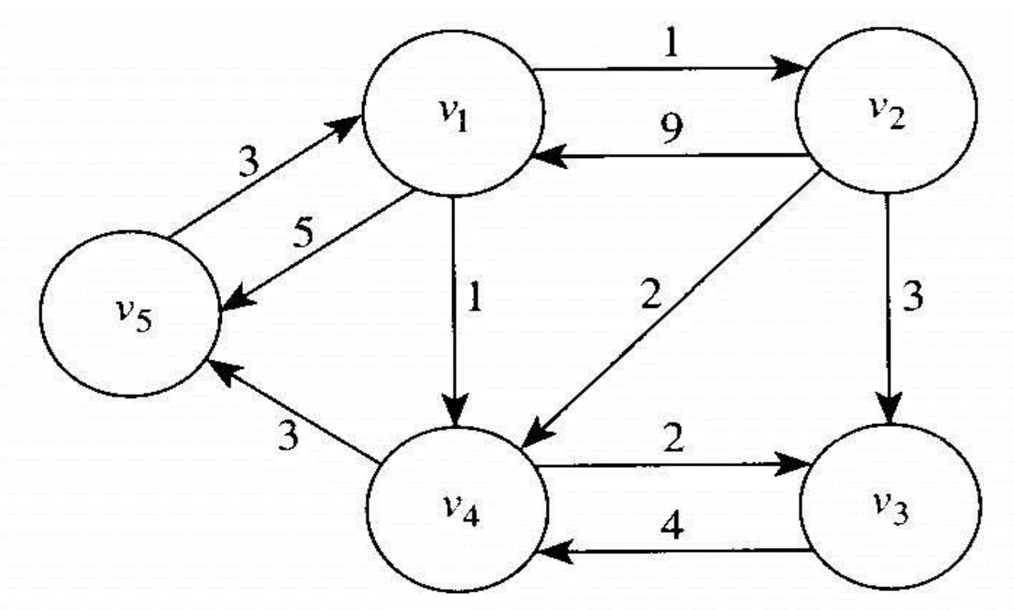


Figure 3.2 A weighted, directed graph.

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.

	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 **(adjacency matrix)**

	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

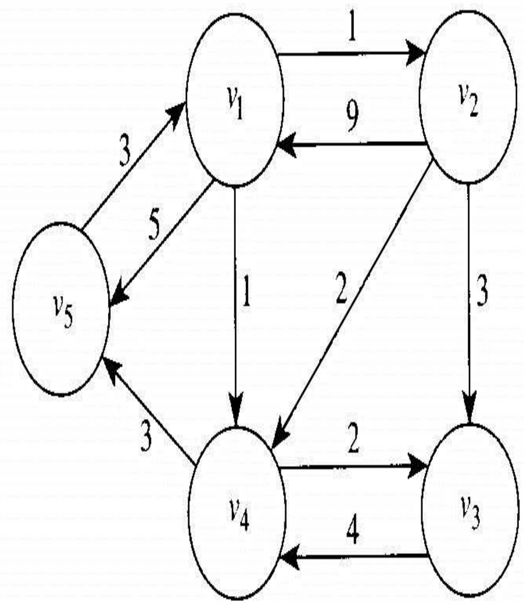


Figure 3.2 A weighted, directed graph.

- **All pairs shortest path problem** is to determine a matrix D such that D(i, j) is the length of a shortest path from vertex i to j.

Basic DP steps for shortest paths

$D^{(k)}[i][j]$: Length of a shortest path from v_i to v_j using only vertices in the set $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices.

$D^{(0)}[i][j] = W[i][j]$ want $D^{(n)}[i][j]$

- Step 1 : Establish a recursive property (process) with which we can compute $D^{(k)}$ from $D^{(k-1)}$
- Step 2 : compute $D^{(0)} = W,\ D^{(1)}, \cdots, D^{(n)} = D$

Example 3.2

- **Calculate some values of** $D^{(k)}[i][j]$

$$
\begin{array}{c|c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}
\end{array}
$$

$$D^{(0)}[2][5] = length\ [v_2, v_5] = \infty$$

$$D^{(1)}[2][5] = \min(\ length\ [v_2, v_5],\ length\ [v_2, v_1, v_5]$$
$$= \min(\infty,\ 14) = 14$$

$$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$$

$$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$$

$$D^{(4)}[2][5] = \min(\ length\ [v_2, v_1, v_5],\ length\ [v_2, v_4, v_5],$$
$$length\ [v_2, v_1, v_4, v_5],\ length\ [v_2, v_3, v_4, v_5])$$
$$= \min(14,\ 5,\ 13,\ 10) = 5$$

$$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$$

General DP processes for shortest paths (1/2)

- **Let $\{v_1, v_2, \ldots, v_k\}$ be intermediate vertices**
- **case 1 : At least one shortest path from v_i to v_j does not use v_k. then** $D^{(k)}[i][j] = D^{(k-1)}[i][j]$
  - **ex.)** $D^{(5)}[1][3] = D^{(4)}[1][3]$ **Shortest path** $v_1 \rightarrow v_4 \rightarrow v_3$

$$
\begin{array}{c}
\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{bmatrix}
0 & 1 & \infty & 1 & 5 \\
9 & 0 & 3 & 2 & \infty \\
\infty & \infty & 0 & 4 & \infty \\
\infty & \infty & 2 & 0 & 3 \\
3 & \infty & \infty & \infty & 0
\end{bmatrix}
\end{array}
$$

General DP processes for shortest paths (2/2)

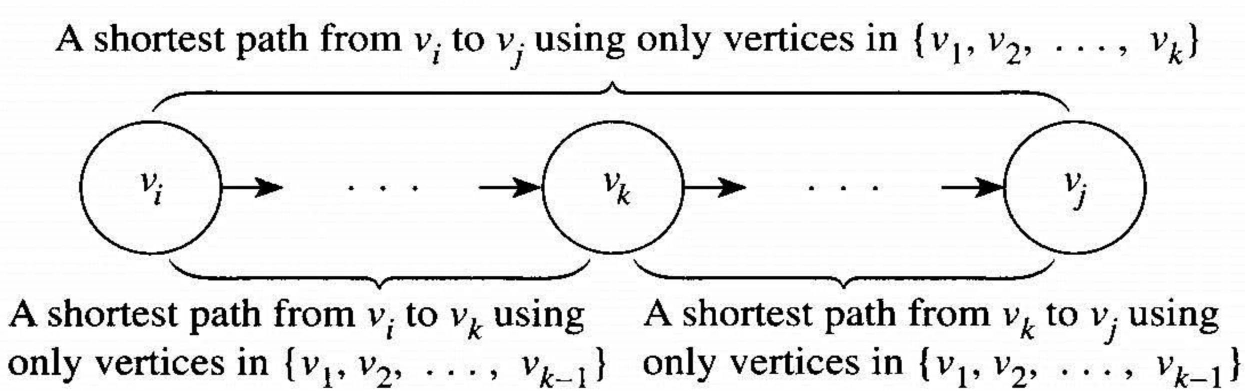
- **case 2 : All shortest path from v_i to v_j use v_k.**

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & & & \\ 2 & & & & & \\ 3 & & & & & \\ 4 & & & & & \\ 5 & & & & & \end{array}$$

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Figure 3.4 The shortest path uses v_k.



- **ex.)** $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$

$$D^{(k)}[i][j] = \min(\underbrace{D^{(k-1)}[i][j]}_{\text{Case1}},\quad \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{Case2}})$$

Example 3.3

$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D^{(1)}[2][4] = \min(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4])$$
$$= \min(2,\ 9+1) = 2$$

$$D^{(1)}[5][2] = \min(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2])$$
$$= \min(\infty, 3+1) = 4$$

$$D^{(1)}[5][4] = \min(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4])$$
$$= \min(\infty, 3+1) = 4$$

$$D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4])$$
$$= \min(4, 4+2) = 4$$

Space requirements for shortest paths

- **We need to use only one array D**

  $D^{(k)}[i][j]$ **is computed from only its value and values in the *k*-th row and the *k*-th column**

$$D^{(k)}[i][k] = \min(D^{(k-1)}[i][k],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][k]) = D^{(k-1)}[i][k]$$

Similarly $D^{(k)}[k][j] = D^{(k-1)}[k][j]$

- **At the *k*-th iteration, the value in the *k*-th row and the *k*-th column are not changed → No need for extra array**

Algorithm 3.3 Floyd's Algorithm (1/2)

- **Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.**
- **Inputs: A weighted, directed graph and *n*, the number of vertices in the graph. The graph is represented by a 2D array *W*, which has both its rows and columns indexed from 1 to *n*, where *W*[*i*][*j*] is the weight on the edge from the *i*-th vertex to the *j*-th vertex.**
- **Outputs: A 2D 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.**

Algorithm 3.3 Floyd's Algorithm (2/2)

$$D^{(k)}[i][j] = \min(\underbrace{D^{(k-1)}[i][j]}_{\text{Case 1}},\ \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{Case 2}})$$

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

- **T(n) ?**
  - **Basic operation: instruction in the for-j loop**
  - **Input size: $n = |V|$**

  $T(n) = n \times n \times n \in \Theta(n^3)$

Algorithm 3.4 Floyd's Algorithm (1/2)

- **Problem: Same as in Alg. 3.3 except shortest paths are also created.**
- **Additional Outputs: an array *P*, which has both its rows and columns indexed from 1 to *n*, where**

$$P[i][j] = \begin{cases} \text{highest index of an intermediate vertex} \\ \text{on the shortest path from } v_i \text{ to } v_j \\ \text{if at least one intermediate vertex exists.} \\ \\ 0 \text{ if no intermediate vertex exists.} \end{cases}$$

Algorithm 3.4 Floyd's Algorithm (2/2)

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

Algorithm 3.5 Print Shortest Path (1/2)

- **Problem: Print the intermediate vertices on a shortest path from one vertex to another vertex in a weighted graph.**
- **Inputs: the array *P* produced by Alg. 3.4, and two indices, *q* and *r*, of vertices in the graph that is the input to Alg. 3.4.**
- **Outputs: the intermediate vertices on a shortest path from v_q to v_r.**

Algorithm 3.5 (1/2)

- **T(n) of Algorithm 3.5**

  $W(n) = \Theta(n)$

- v_5 **to** v_3 $\qquad v_5 \rightarrow v_1 \rightarrow v_4 \rightarrow v_3$

	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

```
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);
   }
}
```

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