

Unit 3

Dynamic programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem. We typically apply dynamic programming to **optimization problems**. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Elements of dynamic programming –

1. Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems. Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.

Common pattern in discovering optimal substructure:

- ✓ You show that a solution to the problem consists of making a choice.
- ✓ You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
- ✓ Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
- ✓ You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a “cut-and-

paste” technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

2. Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has **overlapping subproblems**. In contrast, a problem for which a divide-and conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

3. Reconstructing an optimal solution

As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.

4. Memoization

The idea is to **memoize** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm. A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table. Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.

6

★ Multistage Graph :-

A multistage graph $G(V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint set V_i where $1 \leq i \leq k$.

If (u, v) is an edge in E then $u \in V_i$ and $v \in V_{i+1}$ for same $1 \leq i \leq k$.

The sets V_1 & V_k are such that $V_1 = V_k = 1$.

Let 's' & 't' be the vertices in V_1 & V_k such that $s \rightarrow$ source $t \rightarrow$ destination.

The cost of a path from 's' to 't' is the sum cost of the edges on path.

The multistage graph problem is to find a minimum cost path from $s \rightarrow t$

Each set V_i defines a stage in a graph. Because of the constraints on E , every path from s to t starts in stage 1 & goes to stage 2 --- terminate at k .

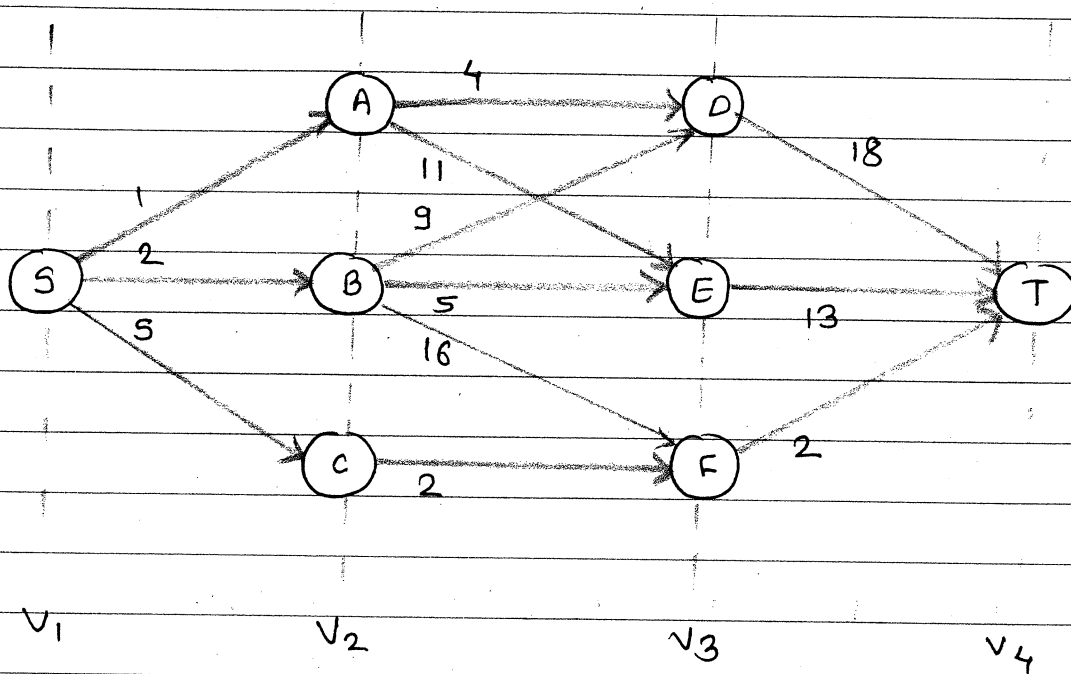
Initially divide graph into stages then find out cost of each node.

$$\text{cost}(i, j) = \min \{ C(j, l) + \text{cost}(i+1, l) \}$$

while solving you have to start from V_k stage and then find out cost for (8) & (7) vertices and continue procedure in V_3, V_2, V_1 .

7

Q. calculate the minimum cost path from source to destination in multistage graph using dynamic strategy.



There are total 4 stages in graph.
Start from $n-1$ stage i.e. $4-1 = 3$

cost (stage, vertex)

$$\therefore \text{cost}(3, D) = 18$$

$$\text{cost}(3, E) = 13$$

$$\text{cost}(3, F) = 2$$

Jump to previous stage. i.e. 2

$$\begin{aligned} \text{cost}(2, A) &= \min \{ 4 + \text{cost}(3, D), 11 + \text{cost}(3, E) \} \\ &= \min \{ 4 + 18, 11 + 13 \} \end{aligned}$$

9

Date / /

Page

bilt 10 on 10
Student Notebooks

$$\text{cost}(2, B) = \min \{ 9 + c(3, D) \mid \\ 5 + c(3, E) \mid \\ 16 + c(3, F) \}$$

$$= \min \{ 9 + 18, 5 + 13, 16 + 2 \}$$

$$= \min \{ 27, 18, 18 \}$$

$$= 18$$

$$\text{cost}(2, C) = \min \{ 2 + \text{cost}(3, F) \}$$

$$= \min \{ 2 + 2 \}$$

$$= 4$$

Stage ~~2~~ 1 i.e. V_1

$$\text{cost}(1, S) = \min \{ 1 + \text{cost}(2, A) \mid$$

$$2 + \text{cost}(2, B) \mid$$

$$5 + \text{cost}(2, C) \}$$

$$= \min \{ 1 + 22, 2 + 18, 5 + 4 \}$$

$$= \min \{ 23, 20, 9 \}$$

$$= 9$$

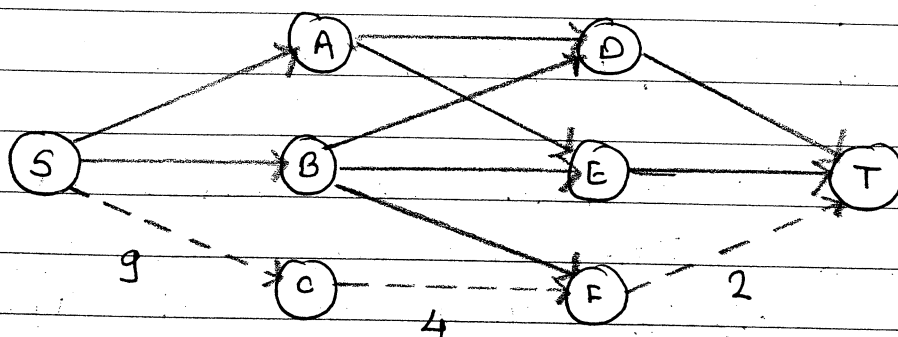
We have to find out the path from source to destination node i.e. from S to t.
Start from node 'S'.

From node 'S' cost of 'C' is minimum
hence path is $S \rightarrow C$.

From node 'C' there is only one path to F
 \therefore path is $S \rightarrow C \rightarrow F$. & from node F to T

\therefore path is finally $S \rightarrow C \rightarrow F \rightarrow T$.

9



* Knapsack Problem :-

There are n objects & a knapsack or bag, object i has weight w_i and profit P_i ; also knapsack has capacity M .

If object i is placed into knapsack then we get profit P_i .

The objective is to obtain a filling of the knapsack with capacity M we required the total weight of all chosen objects to be at most m .

A solution to the knapsack problem can be obtained by making sequence of decision on variables x_1, x_2, \dots, x_n .

Decision on variable x_i involves determining which of the values 0 or 1.

We have to compute decisions

initially $S^0 = \{(0,0)\}$ we can compute $S_1^i = \{(P, w) \mid (P - P_i, w - w_i) \in S^i\}$

Now S^{i+1} can be computed by merging the pairs in S^i & S_1^i together. Note if:

10

Purging rule / Dominance Rule →

If S^{i+1} contains two pairs (P_j, w_j) & (P_k, w_k) with property that $P_j \leq P_k$ and $w_j \geq w_k$, then the pair (P_j, w_j) can be discarded. i.e. (P_k, w_k) dominates (P_j, w_j) .

Note - We can also purge all pairs (P, w) with $w \geq m$ since knapsack capacity is m and w is weight of objects.

Q. Consider a knapsack instance $M=8$ & $n=4$. let P_i & w_i are as follow.

i	P_i	w_i	objects
1	1	2	x_1
2	2	3	x_2
3	5	4	x_3
4	6	5	x_4

→ Let build sequence of decisions S^0, S^1, S^2, S^3, S^4

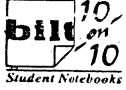
Initially $S^0 = \{(0, 0)\}$ bag is empty

To build S^1 , select next pair and add it into S^0 i.e. pair $(1, 2)$

$$(P_1 \dots P_6) = (100, 50, 20, 70, 7, 3) \quad M = 165$$

$$(W_1 \dots W_7) = (\dots)$$

110101

Date	/ /	
Page		

$$S_0 = \{(0+1, 0+2)\}$$

$$= \{(1, 2)\}$$

Now build S^1 by merging S^0 & S^0 ,

$$S^1 = \{(0, 0), (1, 2)\}$$

S^1 select next pair i.e. (2, 3), add to S^1

$$S^1 = \{(2+0, 3+0), (2+1, 3+2)\}$$

$$= \{(2, 3), (3, 5)\}$$

Now,

$$S^2 = \text{merge } S^1 \& S^1$$

$$S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}$$

apply purging rule

$S^2_1 =$ Next pair (5, 4), add to S^2

$$= \{(5, 4)\}$$

$$S^2 = \{(5, 4), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

Now, $S^3 = \text{merge } S^2 \& S^2$,

$$S^3 = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

apply purging rule.

after that (3, 5) pair is purged / discarded

$$S^3 = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

12

$$S_1^3 = \text{next pair } (6, 5) + S^3$$

$$= \{(6, 5), (7, 7), (8, 8), (11, 9), (12, 11), (14, 12), (14, 14)\}$$

$$S^4 = \text{merge } S^3 \text{ \& } S^3,$$

$$= \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 5), (7, 7), (8, 8), (6, 5), (8, 8), (11, 9), (12, 11), (13, 12), (14, 14)\}$$

apply purging rule

$$= \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 5), (8, 8)\}$$

Knapsack capacity is given as $M = 8$

Now find out pair whose weight i.e.

$w = 8$, (start searching from S^0, S^1, \dots, S^4)

Pair is $(8, 8)$ which is found in S^4

$$\therefore X_4 = 1$$

Now select next object by

$$(P - P_4) \text{ \& } (W - W_4)$$

$$(P, W) = (8, 8)$$

$$\therefore (8 - 6) \text{ \& } (8 - 5)$$

$$(P_4, W_4) = (6, 5)$$

$$\therefore (2, 3)$$

(because $S^4 \text{ \& } X_5 = 1$)
(4th pair)

Now find out $(2, 3)$ through S^0, S^1, \dots, S^4

It is found in S^2 hence $X_2 = 1$

Again select next object.

13

Note - arrange pairs in ascending order ~~1~~ while merging to ~~for better~~ reduce time while purging

$$(P_1, P_4) \& (W_1, W_4)$$

$$(2-2) \& (3-3)$$

$$(0, 0)$$

$$(P_1, W_1) = (2, 3)$$

$$(P_2, W_2) = (2, 3)$$

(2nd pair)

Hence we have to add x_2 & x_4 to bag.

Hence final solution is $(0, \overset{x_1 x_2 x_3 x_4}{1}, 0, 1)$

(Note - ~~th~~ after getting answer cross verify answer manually).

0/1 Knapsack Problem –

Now, let us try to apply the greedy method to solve a more complex problem. This problem is the knapsack problem. We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity M . The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is M , we require the total weight of all chosen objects to be at most M . Formally, the problem may be stated as:

$$\begin{aligned} &\text{maximize } \sum_{1 \leq i \leq n} p_i x_i \\ &\text{subject to } \sum_{1 \leq i \leq n} w_i x_i \leq M \\ &\text{and } 0 \leq x_i \leq 1, \quad 1 \leq i \leq n \end{aligned}$$

The profits and weights are positive numbers.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack. Hence, in case of 0-1 Knapsack, the value of x_i can be either **0** or **1**, where other constraints remain the same. 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

Analyze the 0/1 Knapsack Problem

When analyzing 0/1 Knapsack problem using Dynamic programming, you can find some noticeable points. The value of the knapsack algorithm depends on two factors: (1)How many objects are being considered (2)The remaining weight which the knapsack can store. Therefore, you have two variable quantities.

With dynamic programming, you have useful information:

1. the objective function will depend on two variable quantities
2. the table of options will be a 2-dimensional table.

If calling $B[i][j]$ is the maximum possible value by selecting in objects $\{1, 2, \dots, i\}$ with weight limit j .

- The maximum value when selected in n objects with the weight limit M is $B[n][M]$. In other words: When there are i objects to choose, $B[i][j]$ is the optimal weight when the maximum weight of the knapsack is j .
- The optimal weight is always less than or equal to the maximum weight: $B[i][j] \leq j$.

Time Complexity-

- Each entry of the table requires constant time $\theta(1)$ for its computation.
- It takes $\theta(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $\theta(n)$ time for tracing the solution since tracing process traces the n rows.
- Thus, overall $\theta(nw)$ time is taken to solve 0/1 knapsack problem using dynamic programming.

knapsack problem:-

Solve the 0/1 knapsack problem using dynamic programming

$$W = (3, 4, 6, 5) \quad P = (2, 3, 1, 4) \quad M = 8$$

→

$$(X_1, X_2, X_3, X_4) = (3, 4, 6, 5), (2, 3, 1, 4)$$

			\vec{w}	0	1	2	3	4	5	6	7	8
Obj.	P	W	$\downarrow i$	0	0	0	0	0	0	0	0	0
X_1	2	3	1	0	0	0	2	2	2	2	2	2
X_2	3	4	2	0	0	0	2	3	3	3	5	5
X_4	4	5	3	0	0	0	2	3	4	4	5	6
X_3	1	6	4	0	0	0	2	3	4	4	5	6

$$M[i, w] = \max [M[i-1, w] / M[i-1, w-w[i]] + P[i]]$$

$$\begin{aligned} M[1, 3] &= \max [M[0, 3] / M[0, 3-3] + P[1]] \\ &= \max [M[0, 3] / M[0, 0] + P[1]] \\ &= \max [0 / 0 + 2] \end{aligned}$$

$$M[1, 3] = 2$$

similarly we find all remaining values of table and we get following solution

$$(X_1, X_2, X_3, X_4) = (1, 0, 0, 1)$$

Longest Common Sub-sequence –

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a *subsequence* of X if there exists a strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y , we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y . For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y . The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y , has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y , as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the *longest-common-subsequence problem*, we are given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y . This section shows how to efficiently solve the LCS problem using dynamic programming.

LCS-LENGTH(X, Y)

```
1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i - 1, j - 1] + 1
12             b[i, j] = "↖"
13         elseif c[i - 1, j] >= c[i, j - 1]
14             c[i, j] = c[i - 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j - 1]
17             b[i, j] = "←"
18  return c and b
```

Example – Find out longest common subsequence for

$X = \{ A, B, C, B, D, A, B \}$

$Y = \{ B, D, C, A, B, A \}$

Solution – B C B A

		j	0	1	2	3	4	5	6
			y_j						
				B	D	C	A	B	A
i	x_i								
0			0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖1	←1	↖1
2	B		0	↖1	←1	←1	↑1	↖2	←2
3	C		0	↑1	↑1	↖2	←2	↑2	↑2
4	B		0	↖1	↑1	↑2	↑2	↖3	←3
5	D		0	↑1	↖2	↑2	↑2	↖3	↑3
6	A		0	↑1	↑2	↑2	↖3	↑3	↖4
7	B		0	↖1	↑2	↑2	↑3	↖4	↑4

Shortest Path Algorithms –

- Single Source Shortest Path Algorithm – Bellman Ford Algorithm

The *Bellman-Ford algorithm* solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```

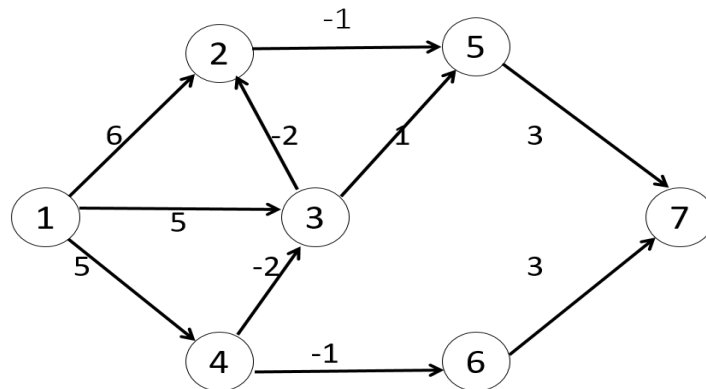
BELLMAN-FORD( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3      for each edge  $(u, v) \in G.E$ 
4          RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6      if  $v.d > u.d + w(u, v)$ 
7          return FALSE
8  return TRUE

```

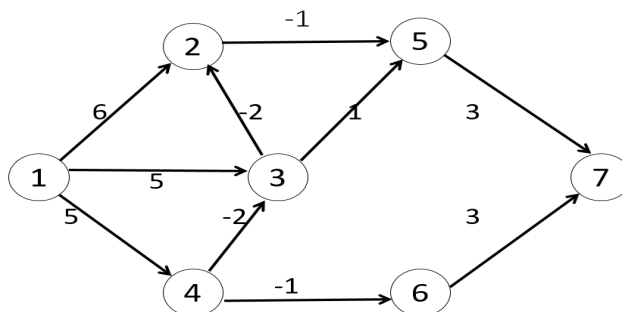
After initializing the d and π values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph. Each pass is one iteration of the for loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures (b)–(e) show the state of the algorithm after each of the four passes over the edges. After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value.

The Bellman-Ford algorithm runs in time $O(VE)$ since the initialization in line 1 takes $\theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes time $\theta(E)$, and the for loop of lines 5–7 takes $O(E)$ time.

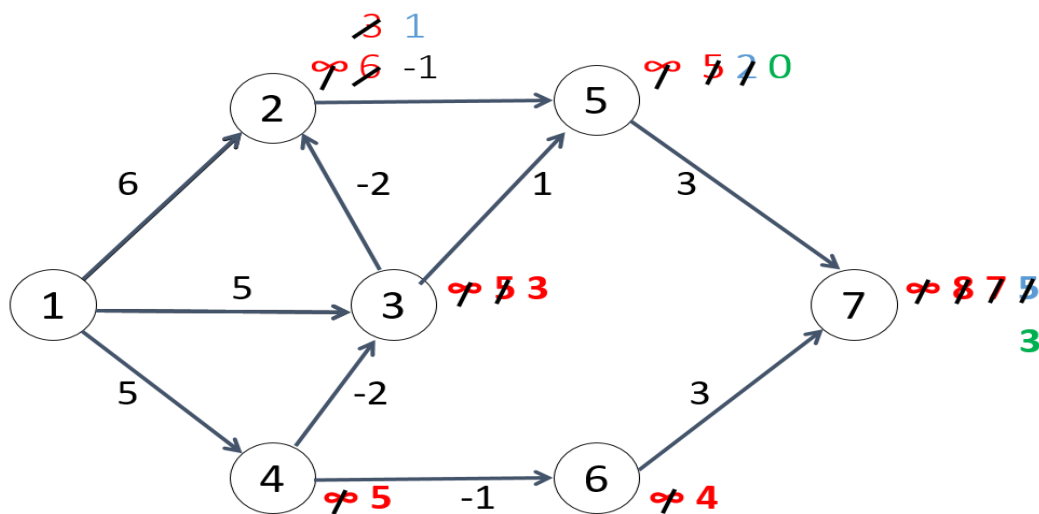
Example –



Solution –



Vertices →	1	2	3	4	5	6	7
Iterations ↓							
1	0	3	3	5	5	4	7
2	0	1	3	5	2	4	5
3	0	1	3	5	0	4	3
4	0	1	3	5	0	4	3
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3



All Pairs Shortest Path Algorithm –

Floyd Warshal Algorithm –

We shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G (V, E)$ The resulting algorithm, known as the **Floyd-Warshall algorithm**, runs in $\theta (V^3)$ time. As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles.

Algorithm –

FLOYD-WARSHALL(W)

```

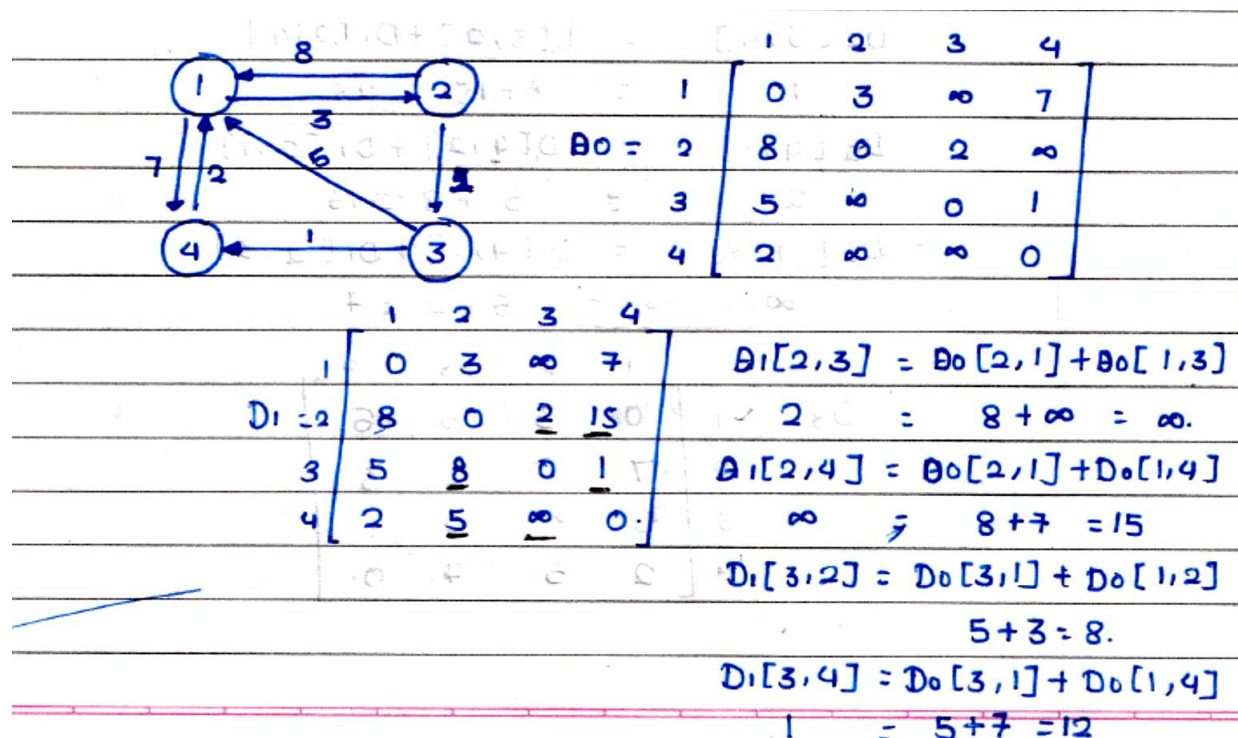
1   $n = W.rows$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5      for  $i = 1$  to  $n$ 
6          for  $j = 1$  to  $n$ 
7               $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 

```

Analysis of the Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm is easy to analyze. The main loop is executed n times and the inner loop considers each of $O(n^2)$ pairs of vertices, performing a constant-time computation for each pair. If we use a data structure, such as the adjacency matrix structure, that supports methods are Adjacent and insertDirectedEdge in $O(1)$ time, we have that the total running time is $O(n^3)$.

Example -



$$D_1[4,2] = D_0[4,1] + D_0[1,2]$$

$$\infty = 2 + 3 = 5$$

$$D_1[4,3] = D_0[4,1] + D_0[1,3]$$

$$\infty = 2 + \infty = \infty$$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

$$D_2[1,3] = D_1[1,2] + D_1[2,3]$$

$$\infty = 3 + 2 = 5$$

$$D_2[1,4] = D_1[1,2] + D_1[2,4]$$

$$7 = 3 + 15 = 18$$

$$D_2[1,3] = D_1[1,2] + D_1[2,3]$$

$$5 = 3 + 2 = 5$$

$$D_2[3,1] = D_1[3,2] + D_1[2,1]$$

$$5 = 8 + 8 = 16$$

$$D_2[3,4] = D_1[3,2] + D_1[2,4]$$

$$1 = 8 + 15 = 23$$

$$D_2[4,1] = D_1[4,2] + D_1[2,1]$$

$$2 = 5 + 8 = 13$$

$$D_2[4,3] = D_1[4,2] + D_1[2,3]$$

$$\infty = 5 + 2 = 7$$

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

$$D_3(1,2) = D_2[1,3] + D_2[3,2]$$

$$3 = 5 + 8 = 13$$

$$D_3(1,4) = D_2[1,3] + D_2[3,4]$$

$$7 = 5 + 1 = 6$$

$$D_3(2,1) = D_2[2,3] + D_2[3,1]$$

$$8 = 2 + 5 = 7$$

$$D_3(2,4) = D_2[2,3] + D_2[3,4]$$

$$15 = 2 + 1 = 3$$

$$D_3(4,1) = D_2[4,3] + D_2[3,1]$$

$$2 = 7 + 5 = 12$$

$$D_3(4,2) = D_2[4,3] + D_2[3,2]$$

$$5 = 7 + 8 = 15$$

$$D_4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{matrix}$$

$$D_4(1,2) = D_3[1,4] + D_3[4,2]$$

$$3 = 6 + 5 = 11$$

$$D_4(1,3) = D_3[1,4] + D_3[4,3]$$

$$5 = 6 + 7 = 13$$

$$D_4(2,1) = D_3[2,4] + D_3[4,1]$$

$$7 = 3 + 2 = 5$$

$$D_4(2,3) = D_3[2,4] + D_3[4,3]$$

$$2 = 3 + 7 = 10$$

$$D_4(3,1) = D_3[3,4] + D_3[4,1]$$

$$5 = 1 + 2 = 3$$

$$D_4(3,2) = D_3[3,4] + D_3[4,2]$$

$$8 = 1 + 5 = 6$$