9 Dynamic Programming

9.1 Introduction

Dynamic programming is a method in which the solution to a problem is obtained by making sequence of decisions. The optimal solution to the given problem is obtained from the sequence of all possible solutions being generated. Dynamic programming is similar to divide and conquer. In divide and conquer the subproblems are solved independantly but in dynamic programming the subproblems share the solutions among themselves. The dynamic programming works on **principle of optimality** to solve the problems. We will discuss the **concept of dynamic programming**, then we will discuss how dynamic programming approach is used in **Warshall's** and **Floyd's algorithm**. In the later part of this chapter we will learn the concept of static trees i.e. optimal binary search trees.

9.2 Dynamic Programming

In this section, we will understand, *"What is dynamic programming ?"* Dynamic programming is typically applied to optimization problem. This technique is invented by a U.S. Mathematician Richard Bellman in 1950. In the word dynamic **programming** the word programming stands for **planning** and it does not mean by computer programming.

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- In this method each subproblem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem.

General method

Dynamic programming is typically applied to optimization problems.

For a given problem we may get any number of solutions. From all those solutions we seek for optimum solution (minimum value or maximum value solution). And such an optimal solution becomes the solution to the given problem.

9.2.1 Difference between Divide and Conquer and Dynamic Programming

Sr. No.	Divide and Conquer	Dynamic Programming
1.	The problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of sub-problems are collected together to get the solution to the given problem.	In dynamic programming many decision sequences are generated and all the overlapping sub-instances are considered.
2.	In this method duplications in sub-solutions are neglected. i.e., duplicate subsolutions may be obtained.	In dynamic computing duplications in solutions is avoided totally.
3.	Divide and conquer is less efficient because of rework on solutions.	Dynamic programming is efficient than divide and conquer strategy.
4.	The divide and conquer uses top down approach of problem solving (recursive methods).	Dynamic programming uses bottom up approach of problem solving (iterative method).
5.	Divide and conquer splits its input at specific deterministic points usually in the middle.	Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points it determines which split point is optimal.

9.2.2 Problems that can be Solved using Dynamic Programming

Various problems those can be solved using dynamic programming are,

i) For computing n^{th} Fibonacci number.

ii) Computing binomial coefficient.

iii) Warshall's algorithm.

iv) Floyd's algorithm.

v) Optimal binary search trees.

Let us discuss these problems one by one.

.3 Fibonacci Number

When it is not possible to apply principle of optimality, it is almost impossible to obtain the solution using dynamic programming approach.

For example : While constructing optimal binary search tree we always select the value of k which is obtained from minimum cost. Thus it follows principle of optimality.

9.8 Knapsack Problem and Memory Function

As we know that dynamic programming is a technique for solving problems with overlapping subproblems. These subproblems are typically based on recurrence relation. In this section we will discuss the method of solving **knapsack problem** using dynamic programming approach. The knapsack problem can be defined as follows : If there are n items with the weights $w_1, w_2, \ldots w_n$ and values (profit associated with each item) $v_1, v_2, \ldots v_n$ and capacity of knapsack to be W, then find the most valuable subset of the items that fit into the knapsack.

To solve this problem using dynamic programming we will write the recurrence relation as :

$$\text{table}[i, j] = \text{maximum}\left\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\right\} \text{ if } j \geq w_i$$

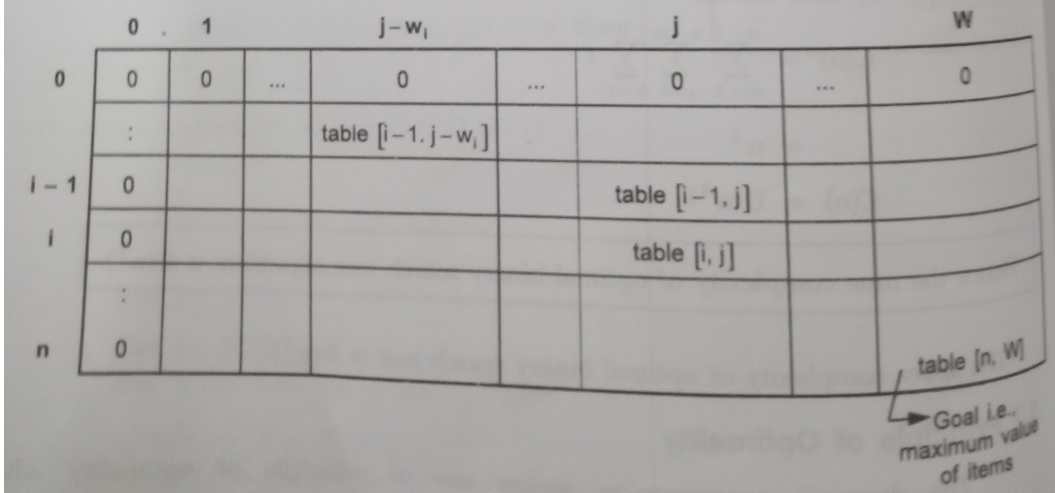
or

$$\text{table}[i-1, j] \text{ if } j < w_i$$

That means, a table is constructed using above given formula. Initially,

table [0, j] = 0 as well as table [i, 0] = 0 when $j \geq 0$ and $i \geq 0$.

The initial stage of the table can be



The table [n, W] is a goal i.e., its gives the total items sum of all the selected items for the knapsack.

From this goal value the selected items can be traced out. Let us solve the knapsack problem using the above mentioned formula -

➡ **Example 9.2 :** *For the given instance of problem obtain the optimal solution for the knapsack problem.*

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

The capacity of knapsack is W = 5.

Solution : Initially, table [0, j] = 0 and table [i, 0] = 0. There are 0 to n rows and 0 to W columns in the table.

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

Now we will fill up the table either row by row or column by column. Let us start filling the table row by row using following formula :

$$\text{table}[i, j] = \begin{cases} \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i] \text{ when } j \geq w_i\} \\ \text{or} \\ \text{table}[i-1, j] \text{ if } j < w_i \end{cases}$$

table [1, 1] With i = 1, j = 1, $w_i = 2$ and $v_i = 3$.

As $j < w_i$ we will obtain table [1, 1] as

table [1, 1] = table [i – 1, j]

= table [0, 1]

$\therefore$ table [1, 1] = 0

table [1, 2] With i = 1, j = 2, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 2] as

table [1, 2] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum $\{(table[0, 2]), (3 + table[0, 0])\}$

= maximum {0, 3 + 0}

$\therefore$ table [1, 2] = 3

table [1, 3] With i = 1, j = 3, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 3] as

table [1, 3] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum $\{table\ [0, 3], 3 + table\ [0, 1]\}$

= maximum {0, 3 + 0}

$\therefore$ table [1, 3] = 3

table [1, 4] With i = 1, j = 4, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$, we will obtain table [1, 4] as

table [1, 4] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum $\{table[0, 4], 3 + table[0, 2]\}$

= maximum {0, 3 + 0}

$\therefore$ table [1, 4] = 3

table [1, 5] With i = 1, j = 5, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 5] as

$$\text{table}[1, 5] = \text{maximum}\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$
$$= \text{maximum}\{\text{table}[0, 5], 3 + \text{table}[0, 3]\}$$
$$= \text{maximum}\{0, 3 + 0\}$$

$\therefore$ table [1, 5] = 3

The table with these values can be

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

Now let us fill up next row of the table.

table [2, 1] With i = 2, j = 1, $w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 1] as

$$\text{table}[2, 1] = \text{table}[i-1, j]$$
$$= \text{table}[1, 1]$$

$\therefore$ table [2, 1] = 0

table [2, 2] With i = 2, j = 2, $w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 2] as

$$\text{table}[2, 2] = \text{table}[i-1, j]$$
$$= \text{table}[1, 2]$$

$\therefore$ table [2, 2] = 3

table [2, 3] With $i = 2$, $j = 3$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 3] as

$$\text{table [2, 3]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum } \{\text{table [1, 3]}, 4 + \text{table [1, 0]}\}$$

$$= \text{maximum } \{3, 4 + 0\}$$

$\therefore$ table [2, 3] = 4

table [2, 4] With $i = 2$, $j = 4$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 4] as

$$\text{table [2, 4]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum } \{\text{table [1, 4]}, 4 + \text{table [1, 1]}\}$$

$$= \text{maximum } \{3, 4 + 0\}$$

$\therefore$ table [2, 4] = 4

table [2, 5] With $i = 2$, $j = 5$, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 5] as

$$\text{table [2, 5]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum } \{\text{table [1, 5]}, 4 + \text{table [1, 2]}\}$$

$$= \text{maximum } \{3, 4 + 3\}$$

$\therefore$ table [2, 5] = 7

The table with these computed values will be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

table [3, 1] With $i = 3$, $j = 1$, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 1] as

table [3, 1] = table [i – 1, j]

= table [2, 1]

$\therefore$ table [3, 1] = 0

table [3, 2] With i = 3, j = 2, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 2] as

table [3, 2] = table [i – 1, j]

= table [2, 2]

$\therefore$ table [3, 2] = 3

table [3, 3] with i = 3, j = 3, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 3] as

table [3, 3] = table [i – 1, j]

= table [2, 3]

$\therefore$ table [3, 3] = 4

table [3, 4] With i = 3, j = 4, $w_i = 4$ and $v_i = 5$.

As $j \leq w_i$, we will obtain table [3, 4] as

table [3, 4] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum {table [2, 4], 5 + table [2, 0]}

= maximum {4, 5 + 0}

$\therefore$ table [3, 4] = 5

table [3, 5] With i = 3, j = 5, $w_i = 4$ and $v_i = 5$

As $j \geq w_i$, we will obtain table [3, 5] as

table [3, 5] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum {table [2, 5], 5 + table [2, 1]}

= maximum {7, 5 + 0}

$\therefore$ table [3, 5] = 7

The table with these values

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

table [4, 1] With i = 4, j = 1, $w_i = 5$, $v_i = 6$

As $j < w_i$, we will obtain table [4, 1] as

table [4, 1] = table [i – 1, j]

= table [3, 1]

∴ table [4, 1] = 0

table [4, 2] With i = 4, j = 2, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 2] as

table [4, 2] = table [i – 1, j]

= table [3, 2]

∴ table [4, 2] = 3

table [4, 3] With i = 4, j = 3, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 3] as

table [4, 3] = table [i – 1, j]

= table [3, 3]

∴ table [4, 3] = 4

table [4, 4] with i = 4, j = 4, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 4] as

table [4, 4] = table [i – 1, j]

= table [3, 4]

$\therefore$ table [4, 4] = 5

table [4, 5] With i = 4, j = 5, $w_i = 5$ and $v_i = 6$

As $j \geq w_i$, we will obtain table [4, 5] as

table [4, 5] = maximum $\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$

= maximum {table [3, 5] , 6 + table [3, 0]}

= maximum {7, 6 + 0}

$\therefore$ table [4, 5] = 7

Thus the table can be finally as given below

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7)

→ This is the total value of selected items

How to find actual knapsack items ?

Now, as we know that table [n, W] is the total value of selected items, that can be placed in the knapsack. Following steps are used repeatedly to select actual knapsack item

```
Let, i = n and k = W then
while (i>0 and k>0)
{
    if(table [i,k] ≠ table[i-1,k])then
    mark ith item as in knapsack
    i = i-1 and k=k-wi          //selection of ith item
else
    i = i-1   //do not select ith item
}
```

Let us apply these steps to the problem given in example 9.1. As we have obtained the final table -

	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0	0	3	4	5	7	
4	0	0	3	4	5	7	→ Start from here

i = 4 and k = 5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i.e., table [4, 5] = table [3, 5]

∴ do not select i^{th} i.e., 4^{th} item.

Now set i = i – 1

i = 3

items ↓ / Capacity →	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] = table [i – 1, k]

i.e., table [3, 5] = table [2, 5]

do not select i^{th} item i.e., 3^{rd} item.

Now set i = i – 1 = 2

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [2, 5] ≠ table [1, 5]

select i^{th} item.

That is, select 2^{nd} item.

Set i = i – 1 and k = k – w_i

i.e. i = 1 and k = 5 – 3 = 2

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [1, 2] ≠ table [0, 2]

select i^{th} item.

That is select 1^{st} item.

Set $i = i - 1$ and $k = k - w_i$

i.e. $i = 0$ and $k = 2 - 2 = 0$

Thus we have selected **item 1** and **item 2** for the knapsack. This solution can also be represented by solution vector **(1, 1, 0, 0)**.

Let us now discuss the algorithm for the knapsack problem using dynamic programming.

Algorithm

```
Algorithm Dynamic_Knapsack(n,W,w[ ],v[ ])
//Problem Description:This algorithm is for obtaining knapsack
//solution using dynamic programming
//Input: n is total number of items, W is the capacity of
//knapsack, w[ ] stores weights of each item and v[ ] stores
//the values of each item.
//Output: Returns the total value of selected items for the
//knapsack.
for (i←0 to n) do
{
 for (j←0 to W) do
 {
    table[i,0]=0    // table initialization
    table[0,j]=0
 }
}
for (i←0 to n) do
{
 for (j←0 to W) do
 {
   if(j<w[i]) then
      table[i,j]←  table[i-1,j]
   else if(j>=w[i]) then
      table[i,j]← max (table[i-1,j],(v[i]+table[i-1,j-w[i]])
```

```
    }
  }
  return table[n,W]
```

Analysis

In this algorithm the basic operation is **if ... else if** statement within two nested for loops.

Hence

$$c(n) = \sum_{i=0}^{n} \sum_{j=0}^{W} 1$$

$$= \sum_{i=0}^{n} W - 0 + 1$$

$$= \sum_{i=0}^{n} (W + 1)$$

$$= \sum_{i=0}^{n} W + \sum_{i=0}^{n} 1$$

$$= W \cdot \sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1$$

$$= W(n - 0 + 1) + (n - 0 + 1)$$

$$= W_n + W + n + 1$$

Thus $c(n) \approx W_n$

The time complexity of this algorithm is $\Theta(nW)$.

9.8.1 Memory Functions

While solving recurrence relation using dynamic programming approach common subproblems may be solved more than once and this makes inefficient solving of the problem. Hence memory function is a method that solves only necessary subproblems. Hence we can define the goal of memory function as : solve only subproblems that are **necessary** and solve these subproblems **only once.**

Memorization is a way to deal with overlapping subproblems in dynamic programming. During memorization -

1. After computing solution the solution to a subproblem, store it in a table.
2. Make use of recursive calls.

The 0-1 Knapsack Memory Function Algorithm is as given below -

Globally some values are to be set before the actual memory function algorithm

```
for (i←1 to n) do
 for (j←1 to W) do
    table[i,j]← -1
 for (j←0 to W) do
    table[0,j]← 0    //making 0th row 0
for i←1 to n do
     table[i,0]←0    //making 0th column 0
 Algorithm Mem_Fun_knapsack(i,j)
//Problem Description: Implementation of memory function method
//for the knapsack problem
//Input: i is the number of items and j denotes the knapsack's
//capacity
//Output: optimal solution subset
if (table[i,j] < 0) then
{
  if (j < w[i]) then
     value Mem_Fun_knapsack(i-1, j)
  else
    value ← max(Mem_Fun_knapsack(i-1, j),
       v[i] + Mem_Fun_knapsack(i-1, j-w[i]))
    table[i,j]← value
}
return table[i,j]
```

9.5 Warshall's and Floyd's Algorithm

In this section we will discuss two important applications of dynamic programming and those are : Warshall's and Floyd's algorithm.

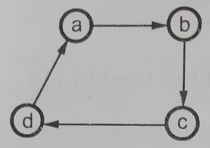
Warshall's algorithm is for computing transitive closure of a directed graph and Floyd's algorithm is for computing all pair shortest paths. Let us discuss them.

9.5.1 Warshall's Algorithm

Before discussing the actual algorithm let us revise some basic concepts.

1. **Digraph** : The graph in which all the edges are directed then it is called **digraph** or **directed graph**.

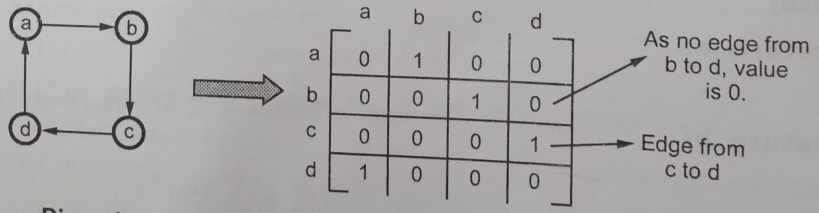
For example,



Digraph

2. **Adjacency matrix** : It is a representation of a graph by using matrix. If there exists an edge between the vertices V_i and V_j directing from V_i to V_j then entry in adjacency matrix in i^{th} row and j^{th} column is 1.

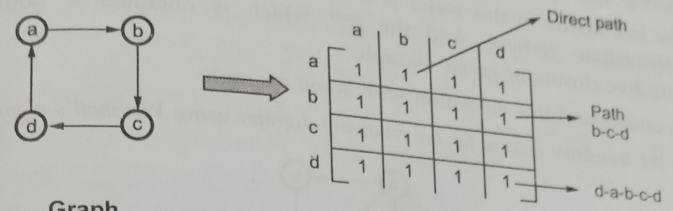
For example :



Digraph **Adjacency matrix**

3. **Transitive closure** : Transitive closure is basically a Boolean matrix (matrix with 0 and 1 values) in which the existence of directed paths of arbitrary lengths between vertices is mentioned.

For example :



Graph

Transitive closure

The transitive closure can be generated with Depth First Search (DFS) or with Breadth First Search (BFS). This traversing can be done any vertex. While computing transitive closure we have to start with some vertex and have to find all the edges which are reachable to every other vertex. The reachable edges for all the vertices has to be obtained.

Basic Concept

While computing the transitive closure, we have to traverse the graph several times. To avoid this repeated traversing through graph a new algorithm is discovered by S. Warshall which is called **Warshall's algorithm**. Warshall's algorithm constructs the transitive closure of given digraph with n vertices through a series of n-by-n Boolean matrices. The computations in Warshall's algorithm are given by following sequence,

$$R^{(0)}, \ldots, R^{(k-1)}, \ldots, R^{(k)}, \ldots R^{(n)}$$

Thus the key idea in Warshall's algorithm is **building of Boolean matrices.**

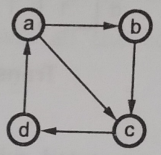
Procedure to be followed :

1. Start with computation of $R^{(0)}$. In $R^{(0)}$ any path with intermediate vertices is not allowed. That means only direct edges towards the vertices are considered. In other words the path length of one edge is allowed in $R^{(0)}$. Thus $R^{(0)}$ is adjacency matrix for the digraph.
2. Construct $R^{(1)}$ in which first vertex is used as intermediate vertex and a path length of two edges is allowed. Note that $R^{(1)}$ is build using $R^{(0)}$ which is already computed.

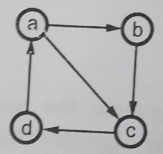
3. Go on building $R^{(k)}$ by adding one intermediate vertex each time and with more path length. Each $R^{(k)}$ has to be built from $R^{(k-1)}$.
4. The last matrix in this series is $R^{(n)}$, in this $R^{(n)}$ all the n vertices are used as intermediate vertices. And the $R^{(n)}$ which is obtained is nothing but the transitive closure of given digraph.

Let us understand this algorithm with some example.

Obtain the transitive closure for the following digraph using Warshall's algorithm.



Let us first obtain adjacency matrix for given digraph. It is denoted by $R^{(0)}$.

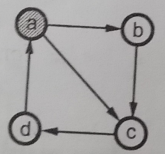


$R^{(0)}$ =

	a	b	c	d
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	1	0	0	0

The boxes at row and column are for getting $R^{(1)}$

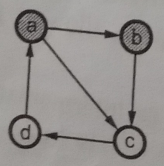
Here we have considered only direct edges from each source vertex. If the direct is edge is present make 1 in matrix.



$R^{(1)}$ =

	a	b	c	d
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	1	1	0	0

Here intermediate vertex is 'a' and using 'a' as intermediate vertex we have to find destination vertices with path length of 2. For instance from d to b a path exists with a as intermediate vertex.

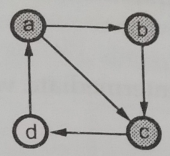


$R^{(2)}$ =

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	1	1	1	1

Here intermediate vertices are a and b, we have to find a path length of upto 3 edges going through intermediate vertices a and b. We will keep adjacency matrix of $R^{(1)}$ as it is and add more 1's for path length 3 with intermediate vertices a and b.

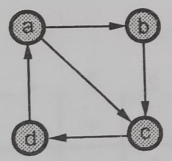
For instance : We get d-a-b-c. Hence matrix [d][c] = 1.



$R^{(3)}$ =

	a	b	c	d
a	0	1	1	1
b	0	0	1	1
c	0	0	0	1
d	1	1	1	1

Here intermediate vertices are a, b and c, we have to find a path length of upto 4 edges going through intermediate vertices a, b and c. For instance : a-b-c-d. Hence matrix [a][d] = 1.



$R^{(4)}$ =

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	1	1	1	1
d	1	1	1	1

Note that every vertex is reachable from every other vertex. Let us formulate this algorithm.

The central idea of this algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from previous $R^{(k-1)}$.

Let $r_{ij}^{(k)} \rightarrow$ is the element in i^{th} row and j^{th} column of matrix $R^{(k)}$. If there is a path from v_i to v_j in a digraph then $r_{ij}^{(k)} = 1$ otherwise 0.

$v_i \rightarrow$ is a list of intermediate vertices each numbered not greater than k.

The path from v_i to v_j can be computed with two cases.

1. The list of vertices that does not contain k^{th} vertex is noted. Then paths from v_i to v_j with intermediate vertices numbered not higher than $k-1$. Then $r_{ij}^{(k-1)} = 1$.

2. The path not containing k^{th} vertex v_k in intermediate vertices.

→ This indicates path from v_i to v_k with each intermediate vertex numbered not higher than $(k-1)$. Therefore $r_{ik}^{(k-1)} = 1$.

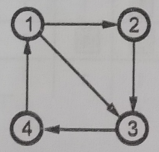
→ This indicates path from v_k to v_j with each intermediate vertex numbered not higher than $(k-1)$. Therefore $r_{kj}^{(k-1)} = 1$.

Then we can generate the elements of matrix $R^{(k)}$ from $R^{(k-1)}$ as follows :

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}$$

We can apply this formula to compute $R^{(k)}$ with elements $r_{ij}^{(k)}$.

Let us apply this formula to compute $R^{(k)}$.



$R^{(0)}$ =

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

Now we will compute $R^{(1)}$.

Assume $i = 1$ to 4 and $j = 1$ to 4, $k = 1$

$r_{11}^{1} = r_{ij}^{(k=1)}$ or $r_{ik}^{(k=1)}$ and $r_{kj}^{(k=1)}$ ← $i = 1, j = 1, k = 1$

$= r_{11}^{0}$ or r_{11}^{0} and r_{11}^{0}

$= 0$ or 0 and 0

$$r_{11}^{1} = 0$$

$r_{12}^{1} = r_{ij}^{(k=1)}$ or $r_{ik}^{(k=1)}$ and $r_{kj}^{(k=1)}$ ← $i = 1, j = 2, k = 1$

$= r_{12}^{0}$ or r_{11}^{0} and r_{12}^{0}

$= 1 \quad or \quad 0 \quad and \quad 1$

$= 1 \quad or \quad 0$

$$\boxed{r_{12}^1 = 1}$$

Continuing in this fashion we can compute remaining element. Only one change occurs from $R^{(0)}$ to $R^{(1)}$. i.e. when i = 4, j = 2, k = 1.

$$R^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$r_{42}^1 = r_{ij}^{k-1} \quad or \quad r_{ik}^{k-1} \quad and \quad r_{kj}^{k-1}$$

$$= r_{42}^0 \quad or \quad r_{41}^0 \quad and \quad r_{12}^0$$

$$= 0 \quad or \quad 1 \quad and \quad 1$$

$$= 0 \quad or \quad 1$$

$$\therefore \quad r_{42}^1 = 1$$

Thus we get $R^{(1)}$ as,

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

Thus we can compute $R^{(1)}$, $R^{(2)}$, $R^{(3)}$ and $R^{(4)}$ using above given formula.

Algorithm

```
Algorithm Warshall (Matrix [1... n, 1... n]
//Problem Description : This algorithm is for computing
//transitive closure using Warshall's algorithm
//Input : The adjacency matrix given by Matrix[1...n, 1...n]
//Output: The transitive closure of digraph
    R(0) ← Matrix //Initially adjacency matrix of
                  //digraph becomes R(0)
    for (K ← 1 to n) do
    {
       for (i ← 1 to n) do
       {
          for (j ← 1 to n) do
          {
             R(k)[i,j] ← R(k-1)[i,j] OR R(k-1)[i,k] AND
                                        R(k-1)[k,j]
          }
       }
    }
return R(n)
```

Analysis

Clearly time complexity of above algorithm is $\Theta(n^3)$ because in above algorithm the basic operation is computation of $R^{(k)}[i, j]$.

This operation is located within three nested for loops.

> The time complexity Warshall's algorithm is $\Theta(n^3)$.

C Program

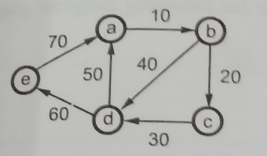
```
/*****************************************************************
This program is for computing transitive closure using Warshall's algorithm
*****************************************************************/

#include<stdio.h>
#include<conio.h>
```

```
Computing Transitive closure ...

R(1) =
 0 1 0 0
 0 0 0 1
 0 0 0 0
 1 1 1 0

R(2) =
 0 1 0 1
 0 0 0 1
 0 0 0 0
 1 1 1 1

R(3) =
 0 1 0 1
 0 0 0 1
 0 0 0 0
 1 1 1 1

R(4) =
 1 1 1 1
 1 1 1 1
 0 0 0 0
 1 1 1 1
```

9.5.2 Floyd's Algorithm

Floyd's algorithm is for finding the shortest path between every pair of vertices of a graph. The algorithm works for both directed and undirected graphs. This algorithm is invented by **R. Floyd** and hence is the name. Before getting introduced with this algorithm, let us revise few concepts related with graph.

Weighted graph : The weighted graph is a graph in which weights or distances are given along the edges. The weighted graph can be represented by weighted matrix as follows,



	a	b	c	d	e
a	0	10	∞	∞	∞
b	∞	0	20	40	∞
c	∞	∞	0	30	∞
d	50	∞	∞	0	60
e	70	∞	∞	∞	0

Here w[i][j] = 0 if i = j

w[i][j] = ∞ if there is no edge (direct edge) between i and j.

w[i][j] = Weight of edge.

Basic concept of Floyd's Algorithm

- The Floyd's algorithm is for computing shortest path between every pair of vertices of graph.
- The graph may contain negative edges but it should not contain negative cycles.
- The Floyd's algorithm requires a weighted graph.
- Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of n by n matrices :

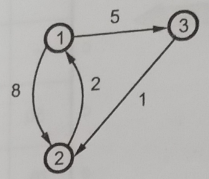
$$D^{(0)}, D^{(1)}, \ldots D^{(k-1)}, \ldots D^{(n)}$$

- In each matrix $D^{(k)}$ the shortest distance "d_{ij}" has to be computed between vertex v_i and v_j.
- In particular the series starts with $D^{(0)}$ with no intermediate vertex. That means $D^{(0)}$ is a matrix in which v_i and v_j i.e. i^{th} row and j^{th} column contains the weights given by direct edges. In $D^{(1)}$ matrix - the shortest distance going through one intermediate vertex (starting vertex as intermediate) with maximum path length of 2 edges is given continuing in this fashion we will compute $D^{(n)}$, containing the lengths of shortest paths

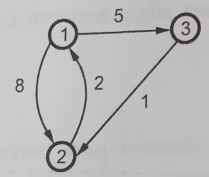
among all paths that can use all n vertices as intermediate. Thus we get all pair shortest paths from matrix $D^{(n)}$.

Let us first understand this algorithm with the help of some example.

Obtain the all pair-shortest path using Floyd's algorithm for the following weighted graph,



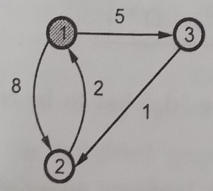
First we will compute weighted matrix with no intermediate vertex. i.e., $D^{(0)}$.



$D^{(0)}$ =

	1	2	3
1	0	8	5
2	2	0	∞
3	∞	1	0

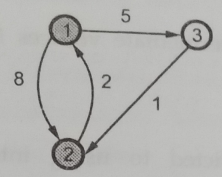
The matrix $D^{(0)}$ is simply a weighted matrix. If i = j then D[i][j] = 0, if there is no direct edge present in vertex v_i and v_j set i^{th} row and j^{th} column of matrix D as ∞. Otherwise put the weight given along the edge between v_i and v_j.



$D^{(1)}$ =

	1	2	3
1	0	8	5
2	2	0	7
3	∞	1	0

While computing $D^{(1)}$ we have considered '1' as intermediate vertex and path containing maximum two edges is taken. For instance : 2-1-3 path gives total weight 2 + 5 = 7. And 7 < ∞. Hence the previous entry of ∞ for 2^{nd} row and 3^{rd} column is replaced by 7.

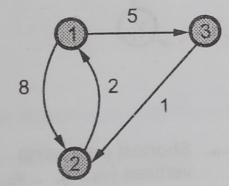


$D^{(2)}$ =

	1	2	3
1	0	8	5
2	2	0	7
3	3	1	0

While computing $D^{(2)}$, we have considered '1' and '2' as intermediate vertices path length of maximum three edges.

For instance : In $D^{(1)}$ we have got ∞ in 3^{rd} row and 1^{st} column as distance between vertex 3 and 1. But we get another distance 3 – 2 - 1 with weight 1 + 2 = 3. As $3 < \infty$. Put 3 in 3^{rd} row and 1^{st} column. Note that this distance is going through intermediate vertices '1' and '2'.



$D^{(3)}$ =

	1	2	3
1	0	6	5
2	2	0	7
3	3	1	0

D[1][2] was 8 in $D^{(2)}$ but it is 6 in $D^{(3)}$ because in computation of $D^{(3)}$ the allowed intermediate vertices are '1', '2' and '3' and the path length of maximum 4 edges is allowed.

Thus we have computed $D^{(3)}$. As n = total number of vertices = 3, we will stop computing series of $D^{(k)}$. The matrix $D^{(3)}$ is representing shortest paths from all pairs of vertices.

Let us now formulate this procedure.

Formulation

Let, $D^k[i, j]$ denotes the weight of shortest path from v_i to v_j using $\{v_1, v_2, v_3 \ldots v_k\}$ as intermediate vertices.

Initially $D^{(0)}$ is computed as weighted matrix.

There exists two cases -

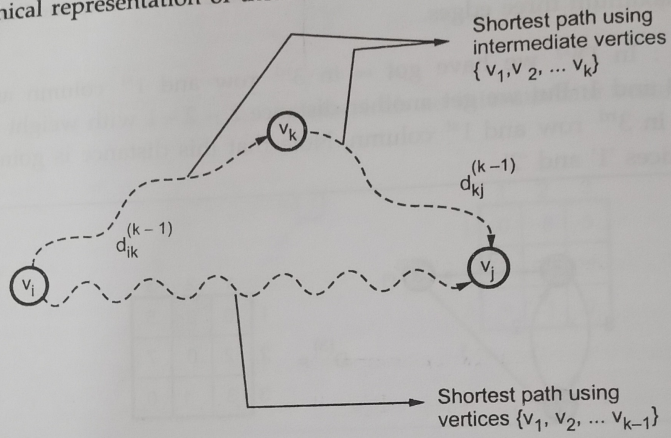
1. A shortest path from v_i to v_j with intermediate vertices from $\{v_1, v_2, \ldots v_k\}$ that does not use v_k. In this case,

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

2. A shortest path from v_i to v_j restricted to using intermediate vertices $\{v_1, v_2, \ldots v_k\}$ which uses v_k. In this case -

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

The graphical representation of these two cases is



Formulation of Floyd's algorithm

As these cases give us

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

We can now write,

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

Algorithm

```
Algorithm Floyd_shortest_path (wt[1...n, 1...n)
//Problem Description : This algorithm is for computing
//shortest path between all pairs of vertices.
//Input : The weighted matrix. wt[1...n, 1...n] for
//given graph.
```

```
//Output : The distance matrix D containing shortest
//paths.
    D ← wt    //Copy weighted matrix to matrix D
    //This initialization gives D(0)
    for k ← 1 to n do
    {
        for i ← 1 to n do
        {
            for j ← 1 to n do
            {
                D[i,j] ← min{D[i,j], (D[i,k] + D[k,j])}
            }
        }
    }
    return D   //computed D(n) is returned
```

Computing $D^{(k)}$ using two cases ① with k ② without k as intermediate vertex

Note that this is the basic operation in this algorithm

Analysis

In the above given algorithm the basic operation is -

$$D[i, j] \leftarrow \min\{D[i,j], D[i,k] + D[k, j]\}$$

This operation is with in three nested for loops, we can write

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} (n-1+1) \quad \because \sum_{i=l}^{u} 1 = u - l + 1$$

$$= \sum_{k=1}^{n} \sum_{i=1}^{n} n$$

$\sum_{i=1}^{n} i \approx \frac{1}{2} n^2$, we can neglect constants and write it as n^2

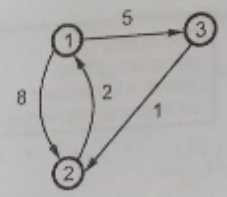
$$= \sum_{k=1}^{n} n^2$$

$C(n) = n^3$ $\quad \because \sum_{i=1}^{n} n^2 = \frac{1}{3}n^3$, we neglect constant

The time complexity of finding all pair shortest path is $\Theta(n^3)$.

Let us compute $D^{(0)}, D^{(1)}, \ldots D^{(n)}$ using the formula for Floyd's algorithm.

Consider,



$W = D^{(0)} =$

	1	2	3
1	0	8	5
2	2	0	∞
3	∞	1	0

For $D^{(1)}$, k = 1 i.e. vertex 1 can be an intermediate vertex.

$\therefore D^{(1)} =$

	1	2	3
1	0	8	5
2	2	0	7
3	∞	1	0

$D^1[2, 3] = \min \{D^0[2, 3], D^0[2, 1] + D^0[1, 3]\}$

$= \min \{\infty, 7\}$

$D^1[2, 3] = 7$

$\therefore D^{(2)} =$

	1	2	3
1	0	8	5
2	2	0	7
3	3	1	0

For $D^{(2)}$, k = 2 i.e., vertices 1 and 2 can be intermediate vertices.

$\therefore \quad D^2[1, 3] = \min \{D^1[1, 3], D^1[1, 2] + D^1[2, 3]\}$

$= \min \{5, (8 + 7)\}$

$D^2[1, 3] = 5$ i.e., remains unchanged

Similarly we will compute $D^2[3, 1]$.

$D^2[1, 3] = D^2[3, 1] = \min \{ D^1[3, 1], D^1[3, 2] + D^1[2, 1] \}$

$= \min \{\infty, 1 + 2\}$

$\therefore \quad D^2[3, 1] = 3$

For $D^{(3)}$ computation, k = 3 i.e. intermediate vertices can be 1, 2 and 3.

$D^{(3)} =$

	1	2	3
1	0	6	5
2	2	0	7
3	3	1	0

$$\therefore D^3[1,2] = \min\{D^2[1,2], D^2[1,3] + D^2[3,2]\}$$
$$= \min\{8, 5+1\}$$
$$D^3[1,2] = 6$$

Continuing in this fashion we can find all the shortest path distances from all the vertices.

'C' Program

```
/*****************************************************************
This program is for computing shortest path using
Floyd's Algorithm
*****************************************************************/
#include<stdio.h>
#include<conio.h>
void main()
{
 int wt[10][10],n,i,j;
 void Floyd_shortest_path(int matrix[10][10],int n);
 clrscr();
 printf("\n create a graph using adjacency matrix");
 printf("\n\n How many vertices are there ?");
 scanf("%d",&n);
 printf("\n Enter the elements");
 printf("[Enter 999 as infinity value]");
 for(i=1;i<=n;i++)
 {
    for(j=1;j<=n;j++)
    {
     printf("\nwt[%d][%d]",i,j);
```