GREEDY METHOD

- These problems have n inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies those constraints is called a feasible solution.
- We need to find a feasible Solution that either maximizes or minimizes a given objective function.
- A feasible solution that does this is called an optimal solution.

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Fractional Knapsack Problem:

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$.

Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$.

Minimum Weight:

(x1, x2, x3)=(0,2/3,1)=0+2/3*24+1*15= Rs. 31

Maximum Profit:

(x1, x2, x3)=(1,2/15,0)=25+2/15*24+0=27

P/w value

```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respective
// of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1]
// m is the knapsack size and x[1 : n] is the solution vector.
{
    for i := 1 to n do x[i] := 0.0; // Initialize x.
    U := m;
    for i := 1 to n do
    {
        if (w[i] > U) then break;
        x[i] := 1.0; U := U − w[i];
    }
    if (i ≤ n) then x[i] := U/w[i];
}
```

- Find an optimal solution to the knapsack instance n=7

M=15

Weights = {2,3,5,7,1,4,1}

Profits ={10,5,15,7,6,18,3}

Job Sequencing with Deadlines

- We are given a set of n jobs. Associated with job i is an integer deadline

di >0 and a profit pi >0.

- For any job i the profit pi is earned iff the job is Completed by its deadline.
- To complete a job, one has to process the job on a machine for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each

Job in this subset can be completed by its deadline.

- The value of a feasible solution J is the sum of the profits of the jobs in J,
- An optimal solution is a feasible solution with maximum value.

Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	**processing sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

different method to determine the feasibility of a partial solution. If J is a feasible subset of jobs, then we can determine the processing times for each of the jobs using the rule: if job i hasn't been assigned a processing time, then assign it to the slot $[\alpha - 1, \alpha]$, where α is the largest integer r such that $1 \leq r \leq d_i$ and the slot $[\alpha - 1, \alpha]$ is free. This rule simply delays the processing of job i as much as possible. Consequently, when J is being built up job by job, jobs already in J do not have to be moved from their assigned slots to accommodate the new job. If for the new job being considered there is no α as defined above, then it cannot be included in J. The proof of the validity of this statement is left as an exercise.

Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$.

Let $n = 5, (p_1, \ldots, p_5) = (20, 15, 10, 5, 1)$ and $(d_1, \ldots, d_5) = (2, 2, 1, 3, 3)$. Using the above feasibility rule, we have

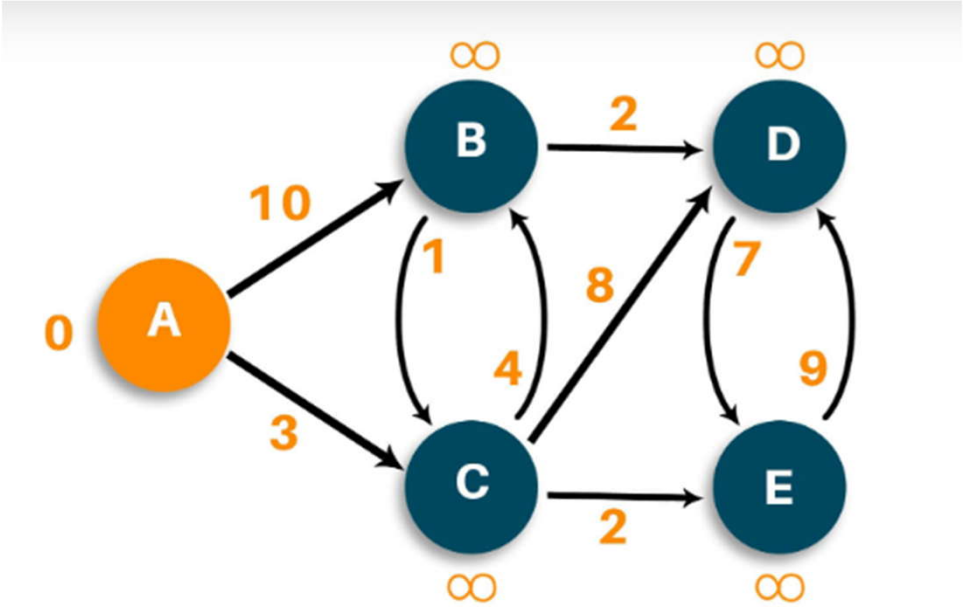
J	assigned slots	job considered	action	profit
$\emptyset$	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	cannot fit; reject	35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2, 3]	5	reject	40

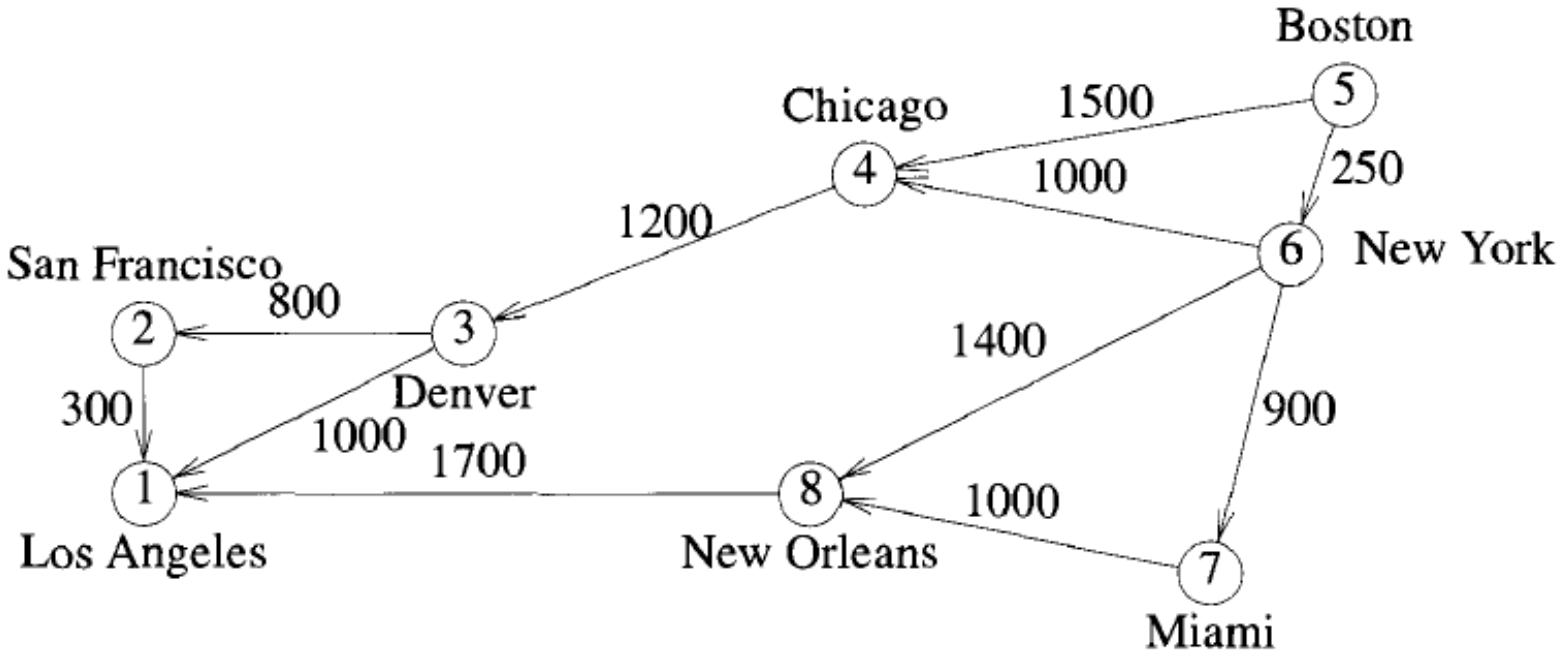
The optimal solution is $J = \{1, 2, 4\}$ with a profit of 40. □

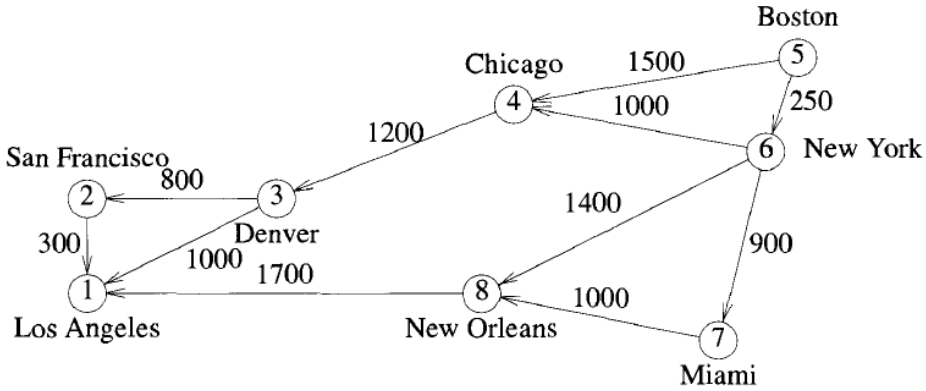
What is the solution generated by the function JS when $n = 7$, $(p_1, p_2, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$, and $(d_1, d_2, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$?

Single source shortest paths or Dijkstra's Algorithm

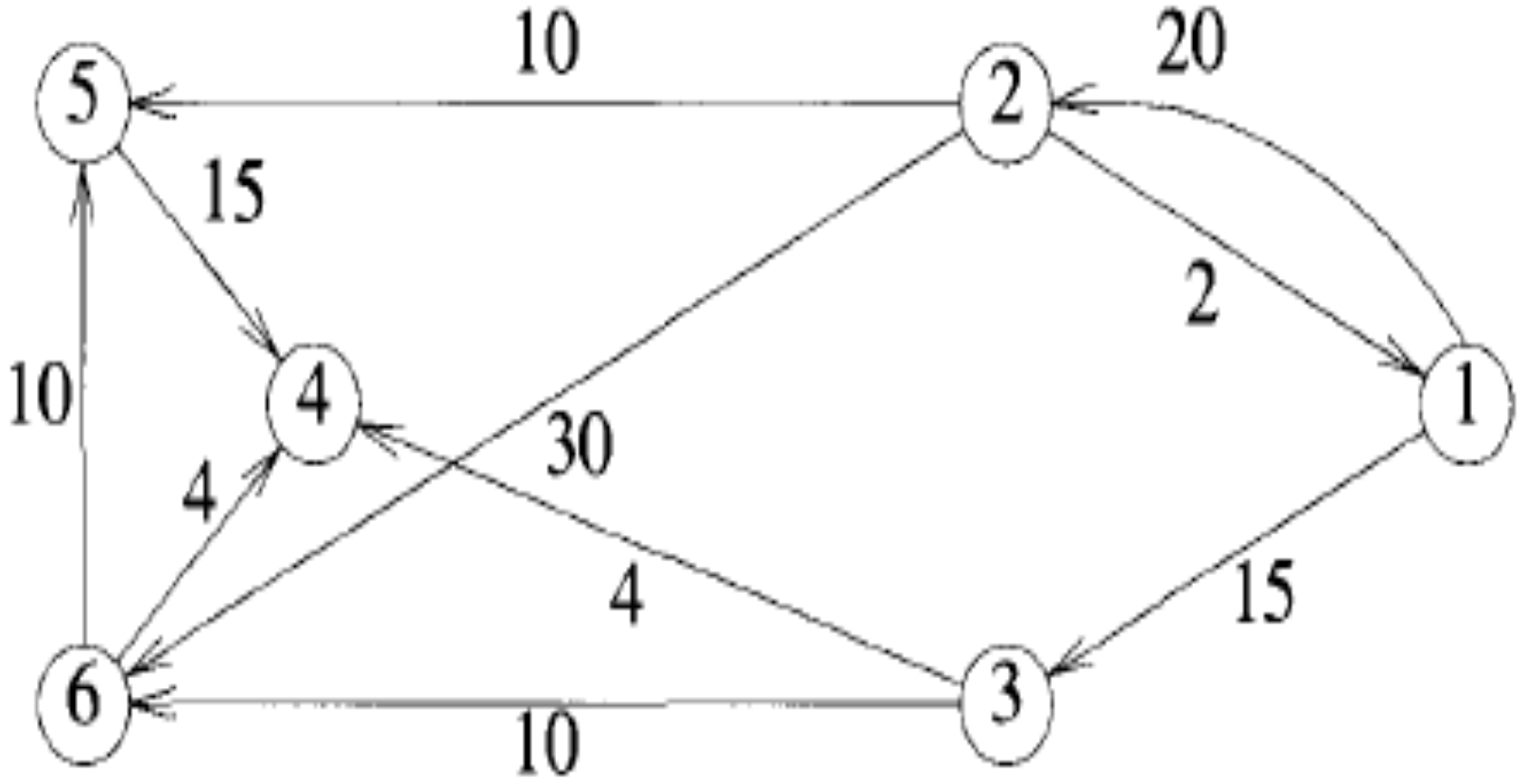
given a directed graph $G = (V, E)$, a weighting function *cost* for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G.

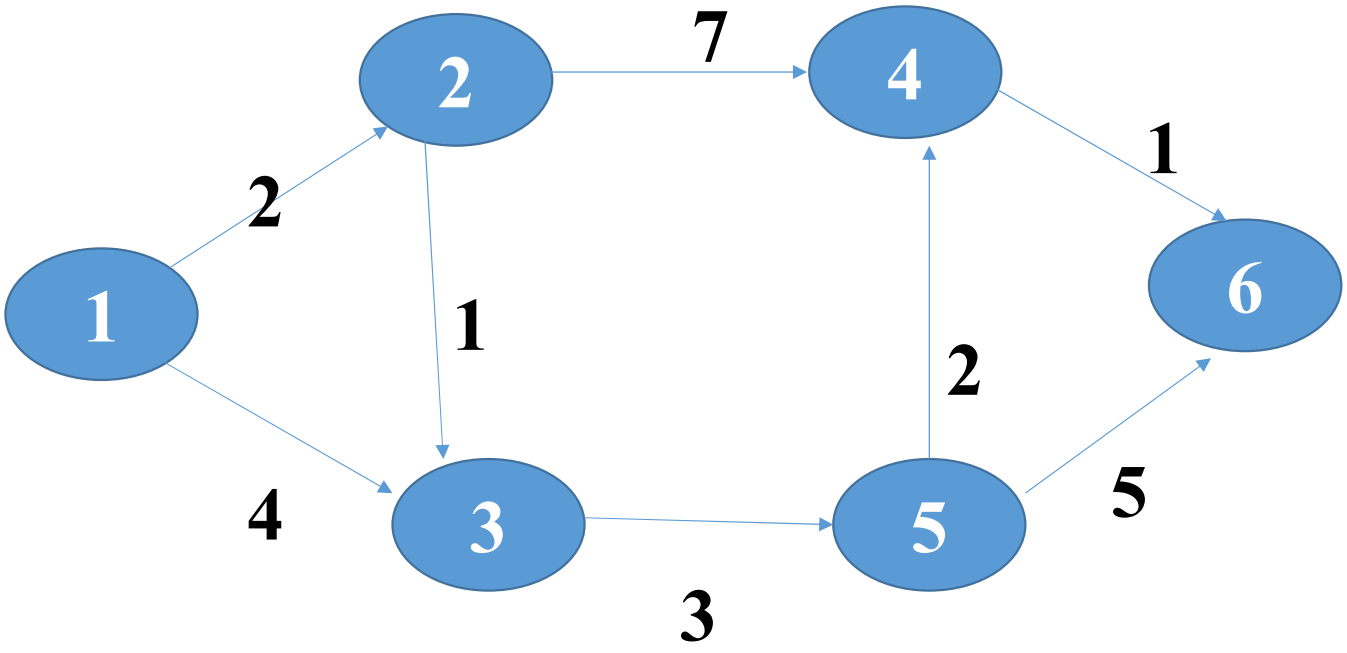



Boston
5
Chicago
1500
4
1000
250
1200
6
New York
San Francisco
800
2
3
1400
300
Denver
900
1000
1700
1
8
1000
New Orleans
7
Los Angeles
Miami



Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	{5}	6	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	{5,6}	7	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	{5,6,7}	4	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	$+\infty$	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									


5
2
4
1
6
3
10
20
15
2
10
30
4
4
15
10


1
2
3
4
5
6
2
4
7
1
1
2
4
5
3

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                  dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

```
1   Algorithm Perm(a, k, n)
2   {
3       if (k = n) then write (a[1 : n]); // Output permutation.
4       else // a[k : n] has more than one permutation.
5           // Generate these recursively.
6           for i := k to n do
7           {
8               t := a[k]; a[k] := a[i]; a[i] := t;
9               Perm(a, k + 1, n);
10              // All permutations of a[k + 1 : n]
11              t := a[k]; a[k] := a[i]; a[i] := t;
12          }
13  }
```

For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Objects:	1	2	3	4	5	6	7
Profit (P):	10	15	7	8	9	4	**3**
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15

n (no of items): 7

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

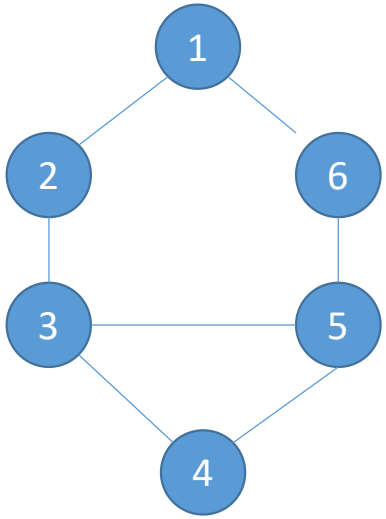
Answer the following questions-

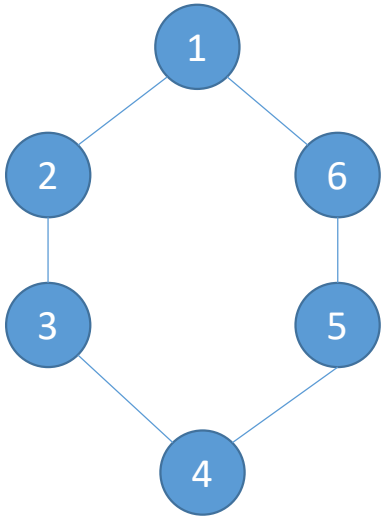
1. Write the optimal schedule that gives maximum profit.

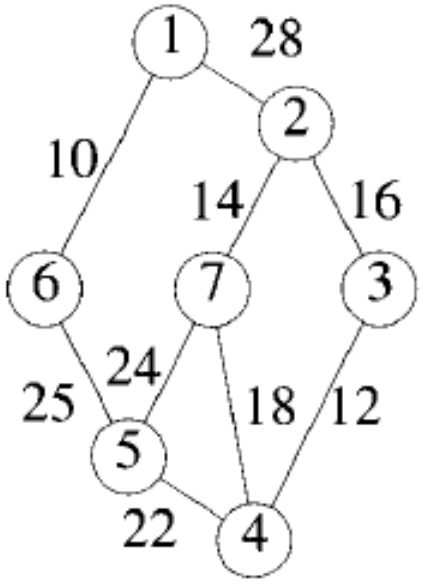
MINIMUM COST SPANNING TREE

KRUSKAL'S

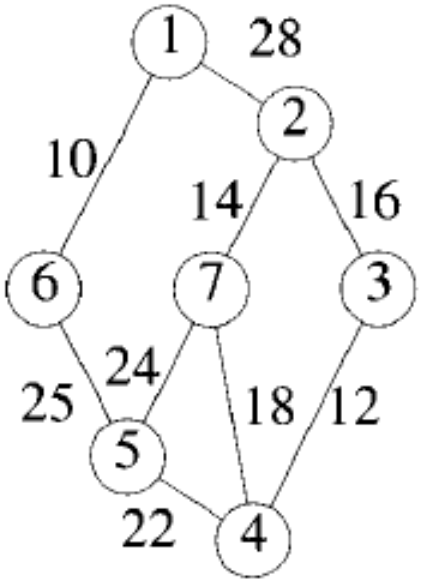
PRIM'S



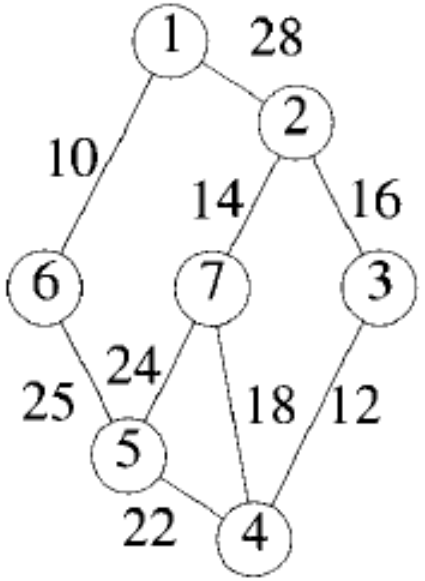



1
28
2
10
14
16
6
7
3
24
25
18
12
5
22
4

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```



Kruskal's Algorithm

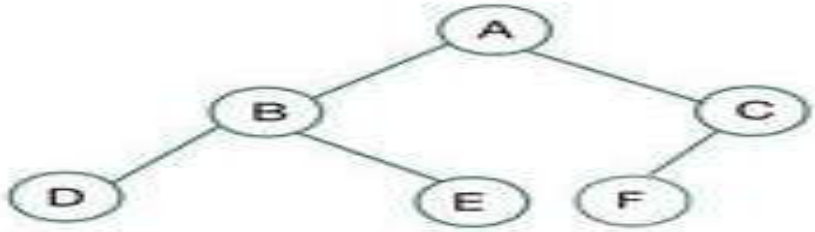


binary tree

We know a **tree** is a non-linear data structure. It has no limitation on the number of children. A binary tree has a limitation as any node of the tree has at most two children: a left and a right child.

What is a Complete Binary Tree?

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.



Complete Binary Tree

The height of the given binary tree is 2 and the maximum number of nodes in that tree is $n= 2^{h+1}-1 = 2^{2+1}-1 = 2^3-1 = 7$.

Heap (data structure)

a **heap** is a specialized tree-based data structure that is essentially an almost complete tree that satisfies the **heap property**

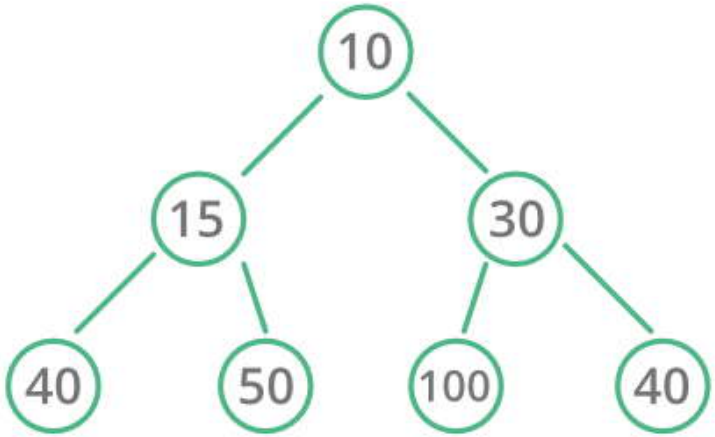
Types of Heap Data Structure

Generally, Heaps can be of two types:

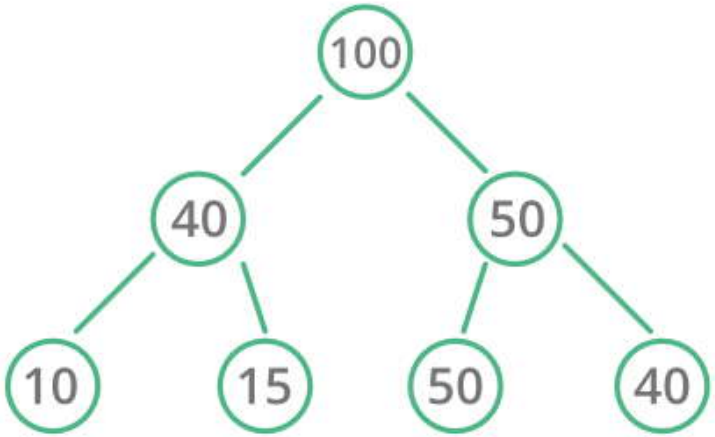
1.Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

2.Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

Heap Data Structure



Min Heap



Max Heap

Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.

- **Insertion:** process to insert an element in existing heap time complexity O(log N).

- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).

(40, 80, 35, 90, 45, 50, and 70)

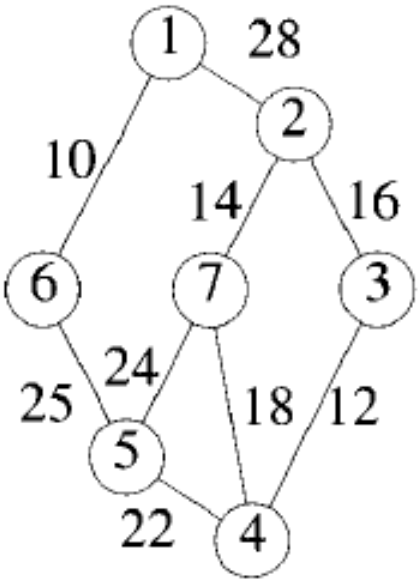
```
1   Algorithm Adjust(a, i, n)
2   // The complete binary trees with roots 2i and 2i + 1 are
3   // combined with node i to form a heap rooted at i. No
4   // node has an address greater than n or less than 1.
5   {
6       j := 2i; item := a[i];
7       while (j ≤ n) do
8       {
9           if ((j < n) and (a[j] < a[j + 1])) then j := j + 1;
10              // Compare left and right child
11              // and let j be the larger child.
12          if (item ≥ a[j]) then break;
13              // A position for item is found.
14          a[⌊j/2⌋] := a[j]; j := 2j;
15      }
16      a[⌊j/2⌋] := item;
17  }
```

```
1   Algorithm Heapify(a, n)
2   // Readjust the elements in a[1 : n] to form a heap.
3   {
4       for i := ⌊n/2⌋ to 1 step −1 do Adjust(a, i, n);
5   }
```

```
1   Algorithm HeapSort(a, n)
2   // a[1 : n] contains n elements to be sorted. HeapSort
3   // rearranges them inplace into nondecreasing order.
4   {
5       Heapify(a, n); // Transform the array into a heap.
6       // Interchange the new maximum with the element
7       // at the end of the array.
8       for i := n to 2 step −1 do
9       {
10          t := a[i]; a[i] := a[1]; a[1] := t;
11          Adjust(a, 1, i − 1);
12      }
13  }
```

Kruskal's Algorithm

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```



Dynamic Programming

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

[Principle of optimality] The principle of optimality stat that an optimal sequence of decisions has the property that whatever t initial state and decision are, the remaining decisions must constitute optimal decision sequence with regard to the state resulting from the fi decision.

1. Memorization Method : Top Down

Memorization is a technique based on dynamic programming which is used to improve the performance of a recursive algorithm by ensuring that the method does not run for the same set of inputs more than once by keeping a record of the results for the provided inputs(stored in an array).Memorization can be achieved by implementing top down approach of the recursive method.
Let us understand this scenario with the help of basic Fibonacci example

```
Int fib(int n)
{
If (n<=1)
return n;
else
return fib(n-2)+fib(n-1);
}
```

$$Fb(n)=\begin{cases}0 \text{ if } n=0\\ 1 \text{ if } n=1\\ Fib(n-2)+fib(n-1) \text{ if } n>1\end{cases}$$

1. Tabulation Method : Bottom Up

Tabulation is an approach where you solve a dynamic programming problem by first filling up a table, and then compute the solution to the original problem based on the results in this table.

Let us understand this scenario with the help of basic Fibonacci example

```
Int fib(int n)
{
if(n<=1)
return n;
F[0]=0;
F[1]=1;
for(i=2;i<=n;i++)
{ F[i]=F[i-2]+F[i-1];
}
Return F[n];
}
```

Fb(n)= { 0 if n=0
1 if n=1
Fib(n-2)+fib(n-1) if n>1

1. Floyd-Warshall Algorithm or All pairs Shortest paths
2. Optimal Binary Search Trees
3. Knapsack Problem
4. Traveling Salesman Problem
5. Matrix chain multiplication

Travelling Salesperson Problem

Let $G = (V, E)$ be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A *tour* of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

Problem

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

$$
\begin{aligned}
g(1,\{2,3,4\}) &= \min\{c_{12}+g(2,\{3,4\}), c_{13}+g(3,\{2,4\}), c_{14}+g(4,\{2,3\})\} \\
&= \min\{35, 40, 43\} \\
&= 35
\end{aligned}
$$

$$
\begin{array}{llll}
g(2,\{3\}) = c_{23}+g(3,\phi) = 15 & & g(2,\{4\}) = 18 & \\
g(3,\{2\}) = 18 & & g(3,\{4\}) = 20 & \\
g(4,\{2\}) = 13 & & g(4,\{3\}) = 15 &
\end{array}
$$

$$
\begin{aligned}
g(2,\{3,4\}) &= \min\{c_{23}+g(3,\{4\}), c_{24}+g(4,\{3\})\} &= 25 \\
g(3,\{2,4\}) &= \min\{c_{32}+g(2,\{4\}), c_{34}+g(4,\{2\})\} &= 25 \\
g(4,\{2,3\}) &= \min\{c_{42}+g(2,\{3\}), c_{43}+g(3,\{2\})\} &= 23
\end{aligned}
$$

0/1 Knapsack Problem

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Tabulation Method: Bottom Up

Example of 0/1 knapsack problem.
Consider the problem having weights and profits:
Weights: {3, 4, 6, 5}
Profits: {2, 3, 1, 4}
The weight of the knapsack is 8 kg
The number of items is 4

The above problem can be solved by using the following method:

xi = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

24 = 16; So. 16 possible combinations can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

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

In the above matrix, columns represent the weight, i.e., 8. The rows represent the profits and weights of items. Here we have not taken the weight 8 directly, problem is divided into sub-problems, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8. The solution of the sub-problems would be saved in the cells and answer to the problem would be stored in the final cell. First, we write the weights in the ascending order and profits according to their weights shown as below:

w_i = {3, 4, 5, 6}

p_i = {2, 3, 4, 1}

The first row and the first column would be 0 as there is no item for w=0

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

When i=1, W=1

w_1 = 3; Since we have only one item in the set having weight 3, but the capacity of the knapsack is 1. We cannot fill the item of 3kg in the knapsack of capacity 1 kg so add 0 at M[1][1] shown as below:

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

Using Subset Method

Consider the knapsack instance $n = 3, (w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

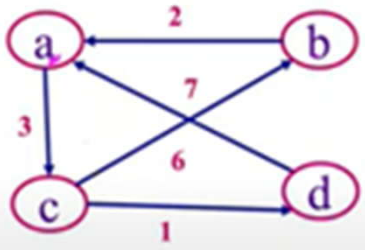
$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

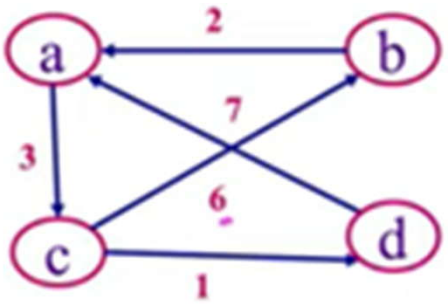
All Pairs Shortest Paths

Let $G = (V, E)$ be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that $cost(i, i) = 0$, $1 \leq i \leq n$. Then $cost(i, j)$ is the length (or cost) of edge $\langle i, j \rangle$ if $\langle i, j \rangle \in E(G)$ and $cost(i, j) = \infty$ if $i \neq j$ and $\langle i, j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i, j)$ is the length of a shortest path from i to j.

FLOYD-WARSHALL'S ALGORITHM

- To find the shortest distance from each nodes to all other nodes
- Floyd's Algorithm is commonly called All pair shortest path problem
- Modified version of Warshall's Algorithm





- Solve the all-pair Shortest path problem for the given digraph

Cost Matrix =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Step 1: Consider Shortest path through vertex a

$R^{(0)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

$R^{(1)}$ =

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

$b,c = \min[(b \to c), (b \to a + a \to c)] = \min[\infty, 2+3] = 5$

$b,d = \min[b \to d, (b \to a + a \to d)] = \min[\infty, 2+\infty] = \infty$

$c,b = \min[c \to b, (c \to a + a \to b)] = \min[7, \infty+\infty] = 7$

$c,d = \min[c \to d, (c \to a + a \to d)] = \min[1, \infty+\infty] = 1$

$d,b = \min[d \to b, (d \to a + a \to b)] = \min[\infty, 6+\infty] = \infty$

$d,c = \min[d \to c, (d \to a + a \to c)] = \min[\infty, 6+3] = 9$

Step 2: Consider Shortest path through vertex b

$R^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

$R^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

$a,c = \min[a \to c,\ a \to b + b \to c] = \min[3,\ \infty + 5] = 3$

$a,d = \min[a \to d,\ a \to b + b \to d] = \min[\infty,\ \infty + \infty] = \infty$

$c,a = \min[c \to a,\ c \to b + b \to a] = \min[\infty,\ 7 + 2] = 9$

$c,d = \min[c \to d,\ c \to b + b \to d] = \min[1,\ 7 + \infty] = 1$

$d,a = \min[d \to a,\ d \to b + b \to a] = \min[6,\ \infty + 2] = 6$

$d,c = \min[d \to c,\ d \to b + b \to c] = \min[9,\ \infty + 5] = 9$

Step 3: Consider Shortest path through vertex c

$R^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

$R^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

a,b = Min [a→b, a→c+c→b] = Min [∞, 3+7] = 10

a,d = Min [a→d, a→c+c→d] = Min [∞, 3+1] = 4

b,a = Min [b→a, b→c+c→a] = Min [2, 5+9] = 2

b,d = Min [b→d, b→c+c→d] = Min [∞, 5+1] = 6

d,a = Min [d→a, d→c+c→a] = Min [6, 9+9] = 6

d,b = Min [d→b, d→c+c→b] = Min [∞, 9+7] = 16

Step 4: Consider Shortest path through vertex d

$R^{(3)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$R^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

a,b = Min[a→b, a→d+d→b] = Min[10, 4+16] = 10

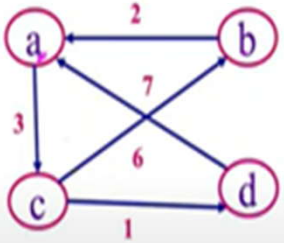
a,c = Min[a→c, a→d+d→c] = Min[3, 4+9] = 3

b,a = Min[b→a, b→d+d→a] = Min[2, 6+6] = 2

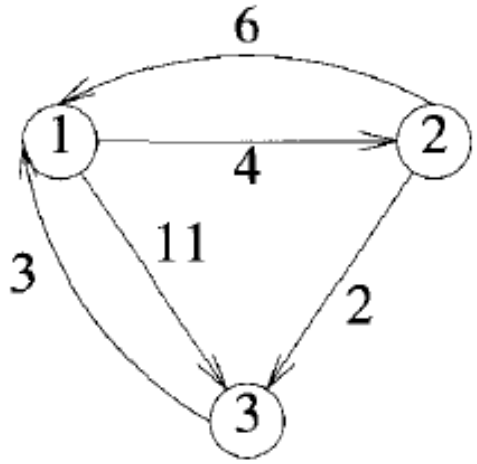
b,c = Min[b→c, b→d+d→c] = Min[5, 6+9] = 5

c,a = Min[c→a, c→d+d→a] = Min[9, 6+1] = 7

c,b = Min[c→b, c→d+d→b] = Min[7, 1+16] = 7



$$A(i,j) = \min \{ \min_{1 \le k \le n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\}$$



(a) Example digraph

Step 1: Solving the equation for, k = 1;

$A^1(1, 1) = \min\{(A^0(1, 1) + A^0(1, 1)), c(1, 1)\} = \min\{0 + 0, 0\} = 0$
$A^1(1, 2) = \min\{(A^0(1, 1) + A^0(1, 2)), c(1, 2)\} = \min\{(0 + 4), 4\} = 4$
$A^1(1, 3) = \min\{(A^0(1, 1) + A^0(1, 3)), c(1, 3)\} = \min\{(0 + 11), 11\} = 11$
$A^1(2, 1) = \min\{(A^0(2, 1) + A^0(1, 1)), c(2, 1)\} = \min\{(6 + 0), 6\} = 6$
$A^1(2, 2) = \min\{(A^0(2, 1) + A^0(1, 2)), c(2, 2)\} = \min\{(6 + 4), 0)\} = 0$
$A^1(2, 3) = \min\{(A^0(2, 1) + A^0(1, 3)), c(2, 3)\} = \min\{(6 + 11), 2\} = 2$
$A^1(3, 1) = \min\{(A^0(3, 1) + A^0(1, 1)), c(3, 1)\} = \min\{(3 + 0), 3\} = 3$
$A^1(3, 2) = \min\{(A^0(3, 1) + A^0(1, 2)), c(3, 2)\} = \min\{(3 + 4), \propto\} = 7$
$A^1(3, 3) = \min\{(A^0(3, 1) + A^0(1, 3)), c(3, 3)\} = \min\{(3 + 11), 0\} = 0$

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min\{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min\{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min\{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min\{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min\{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min\{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min\{(A(2, 2) + A(2, 1), c(2, 1)\} = \min\{(0 + 6), 6\} = 6$

$A^2(2, 2) = \min\{(A(2, 2) + A(2, 2), c(2, 2)\} = \min\{(0 + 0), 0\} = 0$

$A^2(2, 3) = \min\{(A(2, 2) + A(2, 3), c(2, 3)\} = \min\{(0 + 2), 2\} = 2$

$A^2(3, 1) = \min\{(A(3, 2) + A(2, 1), c(3, 1)\} = \min\{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min\{(A(3, 2) + A(2, 2), c(3, 2)\} = \min\{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min\{(A(3, 2) + A(2, 3), c(3, 3)\} = \min\{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min\{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min\{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min\{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min\{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min\{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min\{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min\{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min\{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min\{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min\{(2 + 7), 0\} = 0$

$A^3(2, 3) = \min\{A^2(2, 3) + A^2(3, 3), c(2, 3)\} = \min\{(2 + 0), 2\} = 2$

$A^3(3, 1) = \min\{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min\{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min\{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min\{(0 + 7), 7\} = 7$

$A^3(3, 3) = \min\{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min\{(0 + 0), 0\} = 0$

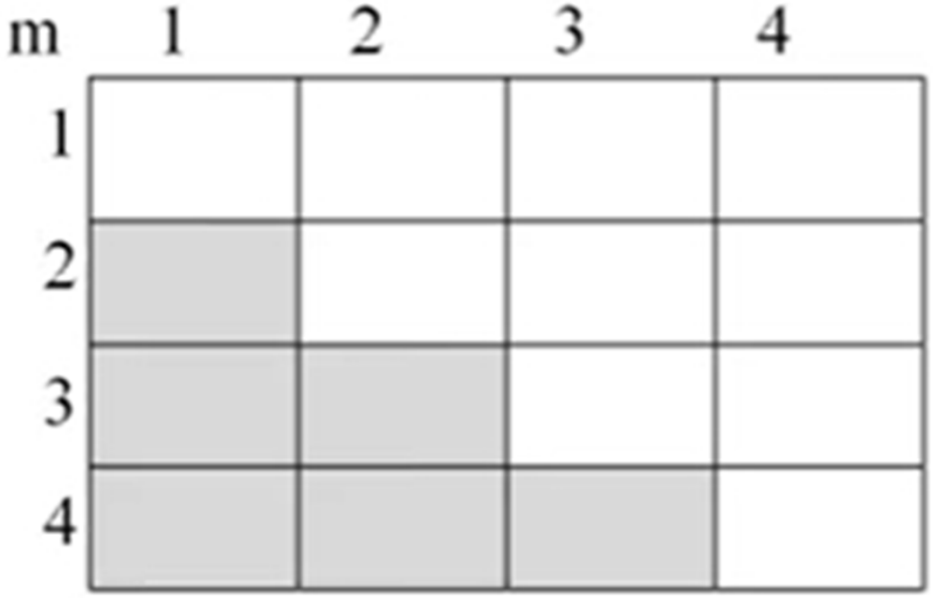
```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

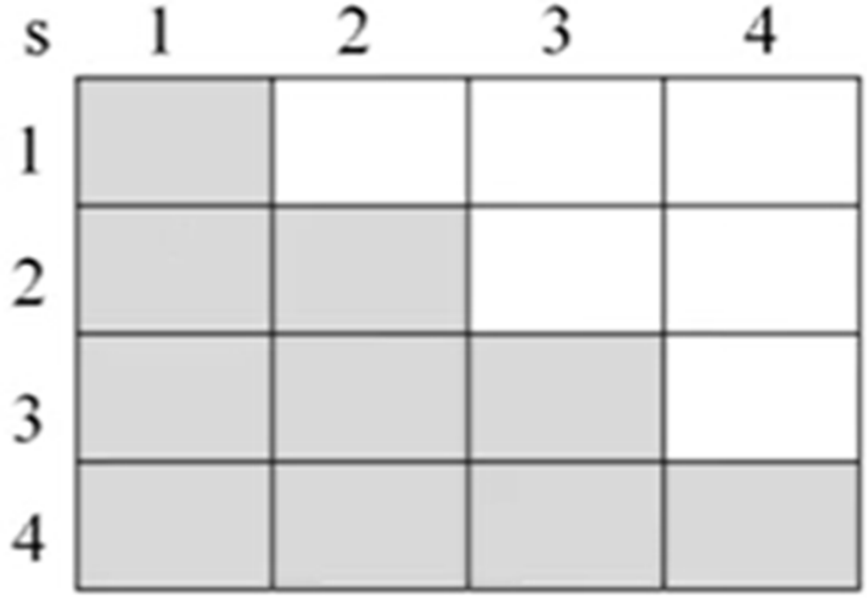
Matrix Chain Multiplication

Given a chain of 4 matrices $<A_1, A_2, A_3, A_4>$ with dimensions $<5X4>, <4X6>, <6X2>, <2X7>$ respectively. Using Dynamic programming find the minimum number of scalar multiplications needed and also write the optimal multiplication order

$$\begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} \times \begin{bmatrix} a_2 & b_2 \\ c_2 & d_2 \end{bmatrix} = \begin{bmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{bmatrix}$$

A_1	A_2	A_3	A_4
5x4	4x6	6x2	2x7





A_1	A_2	A_3	A_4
5x4	4x6	6x2	2x7

m	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

s	1	2	3	4
1				
2				
3				
4				

A_1	A_2	A_3	A_4
5x4	4x6	6x2	2x7

m	1	2	3	4
1	0	120		
2		0	48	
3			0	
4				0

s	1	2	3	4
1		1		
2			2	
3				
4				

A_1	A_2	A_3	A_4
5x4	4x6	6x2	2x7

m	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

s	1	2	3	4
1		1		
2			2	
3				
4				

A_1 A_2 A_3 A_4
5x4 4x6 6x2 2x7

m	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

s	1	2	3	4
1		1		
2			2	
3				3
4				

A_1 $(A_2$ $A_3)$
5x4 4x6 6x2

4x2

0 48 **88**

$(A_1$ $A_2)$ A_3
5x4 4x6 6x2

5x6

120 0 180

A_1 A_2 A_3 A_4

5x4 4x6 6x2 2x7

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	
3				3
4				

A_2 $(A_3$ $A_4)$

4x6 6x2 2x7

6x7

0 84

168 252

$(A_2$ $A_3)$ A_4

4x6 6x2 2x7

4x2

48 0

56 104

A_1 A_2 A_3 A_4
5x4 4x6 6x2 2x7

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	3
3				3
4				

$A_1 (A_2A_3A_4)$ ➡ 244

$(A_1A_2) (A_3A_4)$

$(A_1A_2A_3) A_4$

A_1 $(A_2$ A_3 $A_4)$
5x4 4x6 6x2 2x7

4x7

0 104 244

140

A_1	A_2	A_3	A_4
5x4	4x6	6x2	2x7

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	3
3				3
4				

$A_1 (A_2 A_3 A_4)$ ➡ 244

$(A_1 A_2)(A_3 A_4)$ ➡ 414

$(A_1 A_2 A_3) A_4$

$(A_1$	$A_2)$	$(A_3$	$A_4)$
5x4	4x6	6x2	2x7
	5x6		6x7
	120		84
	210		414

A_1 A_2 A_3 A_4

5x4 4x6 6x2 2x7

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	3
3				3
4				

$A_1 (A_2A_3A_4)$ ➡ 244

$(A_1A_2) (A_3A_4)$ ➡ 414

$(A_1A_2A_3) A_4$ ➡ 158

$(A_1$ A_2 $A_3)$ A_4

5x4 4x6 6x2 2x7

5x2

88 0

158

70

	A_1	A_2	A_3	A_4
	5x4	4x6	6x2	2x7

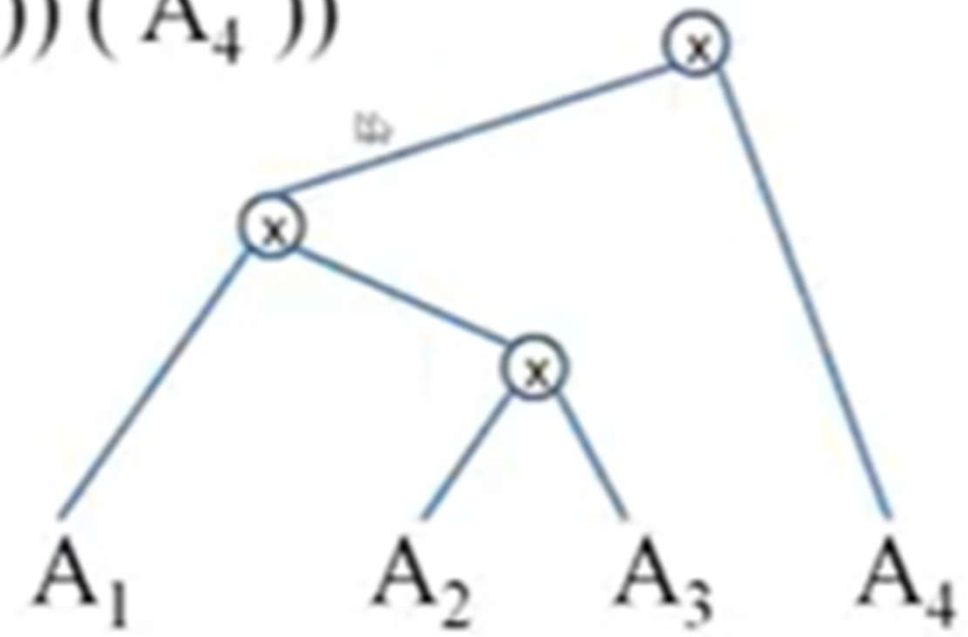
m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

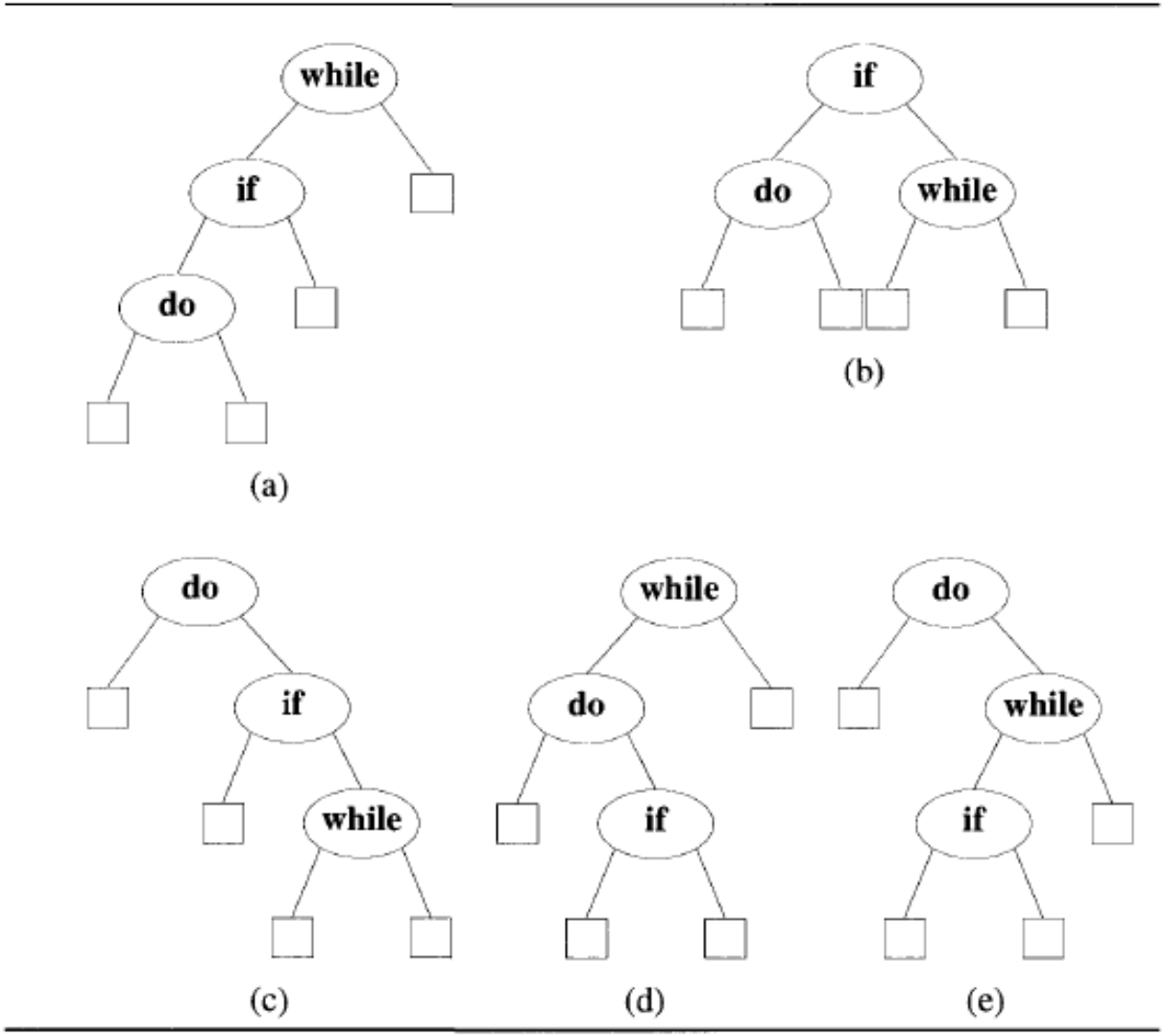
$(A_1 \quad A_2 \quad A_3) \quad (A_4)$

$((A_1 (A_2 \quad A_3)) (A_4))$

$((A_1(A_2\ A_3))(A_4))$



OPTIMAL BINARY SEARCH TREES



$$cost(l) = \sum_{1 \le i < k} p(i) * \text{level}(a_i) + \sum_{0 \le i < k} q(i) * (\text{level}(E_i) - 1)$$

With equal probabilities $p(i) = q(i) = 1/7$ for all i, we have

$cost$(tree a)	=	15/7	$cost$(tree b)	=	13/7
$cost$(tree c)	=	15/7	$cost$(tree d)	=	15/7
$cost$(tree e)	=	15/7			

As expected, tree b is optimal. With $p(1) = .5$, $p(2) = .1$, $p(3) = .05$, $q(0) = .15$, $q(1) = .1$, $q(2) = .05$ and $q(3) = .05$ we have

$cost$(tree a)	=	2.65	$cost$(tree b)	=	1.9
$cost$(tree c)	=	1.5	$cost$(tree d)	=	2.05
$cost$(tree e)	=	1.6			

$$c(i,j) = \min_{i<k\leq j}\{c(i,k-1)+c(k,j)\}+w(i,j)$$

$r(i,j)$ is the value of k that minimizes

$$w(i,j) = (j) + q(j) + w(i,j-1),$$

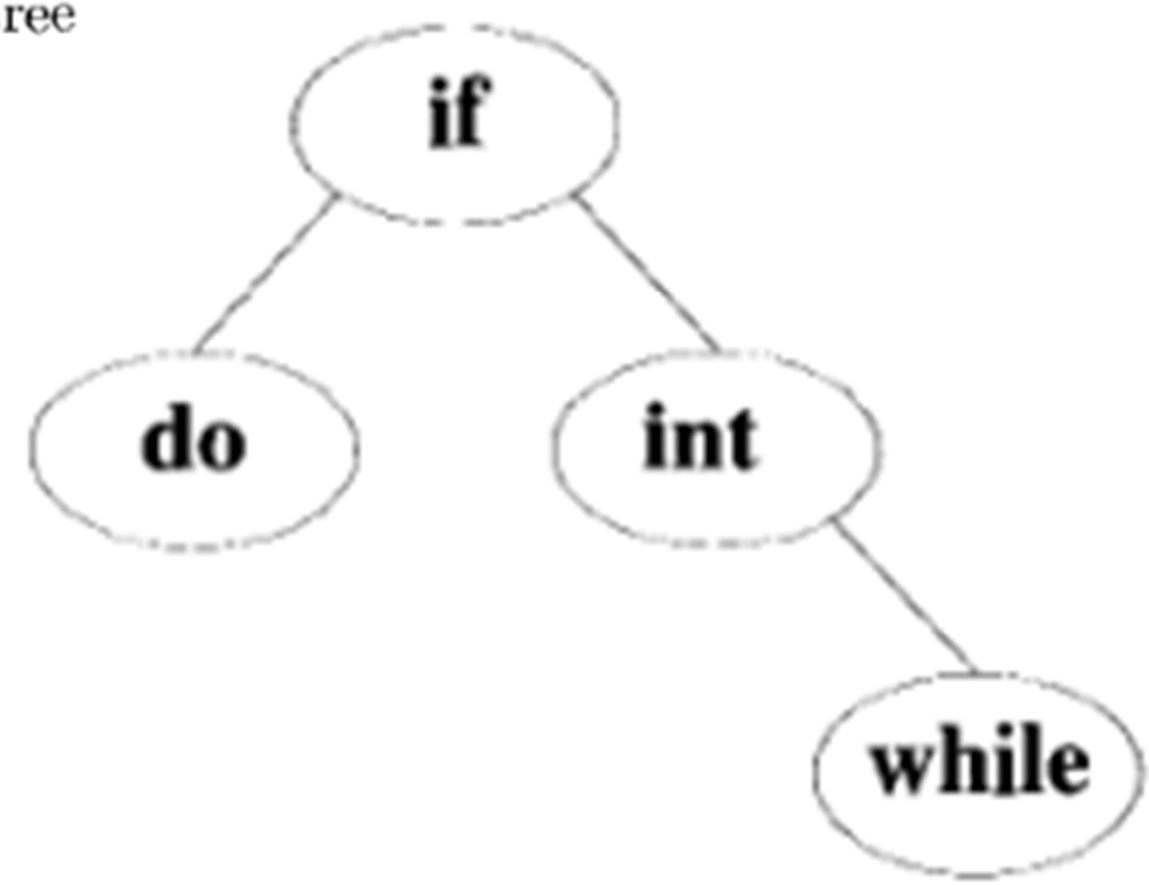
Example Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ **(do, if, int, while)**. Let $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i) = q(i), c(i,i) = 0$ and $r(i,i) = 0, 0 \leq i \leq 4$. the observation $w(i,j) = p(j) + q(j) + w(i,j-1)$, we get

Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ (**do, if, int, while**). Let $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

$$
\begin{aligned}
w(0,1) &= p(1) + q(1) + w(0,0) = 8 \\
c(0,1) &= w(0,1) + \min\{c(0,0) + c(1,1)\} = 8 \\
r(0,1) &= 1 \\
w(1,2) &= p(2) + q(2) + w(1,1) = 7 \\
c(1,2) &= w(1,2) + \min\{c(1,1) + c(2,2)\} = 7 \\
r(0,2) &= 2 \\
w(2,3) &= p(3) + q(3) + w(2,2) = 3 \\
c(2,3) &= w(2,3) + \min\{c(2,2) + c(3,3)\} = 3 \\
r(2,3) &= 3 \\
w(3,4) &= p(4) + q(4) + w(3,3) = 3 \\
c(3,4) &= w(3,4) + \min\{c(3,3) + c(4,4)\} = 3 \\
r(3,4) &= 4
\end{aligned}
$$

Optimal search tree



Module-IV
Graph Traversal Algorithms

Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

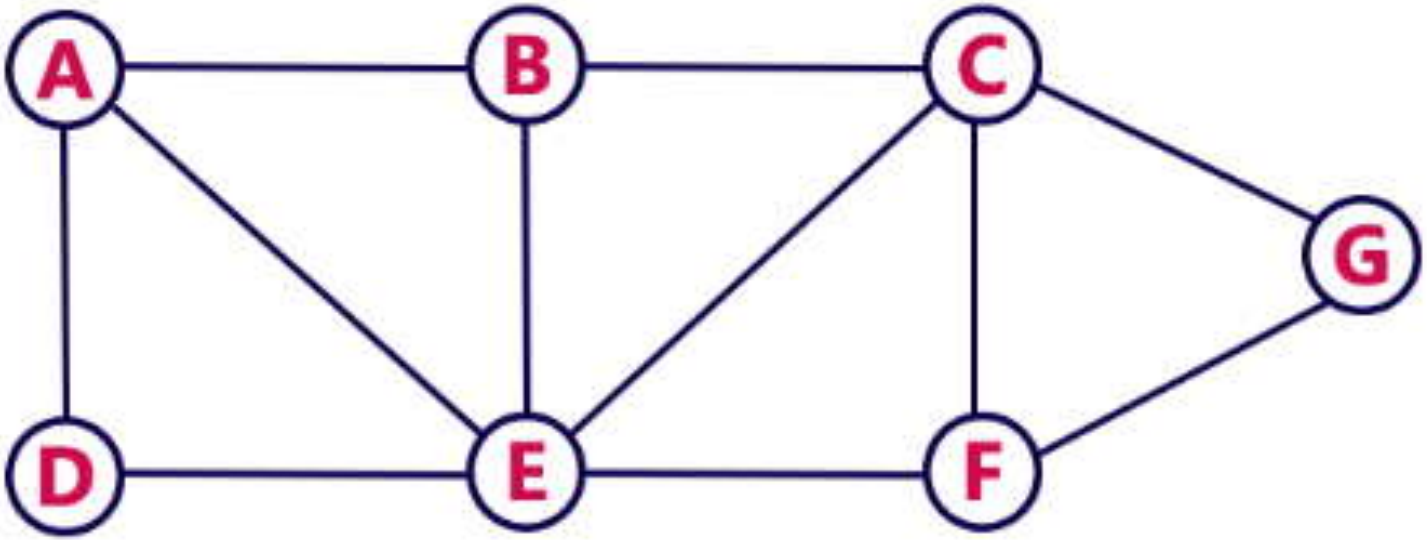
Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

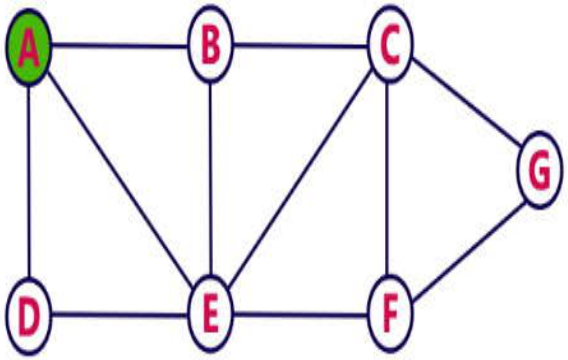
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform DFS traversal



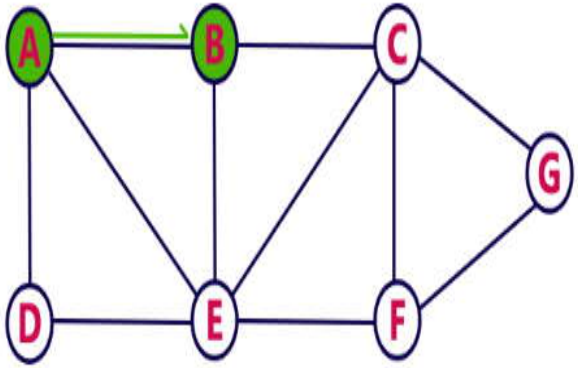
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



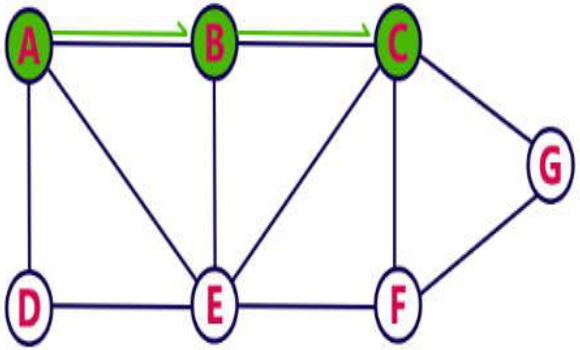
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



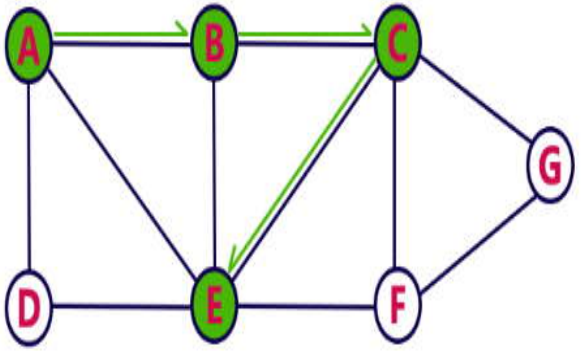
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



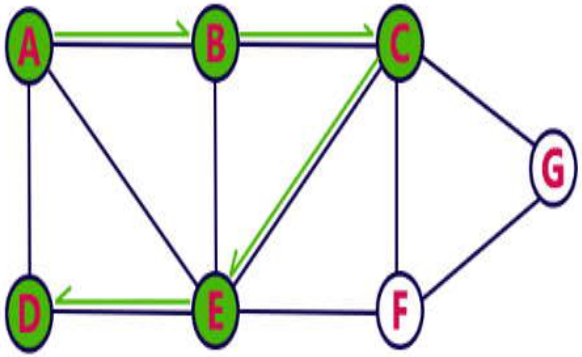
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



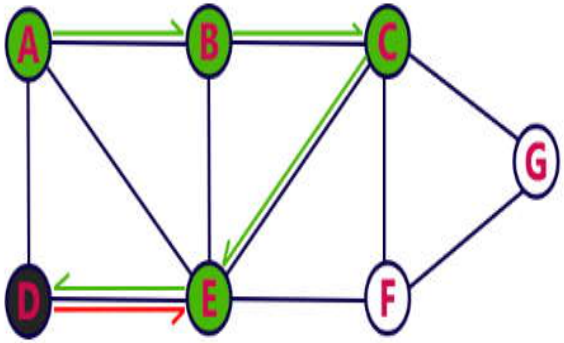
Step 5:

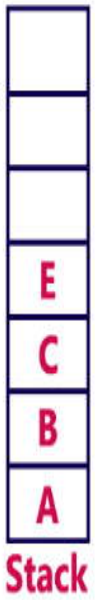
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

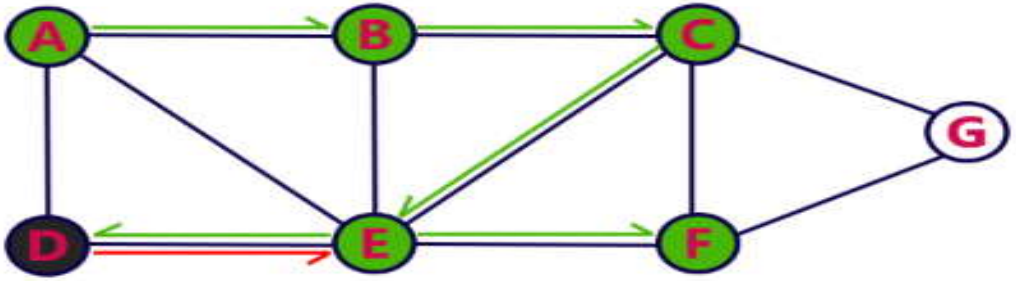
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.





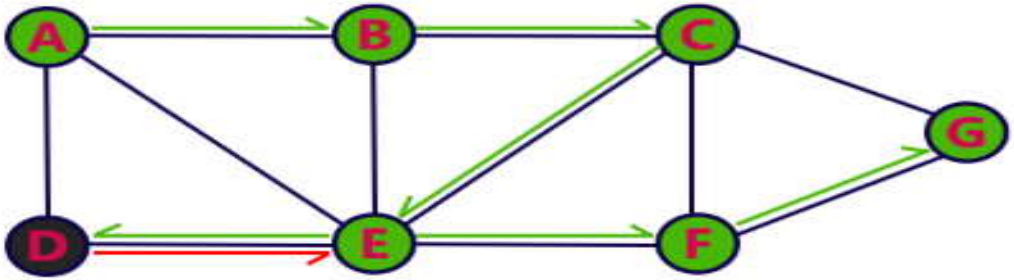
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



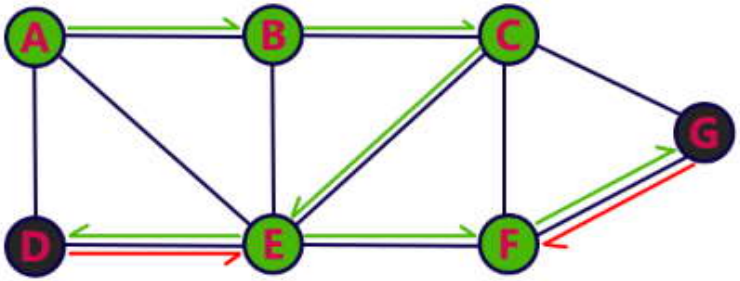
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



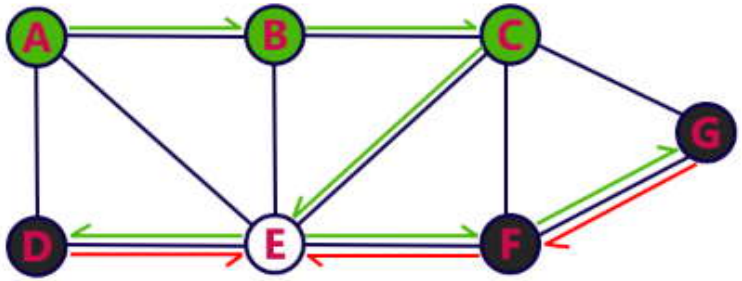
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



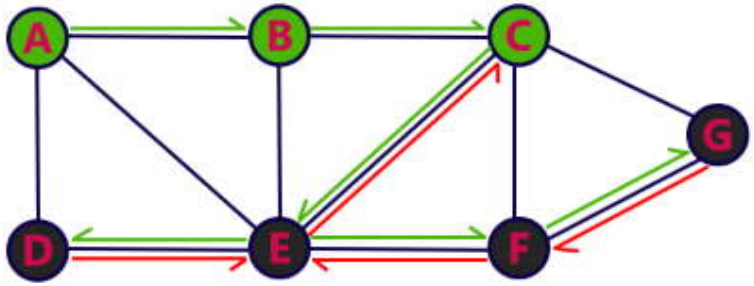
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



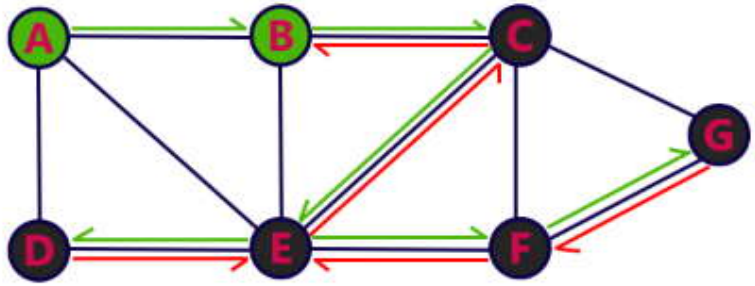
Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



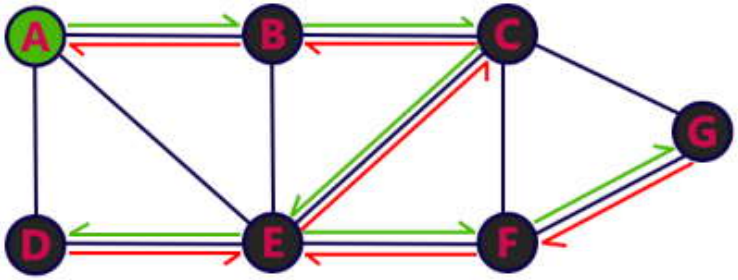
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



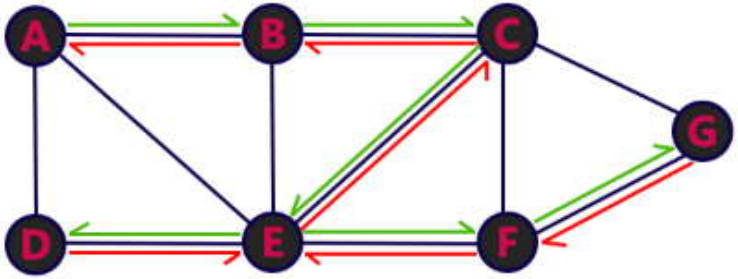
Step 13:

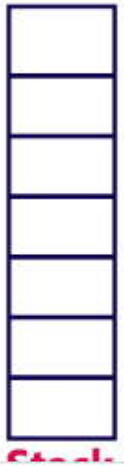
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





```
1   Algorithm DFS(v)
2   // Given an undirected (directed) graph G = (V, E) with
3   // n vertices and an array visited[ ] initially set
4   // to zero, this algorithm visits all vertices
5   // reachable from v. G and visited[ ] are global.
6   {
7       visited[v] := 1;
8       for each vertex w adjacent from v do
9       {
10          if (visited[w] = 0) then DFS(w);
11      }
12  }
```

BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

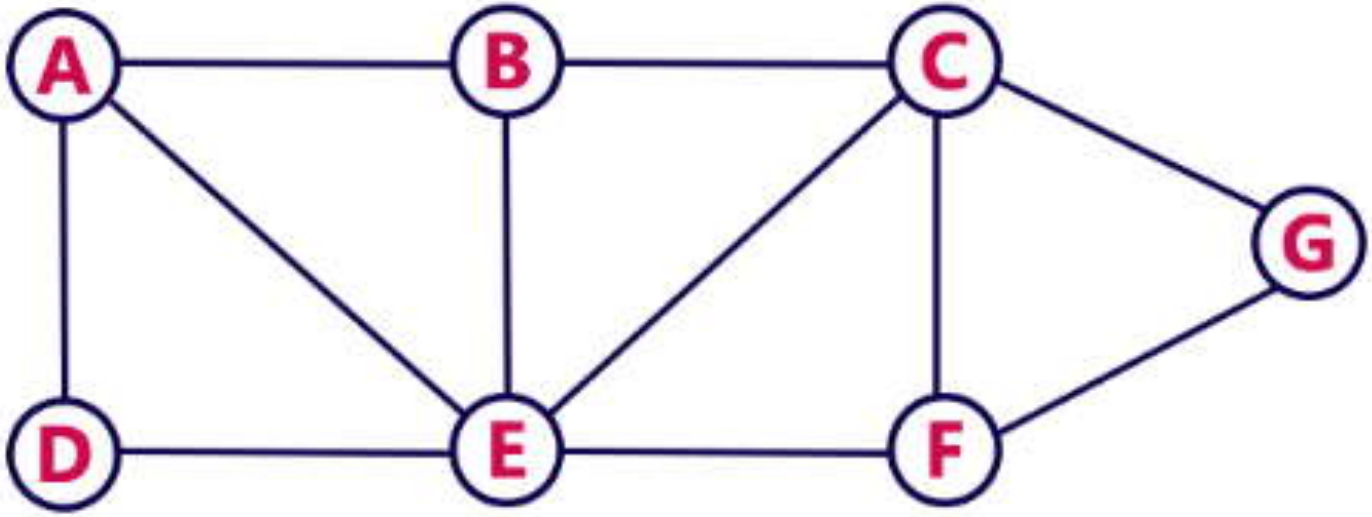
Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

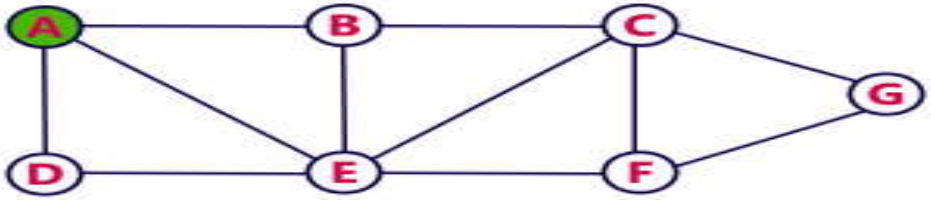
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

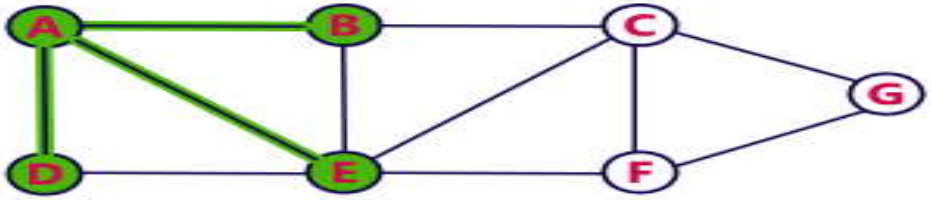


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

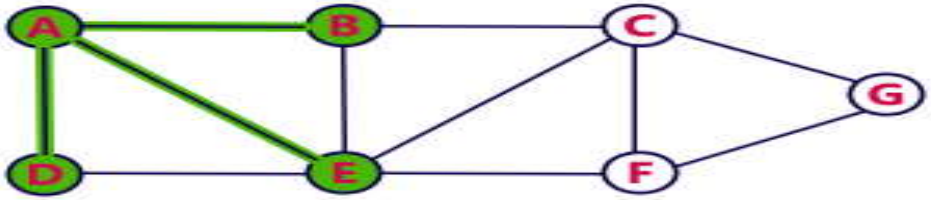


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

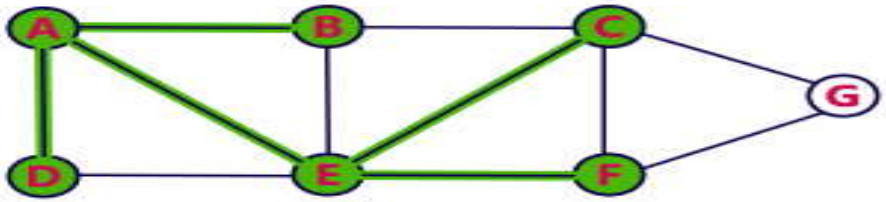


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

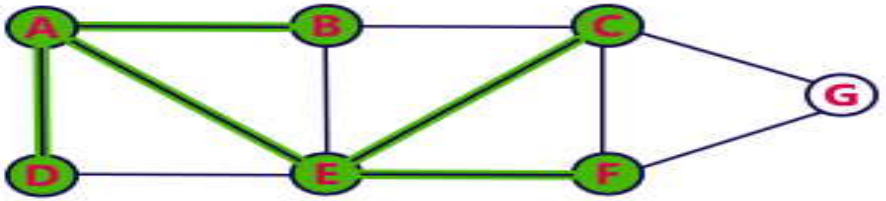


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

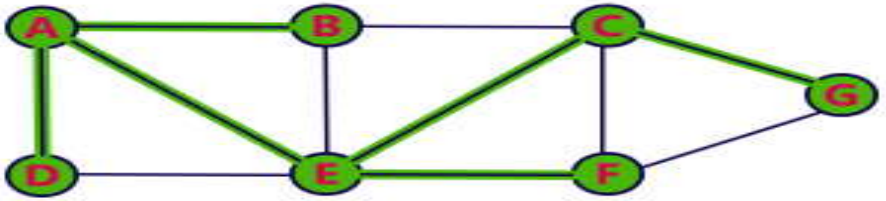


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

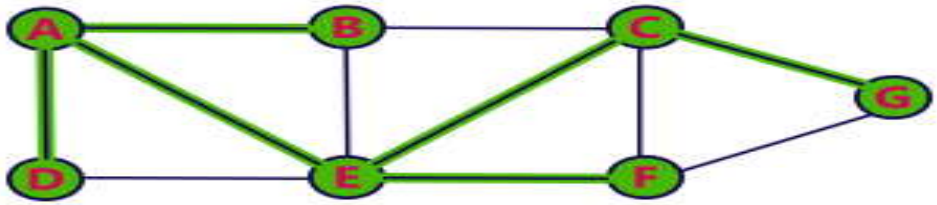


Queue

					F	G

Step 7:

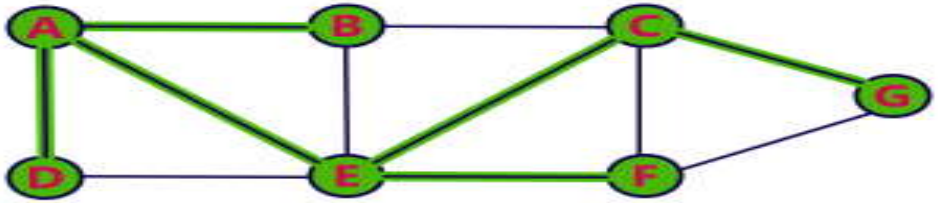
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue

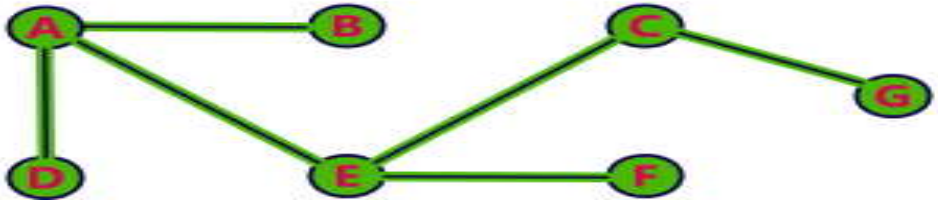
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



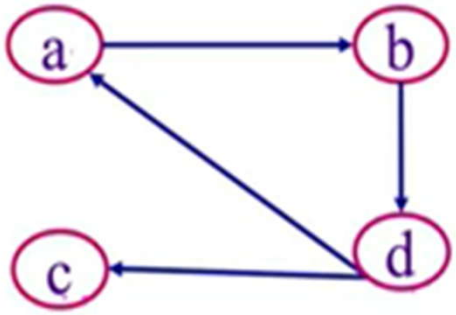
```
1    Algorithm BFS(v)
2    // A breadth first search of G is carried out beginning
3    // at vertex v. For any node i, visited[i] = 1 if i has
4    // already been visited. The graph G and array visited[ ]
5    // are global; visited[ ] is initialized to zero.
6    {
7         u := v; // q is a queue of unexplored vertices.
8         visited[v] := 1;
9         repeat
10        {
11             for all vertices w adjacent from u do
12             {
13                  if (visited[w] = 0) then
14                  {
15                       Add w to q; // w is unexplored.
16                       visited[w] := 1;
17                  }
18             }
19             if q is empty then return; // No unexplored vertex.
20             Delete u from q; // Get first unexplored vertex.
21        } until(false);
22   }
```

Algorithm 6.5 Pseudocode for breadth first search

Warshall's Algorithm(Transitive closure)

DEFINITION The *transitive closure* of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

- Named after Stephen Warshall, who discovered this algorithm
- To determine Transitive Closure of a Directed graph or all paths in a directed graph using adjacency matrix



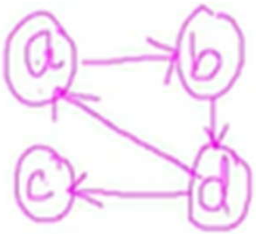
- To check whether there is an existence of path between every pair of vertices

Step 1: Generate Adjacency Matrix

$R^{(0)} =$

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

Step 2: Consider path through vertex a



$R^{(0)} =$

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

$R^{(1)} =$

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

$b \to b = b \to a \;\&\; a \to b = 0 \;\&\; 1 = 0$

$b \to c = b \to a \;\&\; a \to c = 0 \;\&\; 0 = 0$

$c \to b = c \to a \;\&\; a \to b = 0 \;\&\; 1 = 0$

$c \to c = c \to a \;\&\; a \to c = 0 \;\&\; 0 = 0$

$c \to d = c \to a \;\&\; a \to d = 0 \;\&\; 0 = 0$

$d \to b = d \to a \;\&\; a \to b = 1 \;\&\; 1 = 1$

$d \to d = d \to a \;\&\; a \to d = 1 \;\&\; 0 = 0$

Step 3: Consider path through vertex b

$R^{(1)} =$

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

$R^{(2)} =$

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

$a \to a = a \to b \,\&\, b \to a = 1 \,\&\, 0 = 0$

$a \to c = a \to b \,\&\, b \to c = 1 \,\&\, 0 = 0$

$a \to d = a \to b \,\&\, b \to d = 1 \,\&\, 1 = 1$

$c \to a = c \to b \,\&\, b \to a = 0 \,\&\, 0 = 0$

$c \to c = c \to b \,\&\, b \to c = 0 \,\&\, 0 = 0$

$c \to d = c \to b \,\&\, b \to d = 0 \,\&\, 1 = 0$

$d \to d = d \to b \,\&\, b \to d = 1 \,\&\, 1 = 1$

Step 4: Consider path through vertex c

$R^{(2)} =$

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

$R^{(3)} =$

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

$a \to a = a \to c \,\&\, c \to a = 0 \,\&\, 0 = 0$

$b \to a = b \to c \,\&\, c \to a = 0 \,\&\, 0 = 0$

$b \to b = b \to c \,\&\, c \to b = 0 \,\&\, 0 = 0$

Step 5: Consider path through vertex d

$R^{(3)} =$

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

$R^{(4)} =$

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

$a \to a = a \to d \,\&\, d \to a = 1 \,\&\, 1 = 1$

$a \to c = a \to d \,\&\, d \to c = 1 \,\&\, 1 = 1$

$b \to a = b \to d \,\&\, d \to a = 1 \,\&\, 1 = 1$

$b \to b = b \to d \,\&\, d \to b = 1 \,\&\, 1 = 1$

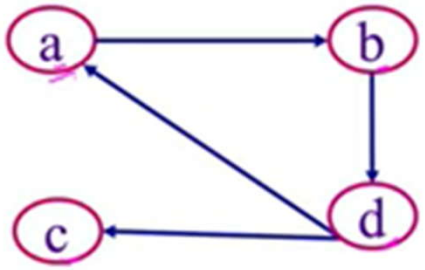
$b \to c = b \to d \,\&\, d \to c = 1 \,\&\, 1 = 1$

$c \to a = c \to d \,\&\, d \to a = 0 \,\&\, 1 = 0$

$c \to b = c \to d \,\&\, d \to b = 0 \,\&\, 1 = 0$

$c \to c = c \to d \,\&\, d \to c = 0 \,\&\, 1 = 0$

- **Adjacency Matrix**



Transitive closure / path Matrix

$P^{(4)} =$

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

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
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)
```

Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$