**UNIT 3**

**(2 MARKS)**

**1.What are greedy algorithms? Explain their characteristics?**

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

following are the characteristics of a greedy method:

* To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
* A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

**2.Define feasible and optimal solution.**

The maximization or minimization of some quantity is the objective in all linear programming problems. A feasible solution satisfies all the problem's constraints. An optimal solution is a feasible solution that results in the largest possible objective function value when maximizing (or smallest when minimizing).

First of all Feasible means output that you expect from your solution of a particular problem and you are getting the exact output.

Optimal means for every worst scenario it should work and gives me the output in minimum possible time and also consuming less space.

Now Feasible solution means you know how to code the program for a particular problem and when you run it the results that you are expecting, gets the exact results from it. But you didnt consider the time and space complexity for it.

Optimal solution means getting the exact results also takes less time and space to solve a particular problem.

**3.Explain Single source shortest path.**

n a shortest- paths problem, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v0,v1,..... vk) is the total of the weights of its constituent edges:

Single Source Shortest Paths

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The shortest path from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

**4.Define principle of optimality.**

A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themesleves optimal solutions for their subproblems.

Examples:

1.The shortest path problem satisfies the Principle of Optimality.

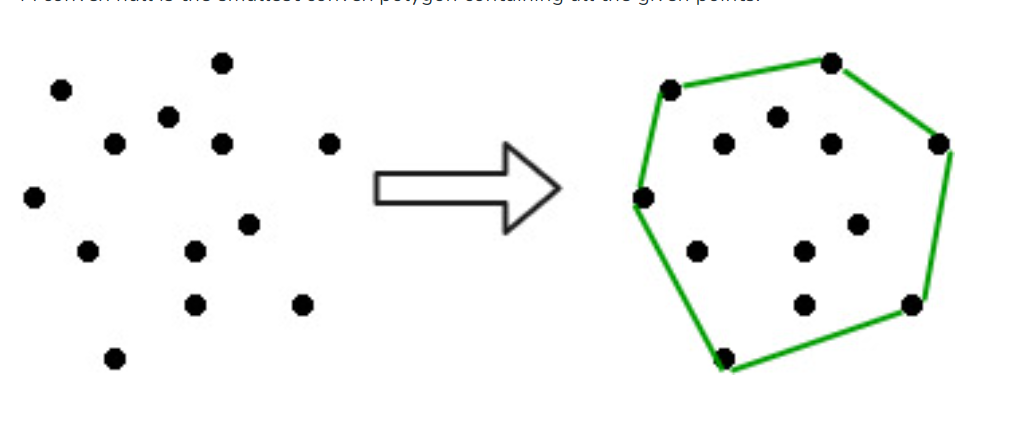
2.This is because if a,x1,x2,...,xn,b is a shortest path from node a to node b in a graph, then the portion of xi to xj on that path is a shortest path from xi to xj.

3.The longest path problem, on the other hand, does not satisfy the Principle of Optimality. Take for example the undirected graph G of nodes a, b, c, d, and e, and edges (a,b) (b,c) (c,d) (d,e) and (e,a). That is, G is a ring. The longest (noncyclic) path from a to d to a,b,c,d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather, b,a,e,d,c is the longest path. Thus, the subpath on a longest path is not necessarily a longest path.

**5.What do you mean by convex hull?**

A convex hull of a set of points is defined as the smallest convex polygon containing all the points. In other words, it is the outer boundary of the set of points that forms a shape with no indentations or concave portions.

The convex hull can be represented as a polygon formed by connecting the outermost points counterclockwise or clockwise. It can also be described as the intersection of all convex sets containing the given points.



### Applications

The convex hull has various applications, such as:

* **Computational geometry**: It is used in algorithms for solving problems like finding the closest pair of points or solving linear programming problems.
* **Image processing:** Convex hulls can be used to analyze and recognize image shapes, particularly for object recognition or tracking.
* **Robotics:** Convex hulls are useful for collision detection and path planning in robotics applications.
* **Game development:** Convex hulls are employed in physics engines for collision detection and response between objects in games.

**6.Compare dynamic and greedy programming strategies.**

| **Feature** | **Greedy method** | **Dynamic programming** |
| --- | --- | --- |
| **Feasibility** | In a [greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms/), we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution. | In [Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/) we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution . |
| **Optimality** | In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution. | It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best. |
| **Recursion** | A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage. | A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states. |
| **Memoization** | It is more efficient in terms of memory as it never look back or revise previous choices | It requires Dynamic Programming table for Memoization and it increases it’s memory complexity. |
| **Time        complexity** | Greedy methods are generally faster. For example, [Dijkstra’s shortest path](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) algorithm takes O(ELogV + VLogV) time. | Dynamic Programming is generally slower. For example, [Bellman Ford algorithm](https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/) takes O(VE) time. |
| **Fashion** | The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices. | Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions. |
| **Example** | Fractional knapsack . | 0/1 knapsack problem |

**7.Describe Activity selection problem.**

he activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well-designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** si and a **finish time** fi, where si ≤fi. If selected activity "i" take place meanwhile the half-open time interval [si,fi). Activities i and j are **compatible** if the intervals (si, fi) and [si, fi) do not overlap (i.e. i and j are compatible if si ≥fi or si ≥fi). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

**GREEDY- ACTIVITY SELECTOR (s, f)**

1. n ← length [s]

2. A ← {1}

3. j ← 1.

4. for i ← 2 to n

5. do if si ≥ fi

6. then A ← A ∪ {i}

7. j ← i

8. return A

**8.How BFS is differ from DFS.**

|  |  |  |
| --- | --- | --- |
| **Key** | **BFS** | **DFS** |
| Definition | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| Data structure | BFS uses a Queue to find the shortest path. | DFS uses a Stack to find the shortest path. |
| Source | BFS is better when target is closer to Source. | DFS is better when target is far from source. |
| Suitability for decision tree | As BFS considers all neighbor so it is not suitable for decision tree used in puzzle games. | DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won. |
| Speed | BFS is slower than DFS. | DFS is faster than BFS. |
| Time Complexity | Time Complexity of BFS = O(V+E) where V is vertices and E is edges. | Time Complexity of DFS is also O(V+E) where V is vertices and E is edges. |
| Memory | BFS requires more memory space. | DFS requires less memory space. |
| Tapping in loops | In BFS, there is no problem of trapping into finite loops. | In DFS, we may be trapped into infinite loops. |
| Principle | BFS is implemented using FIFO (First In First Out) principle. | DFS is implemented using LIFO (Last In First Out) principle. |

**9.Define principal of optimality. When and how dynamic programming is applicable.**

The principle of optimality is a fundamental aspect of dynamic programming, which states that the optimal solution to a dynamic optimization problem can be found by combining the optimal solutions to its sub-problems. While this principle is generally applicable, it is often only taught for problems with finite or countable state spaces in order to sidestep measure-theoretic complexities. Therefore, it cannot be applied to classic models such as inventory management and dynamic pricing models that have continuous state spaces, and students may not be aware of the possible challenges involved in studying dynamic programming models with general state spaces. To address this, we provide conditions and a self-contained simple proof that establish when the principle of optimality for discounted dynamic programming is valid. These conditions shed light on the difficulties that may arise in the general state space case. We provide examples from the literature that include the relatively involved case of universally measurable dynamic programming and the simple case of finite dynamic programming where our main result can be applied to show that the principle of optimality holds.

**(10 MARKS)**

**1.Discuss greedy approach to an activity selection problem of scheduling several competing activities. Solve following activity selection problem S = {A1, A2, A3, A4, A5, A6, A7, A8, A9, A10} Si = {1, 2, 3, 4, 7, 8, 9, 9, 11, 12} Fi = {3, 5, 4, 7, 10, 9, 11, 13, 12, 14}.**

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well-designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** si and a **finish time** fi, where si ≤fi. If selected activity "i" take place meanwhile the half-open time interval [si,fi). Activities i and j are **compatible** if the intervals (si, fi) and [si, fi) do not overlap (i.e. i and j are compatible if si ≥fi or si ≥fi). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

Algorithm Of Greedy- Activity Selector:

**GREEDY- ACTIVITY SELECTOR (s, f)**

1. n ← length [s]

2. A ← {1}

3. j ← 1.

4. for i ← 2 to n

5. do if si ≥ fi

6. then A ← A ∪ {i}

7. j ← i

8. return A

**Example:** Given 10 activities along with their start and end time as

S = (A1 A2 A3 A4 A5 A6 A7 A8 A9 A10)

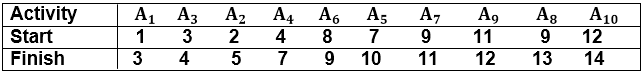
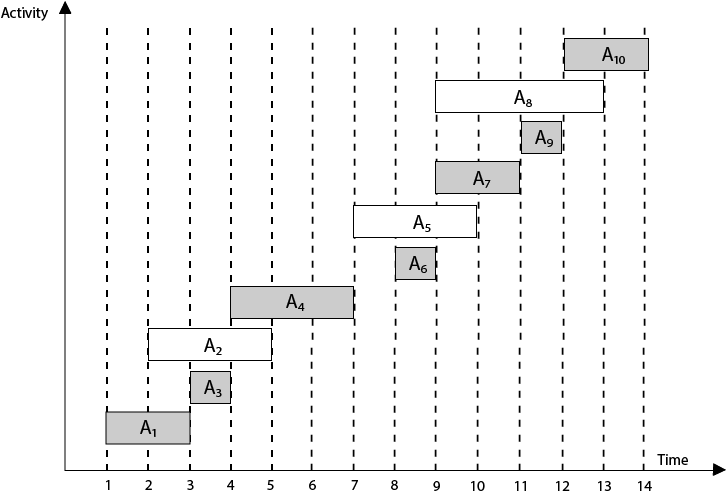
Si = (1,2,3,4,7,8,9,9,11,12)

fi = (3,5,4,7,10,9,11,13,12,14)

Compute a schedule where the greatest number of activities takes place.

**Solution:** The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:

Arranging the activities in increasing order of end time

Now, schedule A1

Next schedule A3 as A1 and A3 are non-interfering.

Next **skip** A2 as it is interfering.

Next, schedule A4 as A1 A3 and A4 are non-interfering, then next, schedule A6 as A1 A3 A4 and A6 are non-interfering.

Skip A5 as it is interfering.

Next, schedule A7 as A1 A3 A4 A6 and A7 are non-interfering.

Next, schedule A9 as A1 A3 A4 A6 A7 and A9 are non-interfering.

Skip A8 as it is interfering.

Next, schedule A10 as A1 A3 A4 A6 A7 A9 and A10 are non-interfering.

Thus the final Activity schedule is:

Activity Selection Problem

**2.Explain “greedy algorithm” Write its pseudo code to prove that fractional Knapsack problem has a greedy-choice property.**

# A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment.

# **Fractional Knapsack Problem**

Given the weights and profits of**N** items, in the form of **{profit, weight}** put these items in a knapsack of capacity **W** to get the maximum total profit in the knapsack. In **Fractional Knapsack**, we can break items for maximizing the total value of the knapsack.

***Input:****arr[]={{60, 10}, {100, 20}, {120, 30}}, W = 50****Output:****240****Explanation:***By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.*Hence total price will be 60+100+(2/3)(120) = 240*

***Input:****arr[]={{500,30}},W = 10****Output:****166.667*

## Fractional Knapsack Problem using [Greedy algorithm](https://www.geeksforgeeks.org/greedy-algorithms/):An efficient solution is to use the Greedy approach.

The basic idea of the greedy approach is to calculate the ratio **profit/weight** for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).

This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

**Illustration:**

Check the below illustration for a better understanding:

*Consider the example:****arr[] = {{100, 20}, {60, 10}, {120, 30}}, W = 50****.*

***Sorting:****Initially sort the array based on the profit/weight ratio. The sorted array will be****{{60, 10}, {100, 20}, {120, 30}}****.*

***Iteration:***

* For **i = 0**, weight = 10 which is less than W. So add this element in the knapsack. **profit = 60** and remaining **W = 50 – 10 = 40**.
* For **i = 1**, weight = 20 which is less than W. So add this element too. **profit = 60 + 100 = 160** and remaining **W = 40 – 20 = 20**.
* For **i = 2**, weight = 30 is greater than W. So add 20/30 fraction = **2/3** fraction of the element. Therefore **profit** = 2/3 \* 120 + 160 = 80 + 160 = **240** and remaining **W** becomes **0**.

So the final profit becomes **240** for **W = 50**.

Follow the given steps to solve the problem using the above approach:

* Calculate the ratio (**profit/weight**) for each item.
* Sort all the items in decreasing order of the ratio.
* Initialize **res = 0**, curr\_cap = given\_cap.
* Do the following for every item **i** in the sorted order:
  + If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
  + Else add the current item as much as we can and break out of the loop.
* Return **res**.

**Time Complexity:** O(N \* logN)  
**Auxiliary Space:** O(N)

3.**Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyse its time complexity.**

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.But before we understand the problem, let us understand what the term subsequence is −

Let us consider a sequence S = <s1, s2, s3, s4, …,sn>. And another sequence Z = <z1, z2, z3, …,zm> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements. In simple words, a subsequence consists of consecutive elements that make up a small part in a sequence.

### Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

### Naïve Method

Let ***X*** be a sequence of length m and ***Y*** a sequence of length n. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***, and return the longest common subsequence found.

There are **2m** subsequences of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes O(n) time. Thus, the naïve algorithm would take O(n2m) time.

## **Longest Common Subsequence Algorithm**

Let X=<x1,x2,x3....,xm> and Y=<y1,y2,y3....,ym> be the sequences. To compute the length of an element the following algorithm is used.

**Step 1** − Construct an empty adjacency table with the size, n × m, where n = size of sequence **X** and m = size of sequence **Y**. The rows in the table represent the elements in sequence X and columns represent the elements in sequence Y.

**Step 2** − The zeroeth rows and columns must be filled with zeroes. And the remaining values are filled in based on different cases, by maintaining a counter value.

* **Case 1** − If the counter encounters common element in both X and Y sequences, increment the counter by 1.
* **Case 2** − If the counter does not encounter common elements in X and Y sequences at T[i, j], find the maximum value between T[i-1, j] and T[i, j-1] to fill it in T[i, j].

**Step 3** − Once the table is filled, backtrack from the last value in the table. Backtracking here is done by tracing the path where the counter incremented first.

**Step 4** − The longest common subseqence obtained by noting the elements in the traced path.

### Pseudocode

In this procedure, table **C[m, n]** is computed in row major order and another table **B[m,n]** is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

m := length(X)

n := length(Y)

for i = 1 to m do

C[i, 0] := 0

for j = 1 to n do

C[0, j] := 0

for i = 1 to m do

for j = 1 to n do

if xi = yj

C[i, j] := C[i - 1, j - 1] + 1

B[i, j] := ‘D’

else

if C[i -1, j] ≥ C[i, j -1]

C[i, j] := C[i - 1, j] + 1

B[i, j] := ‘U’

else

C[i, j] := C[i, j - 1] + 1

B[i, j] := ‘L’

return C and B

Algorithm: Print-LCS (B, X, i, j)

if i=0 and j=0

return

if B[i, j] = ‘D’

Print-LCS(B, X, i-1, j-1)

Print(xi)

else if B[i, j] = ‘U’

Print-LCS(B, X, i-1, j)

else

Print-LCS(B, X, i, j-1)

This algorithm will print the longest common subsequence of **X** and **Y**.

### Analysis

To populate the table, the outer for loop iterates m times and the inner **for** loop iterates **n** times. Hence, the complexity of the algorithm is **O(m,n)**, where **m** and **n** are the length of two strings.

### Example

In this example, we have two strings ***X=BACDB*** and ***Y=BDCB*** to find the longest common subsequence.

Following the algorithm, we need to calculate two tables 1 and 2.

Given n = length of X, m = length of Y

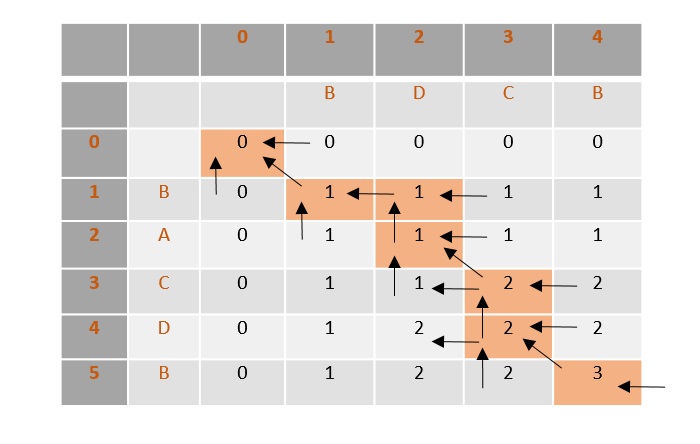
X = BDCB, Y = BACDB

### Constructing the LCS Tables

In the table below, the zeroeth rows and columns are filled with zeroes. Remianing values are filled by incrementing and choosing the maximum values according to the algorithm.



Once the values are filled, the path is traced back from the last value in the table at T[4, 5].

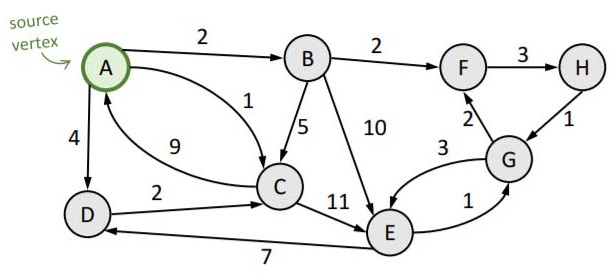


From the traced path, the longest common subsequence is found by choosing the values where the counter is first incremented.

In this example, the final count is 3 so the counter is incremented at 3 places, i.e., B, C, B. Therefore, the longest common subsequence of sequences X and Y is BCB.

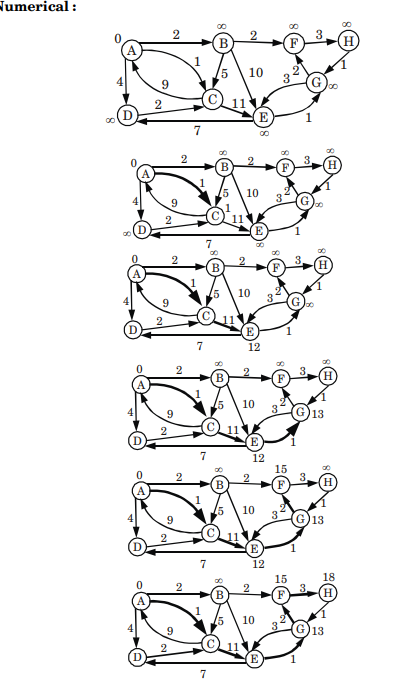
**4.When do Dijkstra and the Bellman-Ford algorithm both fail to find a shortest path?Can Bellman ford detect all negative weight cycles in a graph?Apply Bellman Ford Algorithm on the following graph:**

KUMAR



Both algorithms will not find a shortest path if the graph contains a negative cycle and this cycle is reachable from the source node and the destination node is reachable from the cycle. In this case there is no shortest path - you may perform infinitely many iteration over the cycle always reducing the path length.

If we continue to go around the negative cycle an infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing). As a result, Bellman-Ford is also capable of detecting negative cycles, which is an important feature.



**5.Prove that if the weights on the edge of the connected undirected graph are distinct then there is a unique Minimum Spanning Tree. Give an example in this regard. Also discuss Kruskal’s Minimum Spanning Tree in detail.**

1. Say we have an algorithm that finds an MST (which we will call A) based on the structure of the graph and the order of the edges when ordered by weight.

2. Assume MST A is not unique.

3. There is another spanning tree with equal weight, say MST B.

4. Let e1 be an edge that is in A but not in B.

5. Then B should include at least one edge e2 that is not in A.

6. Assume the weight of e1 is less than that of e2.

7. As B is a MST, {e1} B must contain a cycle.

8. Replace e2 with e1 in B yields the spanning tree {e1} B

**MST using Kruskal’s algorithm**

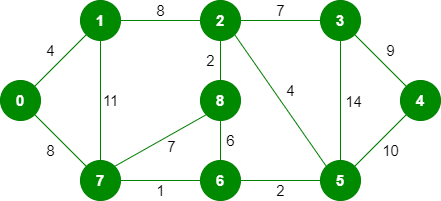
Below are the steps for finding MST using Kruskal’s algorithm:

1.Sort all the edges in non-decreasing order of their weight.

2.Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.

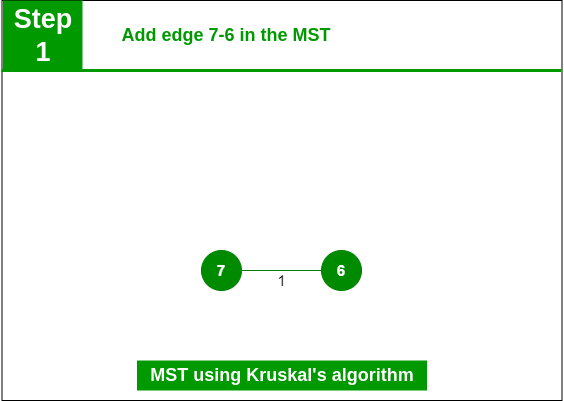
3.Repeat step#2 until there are (V-1) edges in the spanning tree.

***InputGraph:***

1. 

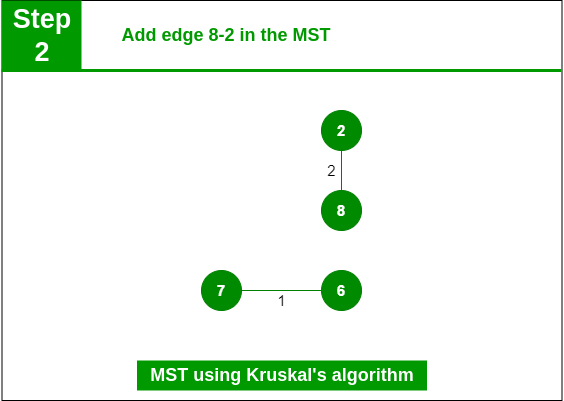
*The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.*

***Step 1:****Pick edge 7-6. No cycle is formed, include it.*



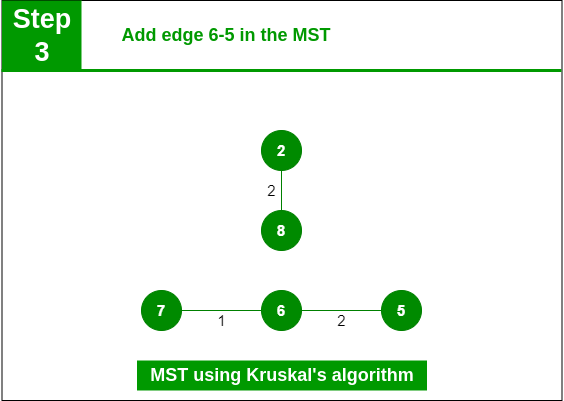
*Add edge 7-6 in the MST*

***Step 2:****Pick edge 8-2. No cycle is formed, include it.*

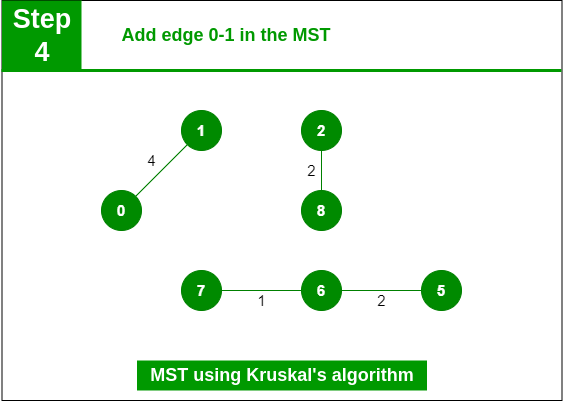


*Add edge 8-2 in the MST*

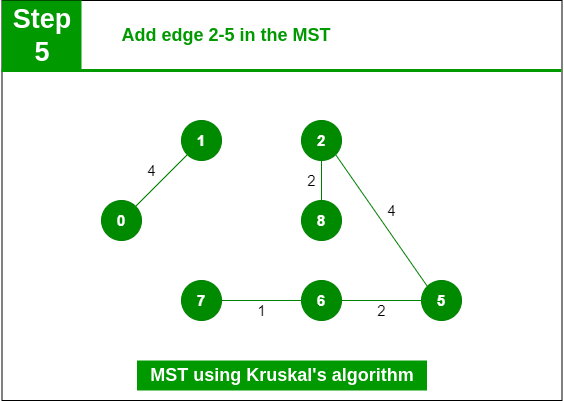
***Step 3:****Pick edge 6-5. No cycle is formed, include it.*



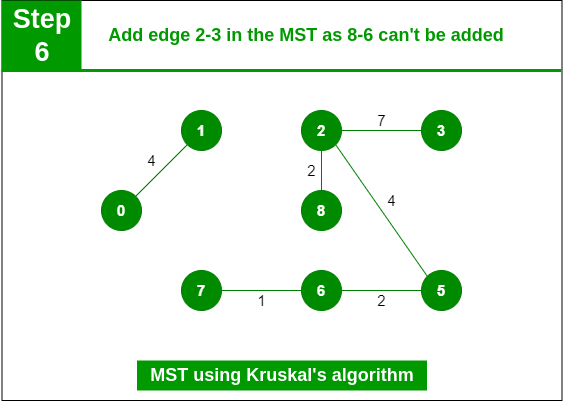
***Step 4:****Pick edge 0-1. No cycle is formed, include it.*



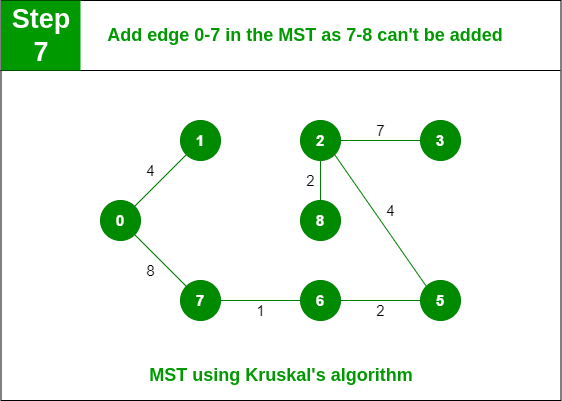
***Step 5:****Pick edge 2-5. No cycle is formed, include it.*



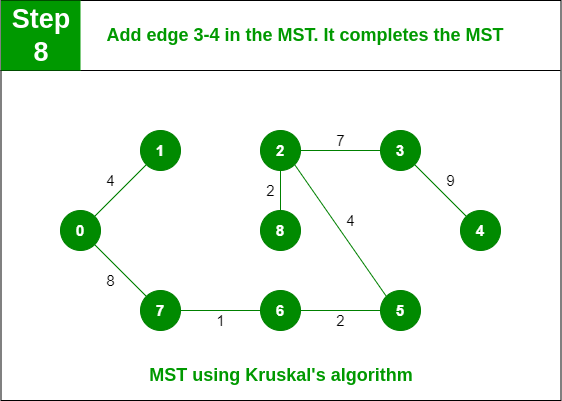
***Step 6:****Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.*



***Step 7:****Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.*



***Step 8:****Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.*



6.**Discuss Prim’s Minimum Spanning Tree Algorithm in detail.**

**Spanning tree -** A spanning tree is the subgraph of an undirected connected graph.

**Minimum Spanning tree -** Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Now, let's start the main topic.

**Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

## **How does the prim's algorithm work?**

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

* First, we have to initialize an MST with the randomly chosen vertex.
* Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
* Repeat step 2 until the minimum spanning tree is formed.

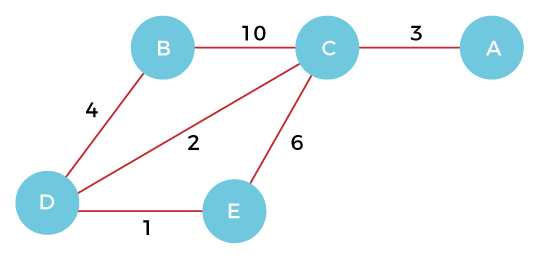
The applications of prim's algorithm are -

* Prim's algorithm can be used in network designing.
* It can be used to make network cycles.
* It can also be used to lay down electrical wiring cables.

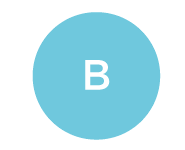
## **Example of prim's algorithm**

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

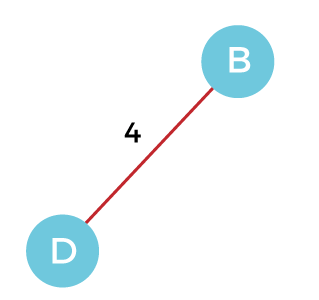
Suppose, a weighted graph is -



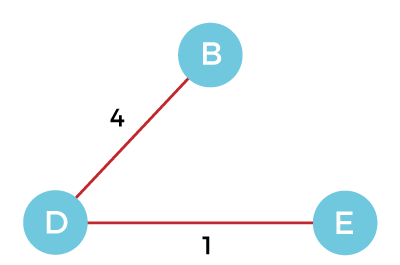
**Step 1 -** First, we have to choose a vertex from the above graph. Let's choose B.



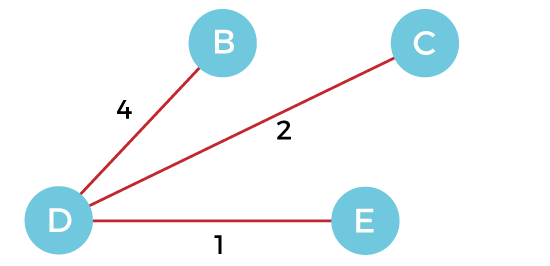
**Step 2 -** Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



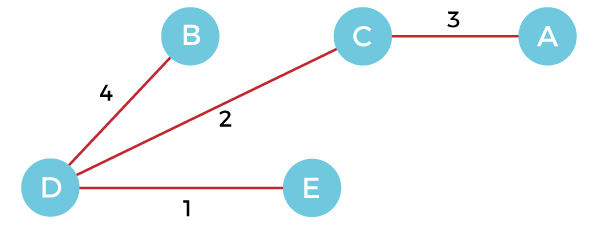
**Step 3 -** Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



**Step 4 -** Now, select the edge CD, and add it to the MST.



**Step 5 -** Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

## **Algorithm**

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

## **Complexity of Prim's algorithm**

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

**Time Complexity**

|  |  |
| --- | --- |
| **Data structure used for the minimum edge weight** | **Time Complexity** |
| Adjacency matrix, linear searching | O(|V|2) |
| Adjacency list and binary heap | O(|E| log |V|) |
| Adjacency list and Fibonacci heap | O(|E|+ |V| log |V|) |

Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm. The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

7**.Define spanning tree. Write Kruskal’s algorithm for finding minimum cost spanning tree. Describe how Kruskal’s algorithm is different from Prim’s algorithm for finding minimum cost spanning tree.**

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edges may or may not have weights assigned to them.

# **Methods of Minimum Spanning Tree**

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

**Kruskal's Algorithm:**

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

**If the graph is not linked, then it finds a Minimum Spanning Tree.**

**Steps for finding MST using Kruskal's Algorithm:**

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

**MST- KRUSKAL (G, w)**

1. A ← ∅

2. for each vertex v ∈ V [G]

3. do MAKE - SET (v)

4. sort the edges of E into non decreasing order by weight w

5. for each edge (u, v) ∈ E, taken in non decreasing order by weight

6. do if FIND-SET (μ) ≠ if FIND-SET (v)

7. then A ← A ∪ {(u, v)}

8. UNION (u, v)

9. return A

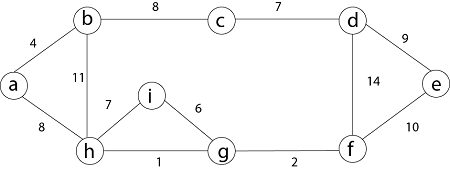
**Analysis:** Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

* E is at most V2 and log V2= 2 x log V is O (log V).
* If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

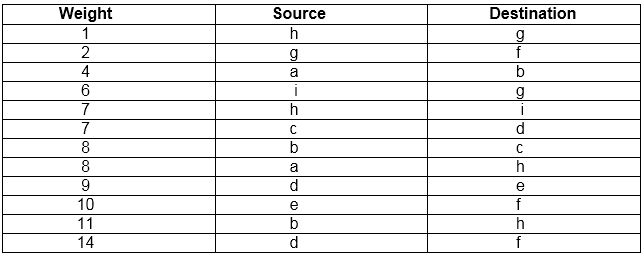
1. O (E log E) = O (E log V).

**For Example:** Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



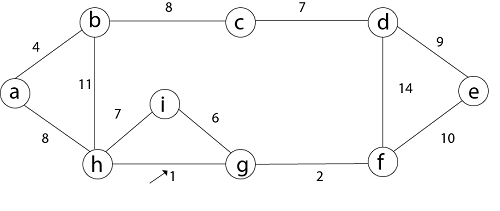
First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

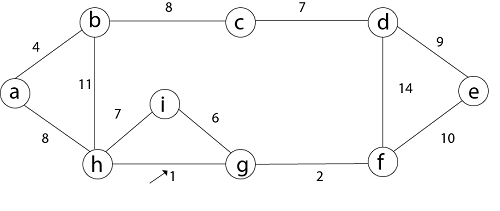


Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

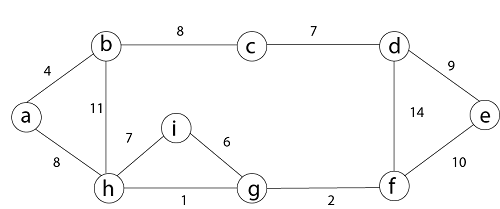
**Step1:** So, first take (h, g) edge



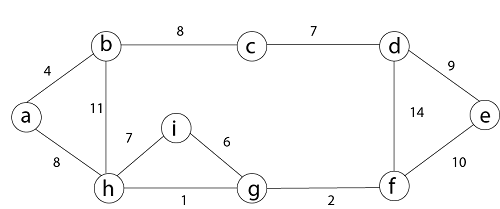
**Step 2:** then (g, f) edge.



**Step 3:** then (a, b) and (i, g) edges are considered, and the forest becomes

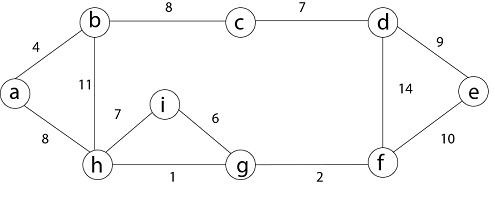


**Step 4:** Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



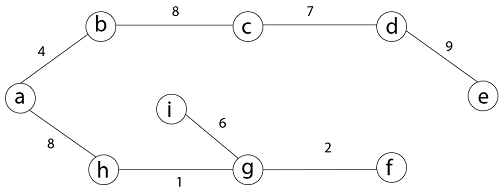
**Step 5:** In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

**Step 6:** After that edge (d, f) and the final spanning tree is shown as in dark lines.



**Step 7:** This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f,  b → h,  d → f [cycle will be formed]



Both Prim’s and Kruskal’s algorithms are developed for discovering the minimum spanning tree of a graph. Both the algorithms are popular and follow different steps to solve the same kind of problem.The prim’s algorithm selects the root vertex in the beginning and then traverses from vertex to vertex adjacently. On the other hand, Krushal’s algorithm helps in generating the minimum spanning tree, initiating from the smallest weighted edge.

**8.Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by Vopt. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by Vgreedy. Find the value of Vopt − Vgreedy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item** | **I1** | **I2** | **I3** | **I4** |
| **W** | **10** | **7** | **4** | **2** |
| **V** | **60** | **28** | **20** | **24** |

This question is about Knapsack problem with greedy and dynamic solution with some constraints.

Total weight = 11 Kgs

Constraint is : Items should not split [either taken full to level it]

So, Vopt = Item1(10)

= 60 Rs

Vgreedy :

Item number Weight in (kgs) Value in (rupees) Value/weight

1 10 60 60/10=6

2 7 28 28/7 = 4

3 4 20 20/4 = 5

4 2 24 24/2 = 12

Sorting in descending order :

Item 4 1 3 2

Value / weight 12 6 5 4

So, item picked = Item4+Item3

= 24 + 20

= 44 Rs

So, the value of Vopt - Vgreedy

= 60 Rs - 44 Rs

= 16 Rs

**9.What are single source shortest paths? Write down Dijkstra’s algorithm for it.**

The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph. Algorithms such as Breadth-First-Search (BFS) for unweighted graphs or Dijkstra [1] solve this problem.

**Dijkstra's Algorithm** is a Graph algorithm **that finds the shortest path** from a source vertex to all other vertices in the Graph (single source shortest path). It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is **O(V2)** with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to **O((V + E) log V)** with the help of an adjacency list representation of the graph, where **V** is the number of vertices and **E** is the number of edges in the graph.

## **Dijkstra's Algorithm :**

**The following is the step that we will follow to implement Dijkstra's Algorithm:**

**Step 1:** First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

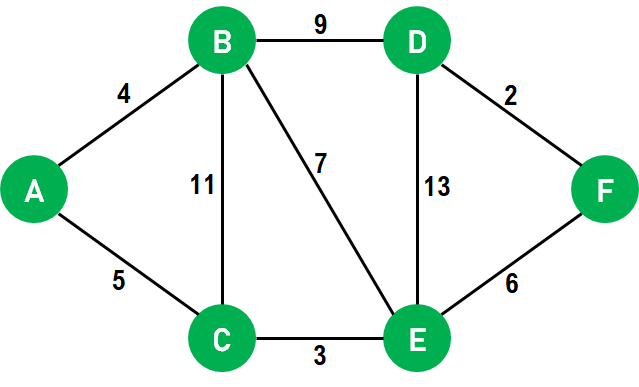
**Step 2:** We will then set the unvisited node with the smallest current distance as the current node, suppose X.

**Step 3:** For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

**Step 4:** We will then mark the current node X as visited.

**Step 5:** We will repeat the process from **'Step 2'** if there is any node unvisited left in the graph.

**Let us now understand the implementation of the algorithm with the help of an example:**

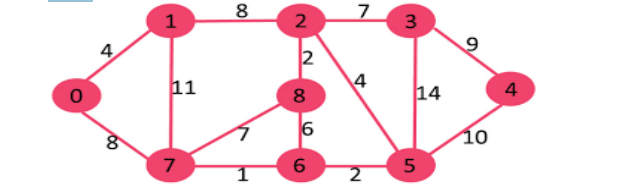
We will use the above graph as the input, with node **A** as the source.First, we will mark all the nodes as unvisited.We will set the path to **0** at node **A** and **INFINITY** for all the other nodes.We will now mark source node **A** as visited and access its neighboring nodes.  
**Note:** We have only accessed the neighboring nodes, not visited them.We will now update the path to node **B** by **4** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **B** is **4**, and the **minimum((0 + 4), INFINITY)** is **4**.We will also update the path to node **C** by **5** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **C** is **5**, and the **minimum((0 + 5), INFINITY)** is **5**. Both the neighbors of node **A** are now relaxed; therefore, we can move ahead.We will now select the next unvisited node with the least path and visit it. Hence, we will visit node **B** and perform relaxation on its unvisited neighbors. After performing relaxation, the path to node **C** will remain **5**, whereas the path to node **E** will become **11**, and the path to node **D** will become **13**.We will now visit node **E** and perform relaxation on its neighboring nodes **B, D**, and **F**. Since only node **F** is unvisited, it will be relaxed. Thus, the path to node **B** will remain as it is, i.e., **4**, the path to node **D** will also remain **13**, and the path to node **F** will become **14 (8 + 6)**.Now we will visit node **D**, and only node **F** will be relaxed. However, the path to node **F** will remain unchanged, i.e., **14**.Since only node **F** is remaining, we will visit it but not perform any relaxation as all its neighbouring nodes are already visited.Once all the nodes of the graphs are visited, the program will end.the final paths we concluded are:

1. A = 0
2. B = 4 (A -> B)
3. C = 5 (A -> C)
4. D = 4 + 9 = 13 (A -> B -> D)
5. E = 5 + 3 = 8 (A -> C -> E)
6. F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

**Pseudocode:**

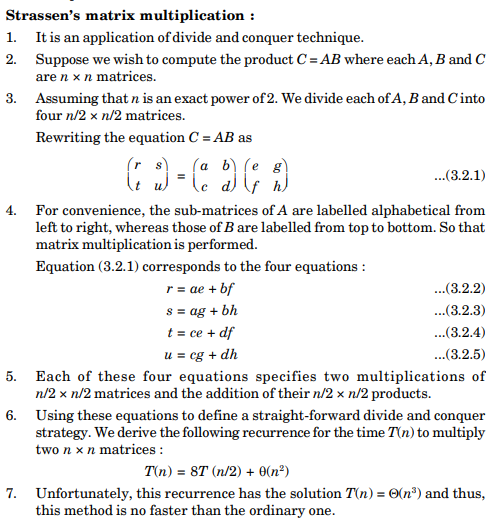
1. function Dijkstra\_Algorithm(Graph, source\_node)
2. // iterating through the nodes in Graph and set their distances to INFINITY
3. for each node N in Graph:
4. distance[N] = INFINITY
5. previous[N] = NULL
6. If N != source\_node, add N to Priority Queue G
7. // setting the distance of the source node of the Graph to 0
8. distance[source\_node] = 0
10. // iterating until the Priority Queue G is not empty
11. while G is NOT empty:
12. // selecting a node Q having the least distance and marking it as visited
13. Q = node in G with the least distance[]
14. mark Q visited
16. // iterating through the unvisited neighboring nodes of the node Q and performing relaxation accordingly
17. for each unvisited neighbor node N of Q:
18. temporary\_distance = distance[Q] + distance\_between(Q, N)
20. // if the temporary distance is less than the given distance of the path to the Node, updating the resultant distance with the minimum value
21. if temporary\_distance < distance[N]
22. distance[N] := temporary\_distance
23. previous[N] := Q
25. // returning the final list of distance
26. return distance[], previous[]

**10.Use single source shortest path algorithm for find the optimal solution for given graph**



**11.Describe in detail the Strassen’s Matrix Multiplication algorithm based on divide & conquer strategies with suitable example.**

Strassen’s Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is **O(n3)**, since it takes two loops to multiply. Strassen’s method was introduced to reduce the time complexity from **O(n3)** to **O(nlog 7)**.



**12.Consider 5 items along their respective weights and values**

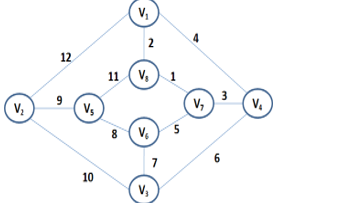
**I=<I1,I2,I3,I4,I5>**

**W=<5,10,20,30,40,>**

**V=<30,20,100,90,160>**

**The capacity of Knapsack w=60. Find the solution to the fractional knapsack problem.**

**13.Define minimum spanning tree (MST). Write Prim’s algorithm to generate a MST for any given weighted graph. Generate MST for the following graph using Prim’s algorithm.**



**14.What is Knapsack problem? Solve Fractional knapsack problem using greedy programming for the following four items with their weights w = {3, 5, 9, 5} and values P = {45, 30, 45, 10} with knapsack capacity is 16.**

**15.Find an optimal parenthesization of a matrix chain product whose sequence of dimensions is {10, 5, 3, 12, 6}.**

**16.Consider the following instance for knapsack problem. Find the solution using Greedy method:**

**N= 10, W=130**

**P [] = {21, 31, 43, 53, 41, 63, 65, 75}**

**V [] = {11, 21, 31, 33, 43, 53, 65, 65}**