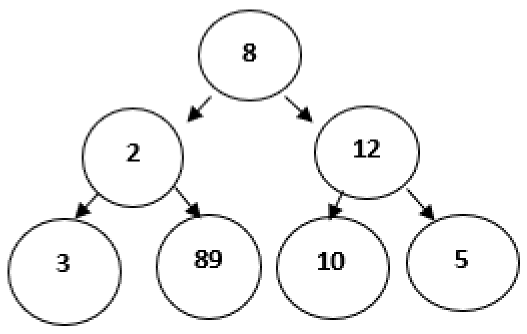
Greedy Approach or Technique

As the name implies, this is a simple approach which tries to find the **best** solution at every step. Thus, it aims to find the local optimal solution at every step so as to find the global optimal solution for the entire problem.

Consider that there is an **objective function** that has to be optimized (maximized/ minimized). This approach makes greedy choices at each step and makes sure that the objective function is optimized.

The greedy algorithm has only one chance to compute the optimal solution and thus, cannot go back and look at other alternate solutions. However, in many problems, this strategy fails to produce a global optimal solution. Let's consider the following binary tree to understand how a basic greedy algorithm works:



For the above problem the objective function is:

To find the path with largest sum.

Since we need to maximize the objective function, Greedy approach can be used. Following steps are followed to find the solution:

Step 1: Initialize **sum = 0**

Step 2: Select the root node, so its value will be added to `sum`, **sum = 0+8 = 8**

Step 3: The algorithm compares nodes at next level, selects the largest node which is **12**, making the **sum = 20**.

Step 4: The algorithm compares nodes at the next level, selects the largest node which is **10**, making the **sum = 30**.

Thus, using the greedy algorithm, we get **8-12-10** as the path. But this is not the optimal solution, since the path **8-2-89** has the largest sum ie **99**.

This happens because the algorithm makes decision based on the information available at each step without considering the overall problem.

When to use Greedy Algorithms?

For a problem with the following properties, we can use the greedy technique:

- **Greedy Choice Property**: This states that a globally optimal solution can be obtained by locally optimal choices.
- **Optimal Sub-Problem**: This property states that an optimal solution to a problem, contains within it, optimal solution to the sub-problems. Thus, a globally optimal solution can be constructed from locally optimal sub-solutions.

Generally, **optimization problem**, or the problem where we have to find maximum or minimum of something or we have to find some optimal solution, greedy technique is used.

An optimization problem has two types of solutions:

- **Feasible Solution**: This can be referred as approximate solution (subset of solution) satisfying the objective function and it may or may not build up to the optimal solution.
- **Optimal Solution**: This can be defined as a feasible solution that either maximizes or minimizes the objective function.

Key Terminologies used in Greedy Algorithms

- **Objective Function**: This can be defined as the function that needs to be either maximized or minimized.
- **Candidate Set**: The global optimal solution is created from this set.
- **Selection Function**: Determines the best candidate and includes it in the solution set.
- **Feasibility Function**: Determines whether a candidate is feasible and can contribute to the solution.

Standard Greedy Algorithm

This algorithm proceeds step-by-step, considering one input, say x, at each step.

- If x gives a local optimal solution (x is feasible), then it is included in the partial solution set, else it is discarded.
- The algorithm then goes to the next step and never considers x again.
- This continues until the input set is finished or the optimal solution is found.

The above algorithm can be translated into the following pseudocode:

```
Algorithm Greedy(a, n)    // n defines the input set

{

    solution= NULL;       // initialize solution set

     for i=1 to n do

     {

          x = Select(a);  // Selection Function

          if Feasible(solution, x) then  // Feasibility solution

               solution = Union (solution, x);    // Include x in the
solution set

     }

     return solution;

}
```

Copy

Advantages of Greedy Approach/Technique

- This technique is easy to formulate and implement.
- It works efficiently in many scenarios.
- This approach minimizes the time required for generating the solution.

Now, let's see a few disadvantages too,

Disadvantages of Greedy Approach/Technique

- This approach does not guarantee a global optimal solution since it never looks back at the choices made for finding the local optimal solution.

Although we have already covered that which type of problem in general can be solved using greedy approach, here are a few popular problems which use greedy technique:

1. Knapsack Problem
2. Activity Selection Problem
3. Dijkstra’s Problem
4. Prim’s Algorithmfor finding Minimum Spanning Tree
5. Kruskal’s Algorithmfor finding Minimum Spanning Tree
6. Huffman Coding
7. Travelling Salesman Problem

Conclusion

Greedy Technique is best suited for applications where:

- Solution is required in real-time.
- Approximate solution is sufficient.

Activity Selection Problem

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have `n` activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

- It might not be possible to complete all the activities, since their timings can collapse.
- Two activities, say **i** and **j**, are said to be non-conflicting if `si >= fj` or `sj >= fi` where `si` and `sj` denote the starting time of activities **i** and **j** respectively, and `fi` and `fj` refer to the finishing time of the activities **i** and **j** respectively.
- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- `act[]` array containing all the activities.
- `s[]` array containing the starting time of all the activities.
- `f[]` array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- `sol[]` array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array `act[]` and add it to `sol[]` array.

Step 3: Repeat steps 4 and 5 for the remaining activities in `act[]`.

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the `sol[]` array.

Step 5: Select the next activity in `act[]` array.

Step 6: Print the `sol[]` array.

Activity Selection Problem Example

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) > f(a2)), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.

C. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

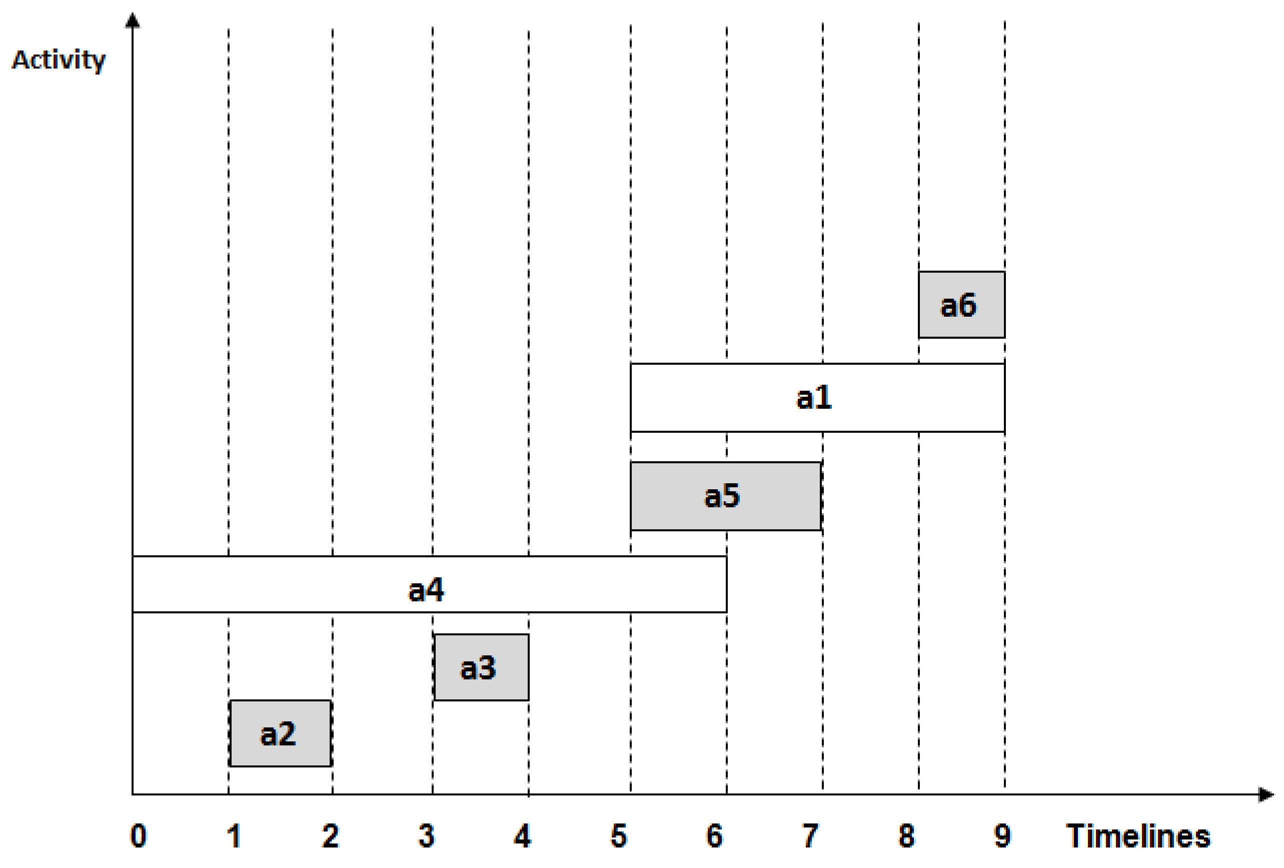
D. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5). Thus **sol = {a2, a3, a5, a6}**.

Step 6: At last, print the array sol[]

Hence, the execution schedule of maximum number of non-conflicting activities will be:

```
(1,2)
(3,4)
(5,7)
(8,9)
```



In the above diagram, the selected activities have been highlighted in grey.

Implementation of Activity Selection Problem Algorithm

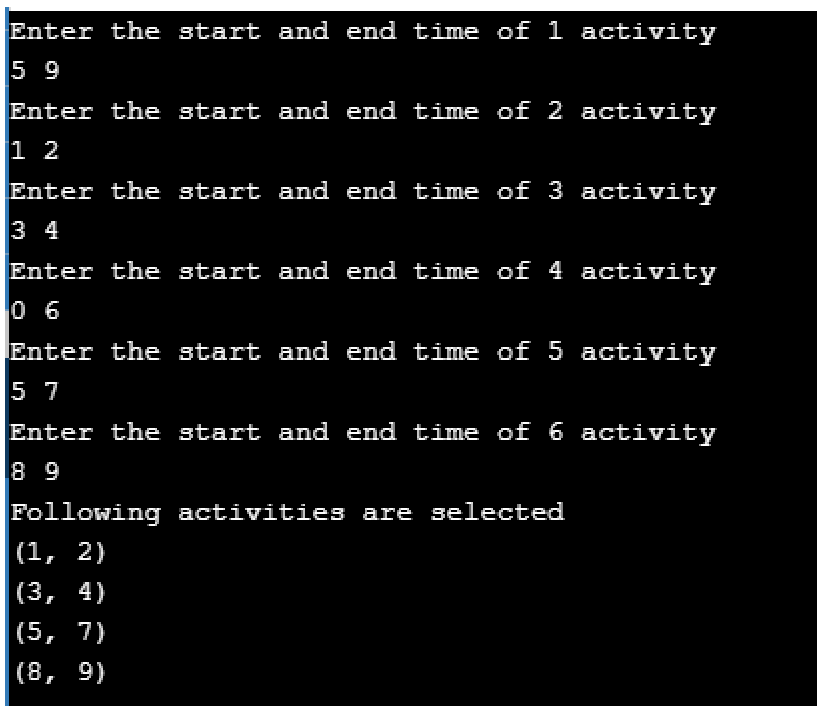
Now that we have an overall understanding of the activity selection problem as we have already discussed the algorithm and its working details with the help of an example, following is the C++ implementation for the same.

Note: The algorithm can be easily written in any programming language.

```
#include <bits/stdc++.h>


using namespace std;

#define N 6          // defines the number of activities


// Structure represents an activity having start time and finish time.

struct Activity

{

    int start, finish;

};


// This function is used for sorting activities according to finish
time

bool Sort_activity(Activity s1, Activity s2)

{

    return (s1.finish< s2.finish);

}


/*   Prints maximum number of activities that can

     be done by a single person or single machine at a time.

*/

void print_Max_Activities(Activity arr[], int n)

{

    // Sort activities according to finish time

     sort(arr, arr+n, Sort_activity);


     cout<< "Following activities are selected \n";


    // Select the first activity

    int i = 0;
```

```
     cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";


    // Consider the remaining activities from 1 to n-1

    for (int j = 1; j < n; j++)

    {

     // Select this activity if it has start time greater than or equal

     // to the finish time of previously selected activity

          if (arr[j].start>= arr[i].finish)

          {

               cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ")
\n";

               i = j;

          }

    }

}


// Driver program

int main()

{

    Activity arr[N];

     for(int i=0; i<=N-1; i++)

     {

          cout<<"Enter the start and end time of "<<i+1<<" activity
\n";

          cin>>arr[i].start>>arr[i].finish;

    }


     print_Max_Activities(arr, N);

    return 0;

}
```
Copy

The program is executed using same inputs as that of the example explained above. This will help in verifying the resultant solution set with actual output.



```
Enter the start and end time of 1 activity
5 9
Enter the start and end time of 2 activity
1 2
Enter the start and end time of 3 activity
3 4
Enter the start and end time of 4 activity
0 6
Enter the start and end time of 5 activity
5 7
Enter the start and end time of 6 activity
8 9
Following activities are selected
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Time Complexity Analysis

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

- **Case 1**: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be `O(n)`
- **Case 2**: When a given set of activities is unsorted, then we will have to use the `sort()` method defined in **bits/stdc++** header file for sorting the activities list. The time complexity of this method will be `O(nlogn)`, which also defines complexity of the algorithm.

Real-life Applications of Activity Selection Problem

Following are some of the real-life applications of this problem:

- Scheduling multiple competing events in a room, such that each event has its own start and end time.
- Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
- Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.

Prim's Minimum Spanning Tree

In this tutorial we will cover another algorithm which uses greedy approach/technique for finding the solution.

Let's start with a real-life scenario to understant the premise of this algorithm:

1. A telecommunications organization, has offices spanned across multiple locations around the globe.

Figure 1

2. It has to use leased phone lines for connecting all these offices with each other.
3. The cost(in units) of connecting each pair of offices is different and is shown as follows:

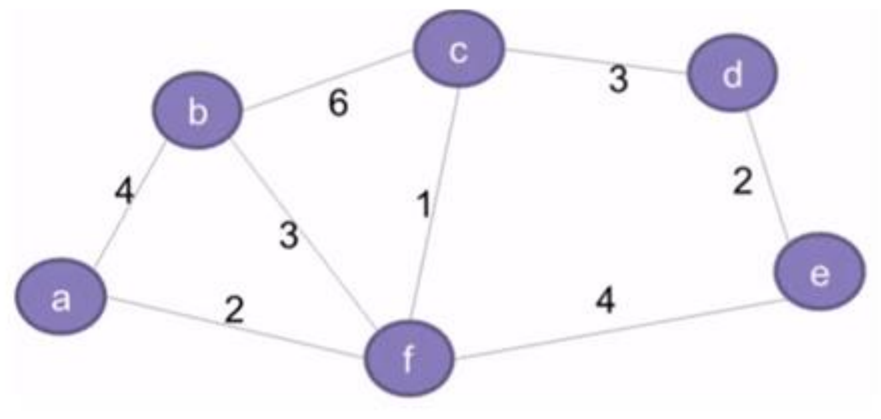


Figure 2

4. The organization, thus, wants to use minimum cost for connecting all its offices. This requires that all the offices should be connected using minimum number of leased lines so as to reduce the effective cost.
5. The solution to this problem can be implemented by using the concept of **Minimum Spanning Tree**, which is discussed in the subsequent section.
6. This tutorial also details the concepts related to Prim's Algorithm which is used for finding the minimum spanning tree for a given graph.

What is a Spanning Tree?

The network shown in the second figure basically represents a graph **G = (V, E)** with a set of vertices **V = {a, b, c, d, e, f}** and a set of edges **E = { (a,b), (b,c), (c,d), (d,e), (e,f), (f,a), (b,f), (c,f) }**. The graph is:

- Connected (there exists a path between every pair of vertices)

- Undirected (the edges do no have any directions associated with them such that (a,b) and (b,a) are equivalent)
- Weighted (each edge has a weight or cost assigned to it)

A spanning tree $G' = (V, E')$ for the given graph G will include:

- All the vertices (V) of G
- All the vertices should be connected by minimum number of edges (E') such that $E' \subset E$
- G' can have maximum $n-1$ edges, where n is equal to the total number of edges in G
- G' should not have any cycles. This is one of the basic differences between a tree and graph that **a graph can have cycles, but a tree cannot**. Thus, a tree is also defined as an **acyclic graph**.

Following is an example of a spanning tree for the above graph. Please note that only the highlighted edges are included in the spanning tree,

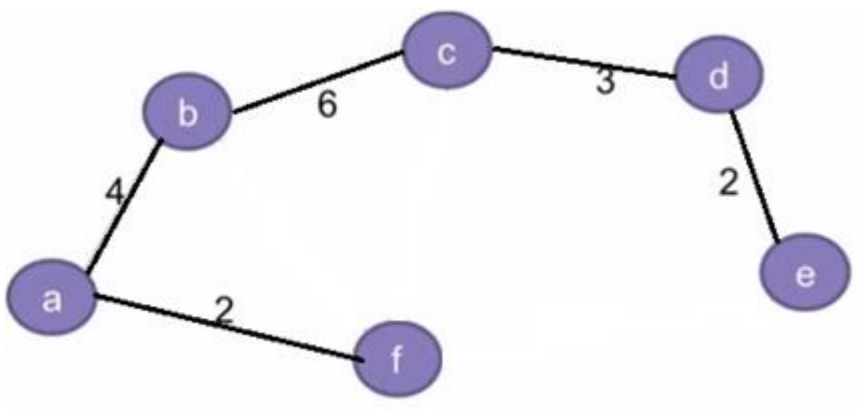


Figure 3

Also, there can be multiple spanning trees possible for any given graph. For eg: In addition to the spanning tree in the above diagram, the graph can also have another spanning tree as shown below:

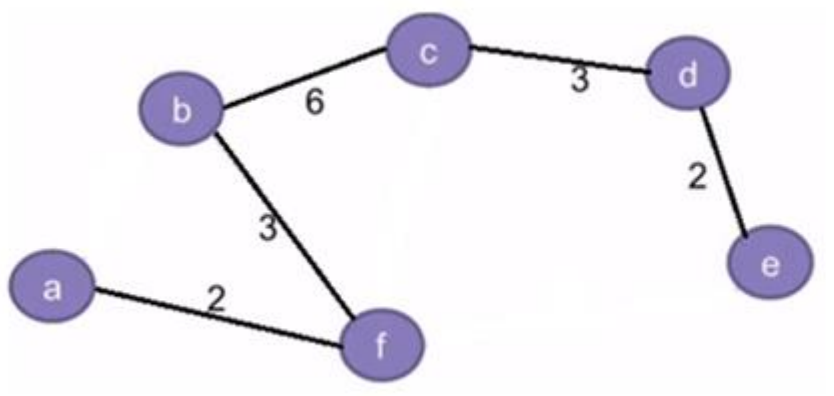


Figure 4

By convention, the total number of spanning trees for a given graph can be defined as:

$^nC_m = n!/(m!*(n-m)!)$, where,

- n is equal to the total number of edges in the given graph
- m is equal to the total number of edges in the spanning tree such that $m <= (n-1)$.

Hence, the total number of spanning trees(S) for the given graph(second diagram from top) can be computed as follows:

- **n = 8**, for the given graph in Fig. 2
- **m = 5**, since its corresponding spanning tree can have only 5 edges. Adding a 6th edge can result in the formation of cycles which is not allowed.
- So, **S = ^nC_m = $^8C_5$ = 8!/ (5! * 3!) = 56**, which means that **56** different variations of spannig trees can be created for the given graph.

What is a Minimum Spanning Tree?

The cost of a spanning tree is the total of the weights of all the edges in the tree. For example, the cost of spanning tree in Fig. 3 is **(2+4+6+3+2) = 17** units, whereas in Fig. 4 it is **(2+3+6+3+2) = 16** units.

Since we can have multiple spanning trees for a graph, each having its own cost value, the objective is to find the spanning tree with minimum cost. This is called a **Minimum Spanning Tree(MST)**.

Note: There can be multiple minimum spanning trees for a graph, if any two edges in the graph have the same weight. However, if each edge has a distinct weight, then there will be only one minimum spanning tree for any given graph.

Problem Statement for Minimum Spanning Tree

Given a weighted, undirected and connected graph **G**, the objective is to find the minimum spanning tree **G'** for G.

Apart from the Prim's Algorithm for minimum spanning tree, we also have Kruskal's Algorithm for finding minimum spanning tree.

However, this tutorial will only discuss the fundamentals of **Prim's Algorithm.**

Since this algorithm aims to find the spanning tree with minimum cost, it uses **greedy approach** for finding the solution.

As part of finding the or creating the minimum spanning tree fram a given graph we will be following these steps:

- Initially, the tree is empty.
- The tree starts building from a random source vertex.
- A new vertex gets added to the tree at every step.
- This continues till all the vertices of graph are added to the tree.

Input Data will be:

A **Cost Adjacency Matrix** for out graph **G**, say `cost`

Output will be:

A Spanning tree with minimum total cost

Algorithm for Prim's Minimum Spanning Tree

Below we have the complete logic, stepwise, which is followed in prim's algorithm:

Step 1: Keep a track of all the vertices that have been visited and added to the spanning tree.

Step 2: Initially the spanning tree is empty.

Step 3: Choose a random **vertex**, and add it to the spanning tree. This becomes the **root node**.

Step 4: Add a new vertex, say **x**, such that

a. **x** is not in the already built spanning tree.

b. **x** is connected to the built spanning tree using minimum weight edge. (Thus, **x** can be adjacent to any of the nodes that have already been added in the spanning tree).

c. Adding **x** to the spanning tree should not form cycles.

Step 5: Repeat the Step 4, till all the vertices of the graph are added to the spanning tree.

Step 6: Print the total cost of the spanning tree.

Example for Prim's Minimum Spanning Tree Algorithm

Let's try to trace the above algorithm for finding the Minimum Spanning Tree for the graph in Fig. 2:

Step A:

a. Define key[] array for storing the key value(or cost) of every vertex. Initialize this to ∞(infinity) for all the vertices

b. Define another array booleanvisited[] for keeping a track of all the vertices that have been added to the spanning tree. Initially this will be **0** for all the vertices, since the spanning tree is empty.

c. Define an array parent[] for keeping track of the parent vertex. Initialize this to **-1** for all the vertices.

d. Initialize minimum cost, **minCost = 0**

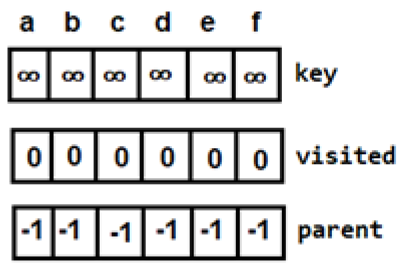


Figure 5: Initial arrays when tree is empty

Step B:

Choose any random vertex, say **f** and set key[f]=0.

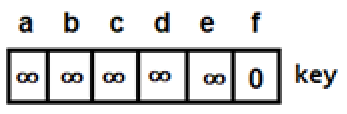


Figure 6: Setting key value of root node

Since its key value is minimum and it is not visited, add **f** to the spanning tree.

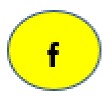
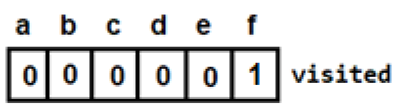


Figure 7: Root Node

Also, update the following:

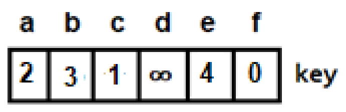
- minCost = 0 + key[f] = 0
- This is how the visited[] array will look like:



a	b	c	d	e	f	
0	0	0	0	0	1	visited

Figure 8: visited array after adding the root node

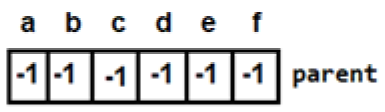
- Key values for all the adjacent vertices of **f** will look like this(key value is nothing but the cost or the weight of the edge, for **(f,d)** it is still infinity because they are not directly connected):



a	b	c	d	e	f	
2	3	1	∞	4	0	key

Figure 9: key array after adding the root node

Note: There will be no change in the parent[] because **f** is the root node.



a	b	c	d	e	f	
-1	-1	-1	-1	-1	-1	parent

Figure 10: parent array after adding the root node

Step C:

The arrays key[] and visited[] will be searched for finding the next vertex.

- **f** has the minimum key value but will not be considered since it is already added (visited[f]==1)
- Next vertex having the minimum key value is **c**. Since visited[c]==0, it will be added to the spanning tree.

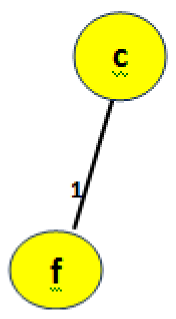
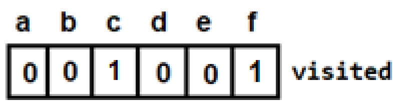


Figure 11: Adding vertex c

Again, update the following:

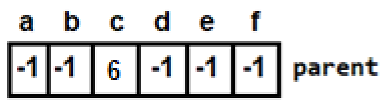
- minCost = 0 + key[c] = 0 + 1 = 1
- This is how the visited[] array will look like:



a	b	c	d	e	f	
0	0	1	0	0	1	visited

Figure 12: visited array after adding vertex c

- And, the parent[] array (**f** becomes the parent of **c**):



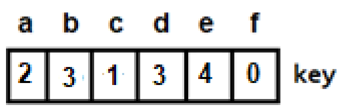
a	b	c	d	e	f	
-1	-1	6	-1	-1	-1	parent

Figure 13: parent array after adding the root node

- For every adjacent vertex of **c**, say **v**, values in key[v] will be updated using the formula:

  key[v] = min(key[v], cost[c][v])

  Thus the key[] array will become:



a	b	c	d	e	f	
2	3	1	3	4	0	key

Figure 14: key array after adding the root node

Step D:

Repeat the Step C for the remaining vertices.

- Next vertex to be selected is **a**. And minimum cost will become minCost=1+2=3

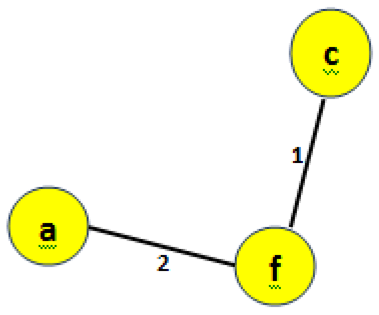
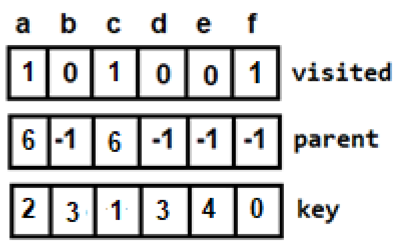


Figure 15: Adding vertex a



a	b	c	d	e	f	
1	0	1	0	0	1	visited
6	-1	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 16: Updated arrays after adding vertex a

- Next, either **b** or **d** can be selected. Let's consider **b**. Then the minimum cost will become $minCost=3+3=6$

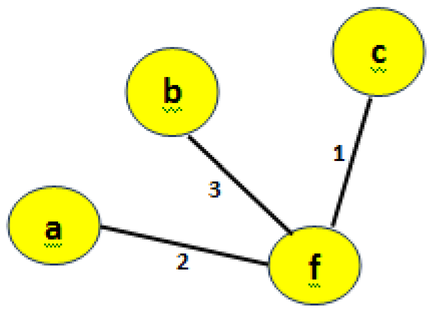
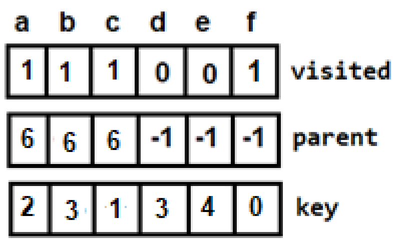


Figure 17: Adding vertex b to minimum spanning tree



a	b	c	d	e	f	
1	1	1	0	0	1	visited
6	6	6	-1	-1	-1	parent
2	3	1	3	4	0	key

Figure 18: Updated arrays after adding vertex b

- Next vertex to be selected is **d**, making the minimum cost $minCost=6+3=9$

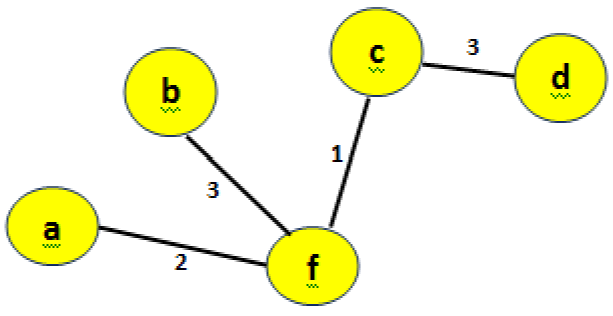
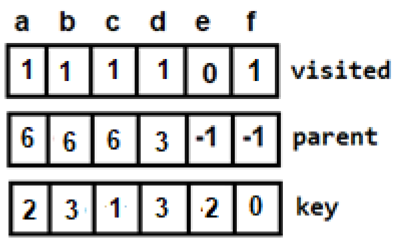


Figure 19: Adding vertex d



a	b	c	d	e	f	
1	1	1	1	0	1	visited
6	6	6	3	-1	-1	parent
2	3	1	3	2	0	key

Figure 20: Updated arrays after adding vertex d

- Then, **e** is selected and the minimum cost will become, $minCost=9+2=11$

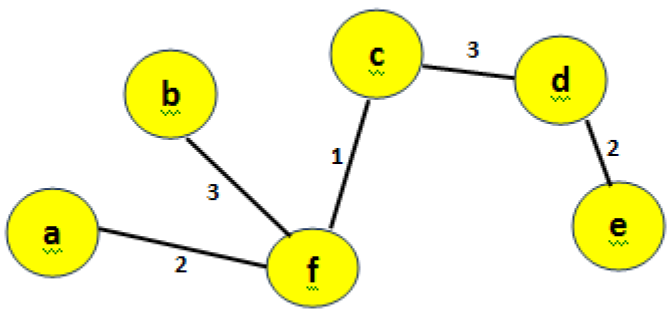
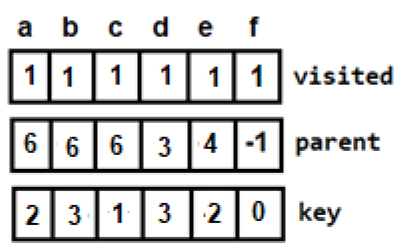


Figure 21: Adding vertex e. This is the final minimum spanning tree



	a	b	c	d	e	f
visited	1	1	1	1	1	1
parent	6	6	6	3	4	-1
key	2	3	1	3	2	0

Figure 22: Updated arrays after adding vertex e (final arrays)

- Since all the vertices have been visited now, the algorithm terminates.
- Thus, Fig. 21 represents the Minimum Spanning Tree with total **cost=11**.

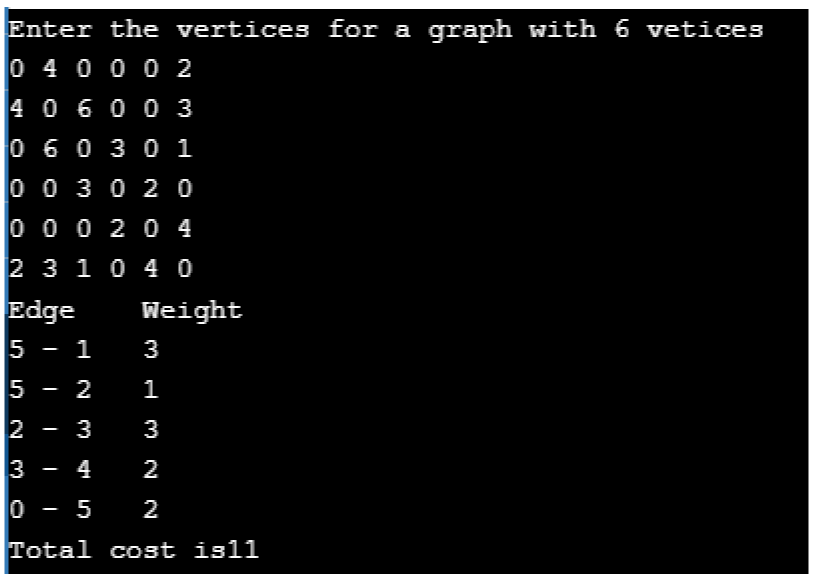
Implementation of Prim's Minimum Spanning Tree Algorithm

Now it's time to write a program in C++ for the finding out minimum spanning tree using prim's algorithm.

```
#include<iostream>


using namespace std;


// Number of vertices in the graph

const int V=6;


// Function to find the vertex with minimum key value

int min_Key(int key[], bool visited[])

{

    int min = 999, min_index;  // 999 represents an Infinite value


    for (int v = 0; v < V; v++) {

        if (visited[v] == false && key[v] < min) {

          // vertex should not be visited

            min = key[v];

               min_index = v;

        }

    }

    return min_index;

}
```

```
// Function to print the final MST stored in parent[]

int print_MST(int parent[], int cost[V][V])

{

    int minCost=0;

     cout<<"Edge \tWeight\n";

    for (int i = 1; i< V; i++) {

          cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";

          minCost+=cost[i][parent[i]];

    }

     cout<<"Total cost is"<<minCost;

}


// Function to find the MST using adjacency cost matrix representation

void find_MST(int cost[V][V])

{

    int parent[V], key[V];

    bool visited[V];


    // Initialize all the arrays

    for (int i = 0; i< V; i++) {

        key[i] = 999;    // 99 represents an Infinite value

        visited[i] = false;

        parent[i]=-1;

    }


    key[0] = 0;  // Include first vertex in MST by setting its key vaue
to 0.

    parent[0] = -1; // First node is always root of MST


    // The MST will have maximum V-1 vertices
    for (int x = 0; x < V - 1; x++)

    {

        // Finding the minimum key vertex from the

        //set of vertices not yet included in MST

        int u = min_Key(key, visited);


        visited[u] = true;  // Add the minimum key vertex to the MST


        // Update key and parent arrays

        for (int v = 0; v < V; v++)

        {

            // cost[u][v] is non zero only for adjacent vertices of u

            // visited[v] is false for vertices not yet included in MST

            // key[] gets updated only if cost[u][v] is smaller than
key[v]

            if (cost[u][v]!=0 && visited[v] == false && cost[u][v] <
key[v])

            {

                parent[v] = u;

                key[v] = cost[u][v];

            }

        }

    }


    // print the final MST

     print_MST(parent, cost);

}


// main function

int main()

{
    int cost[V][V];

     cout<<"Enter the vertices for a graph with 6 vetices";

    for (int i=0;i<V;i++)

    {

        for(int j=0;j<V;j++)

        {

                cin>>cost[i][j];

        }

    }

     find_MST(cost);


    return 0;

}
```
Copy

The input graph is the same as the graph in Fig 2.



```
Enter the vertices for a graph with 6 vetices
0 4 0 0 0 2
4 0 6 0 0 3
0 6 0 3 0 1
0 0 3 0 2 0
0 0 0 2 0 4
2 3 1 0 4 0
Edge    Weight
5 - 1    3
5 - 2    1
2 - 3    3
3 - 4    2
0 - 5    2
Total cost is11
```

Figure 23: Output of the program

Time Complexity Analysis for Prim's MST

Time complexity of the above C++ program is **O(V2)** since it uses adjacency matrix representation for the input graph. However, using an adjacency list representation, with the help of binary heap, can reduce the complexity of Prim's algorithm to **O(ElogV)**.

Real-world Applications of a Minimum Spanning Tree

Finding an MST is a fundamental problem and has the following real-life applications:

1. Designing the networks including computer networks, telecommunication networks, transportation networks, electricity grid and water supply networks.

2. Used in algorithms for approximately finding solutions to problems like Travelling Salesman problem, minimum cut problem, etc.
    - The objective of a **Travelling Salesman problem** is to find the shortest route in a graph that visits each vertex only once and returns back to the source vertex.
    - A **minimum cut problem** is used to find the minimum number of cuts between all the pairs of vertices in a planar graph. A graph can be classified as planar if it can be drawn in a plane with no edges crossing each other. For example,

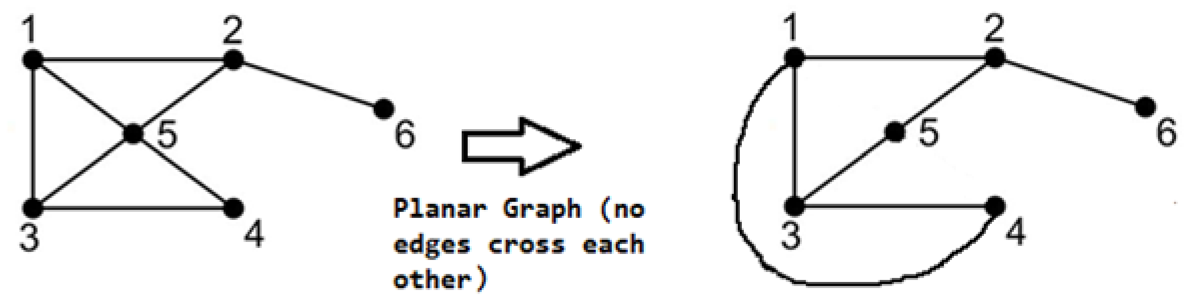


Figure 24: Planar Graph

3. Also, a cut is a subset of edges which, if removed from a planar graph, increases the number of components in the graph

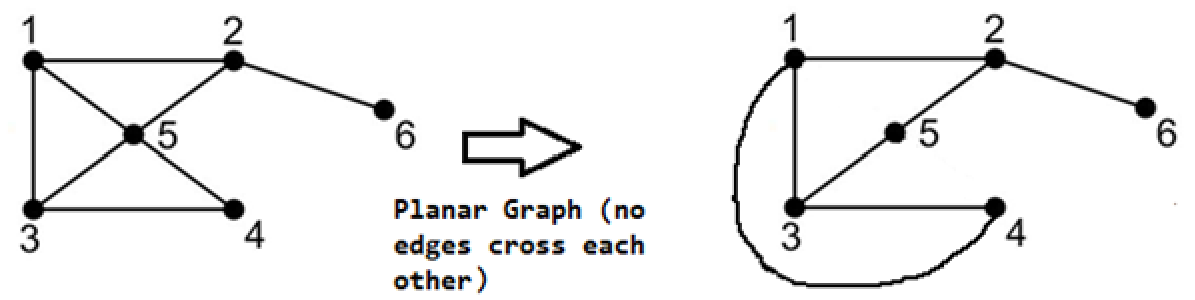


Figure 25: Cut-Set in a Planar Graph

4. Analysis of clusters.
5. Handwriting recognition of mathematical expressions.
6. Image registration and segmentation

Huffman Coding Algorithm

Every information in computer science is **encoded** as strings of **1s and 0s**. The objective of information theory is to usually transmit information using fewest number of bits in such a way that every encoding is unambiguous. This tutorial discusses about fixed-length and variable-length encoding along with Huffman Encoding which is the basis for all data encoding schemes

Encoding, in computers, can be defined as the process of transmitting or storing sequence of characters efficiently. Fixed-length and variable lengthare two types of encoding schemes, explained as follows-

Fixed-Length encoding - Every character is assigned a binary code using same number of bits. Thus, a string like “aabacdad” can require 64 bits (8 bytes) for storage or transmission, assuming that each character uses 8 bits.

Variable- Length encoding - As opposed to Fixed-length encoding, this scheme uses variable number of bits for encoding the characters depending on their frequency in the given text. Thus, for a given string like “aabacdad”, frequency of characters ‘a’, ‘b’, ‘c’ and ‘d’ is 4,1,1 and 2 respectively. Since ‘a’ occurs more frequently than ‘b’, ‘c’ and ‘d’, it uses least number of bits, followed by ‘d’, ‘b’ and ‘c’. Suppose we randomly assign binary codes to each character as follows-

a 0 b 011 c 111 d 11

Thus, the string “aabacdad” gets encoded to **00011011111011 (0 | 0 | 011 | 0 | 111 | 11 | 0 | 11),** using fewer number of bits compared to fixed-length encoding scheme.

But the real problem lies with the decoding phase. If we try and decode the string 00011011111011, it will be quite ambiguous since, it can be decoded to the multiple strings, few of which are-

aaadacdad (0 | 0 | 0 | 11 | 0 | 111 | 11 | 0 | 11) aaadbcad (0 | 0 | 0 | 11 | 011 | 111 | 0 | 11) aabbcb (0 | 0 | 011 | 011 | 111 | 011)

… and so on

To prevent such ambiguities during decoding, the encoding phase should satisfy the **“prefix rule”** which states that no binary code should be a prefix of another code. This will produce uniquely **decodable codes**. The above codes for ‘a’, ‘b’, ‘c’ and ‘d’ do not follow prefix rule since the binary code for a, i.e. 0, is a prefix of binary code for b i.e 011, resulting in ambiguous **decodable codes.**

Lets reconsider assigning the binary codes to characters ‘a’, ‘b’, ‘c’ and ‘d’.

a 0 b 11 c 101 d 100

Using the above codes, string **“aabacdad”** gets encoded to 001101011000100 (0 | 0 | 11 | 0 | 101 | 100 | 0 | 100). Now, we can decode it back to string **“aabacdad”**.

Problem Statement-

Input: Set of symbols to be transmitted or stored along with their frequencies/ probabilities/ weights

Output: Prefix-free and variable-length binary codes with minimum expected codeword length. Equivalently, a tree-like data structure with minimum weighted path length from root can be used for generating the binary codes

Huffman Encoding-
Huffman Encoding can be used for finding solution to the given problem statement.

- Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

The major steps involved in Huffman coding are-

Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

Algorithm for creating the Huffman Tree-

Step 1- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

Step 2- Repeat Steps 3 to 5 while heap has more than one node

Step 3- Extract two nodes, say x and y, with minimum frequency from the heap

Step 4- Create a new internal node z with x as its left child and y as its right child.
Also frequency(z)= frequency(x)+frequency(y)

Step 5- Add z to min heap

Step 6- Last node in the heap is the root of Huffman tree

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15

i	12
o	3
u	4
s	13
t	1

Step A- Create leaf nodes for all the characters and add them to the min heap.

- **Step 1**- Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

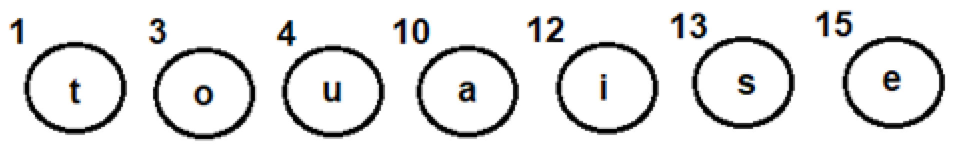


Fig 1: Leaf nodes for each character

Step B- Repeat the following steps till heap has more than one nodes

- **Step 3**- Extract two nodes, say x and y, with minimum frequency from the heap
- **Step 4**- Create a new internal node z with x as its left child and y as its right child. Also frequency(z)= frequency(x)+frequency(y)
- **Step 5**- Add z to min heap

i. Extract and Combinenode u with an internal node having 4 as the frequency
ii. Add the new internal node to priority queue-

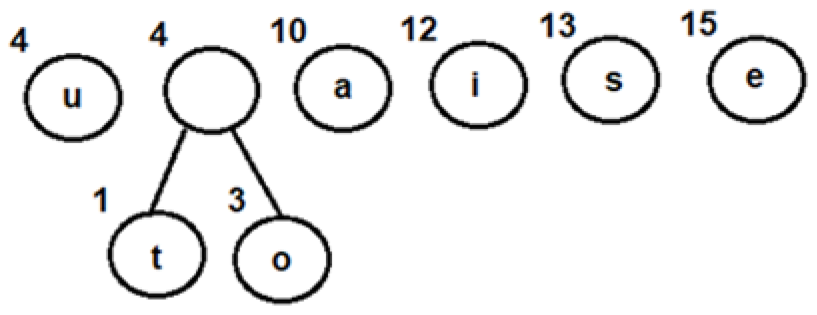


Fig 2: Combining nodes o and t

i. Extract and Combine node awith an internal node having 8 as the frequency
ii. Add the new internal node to priority queue-

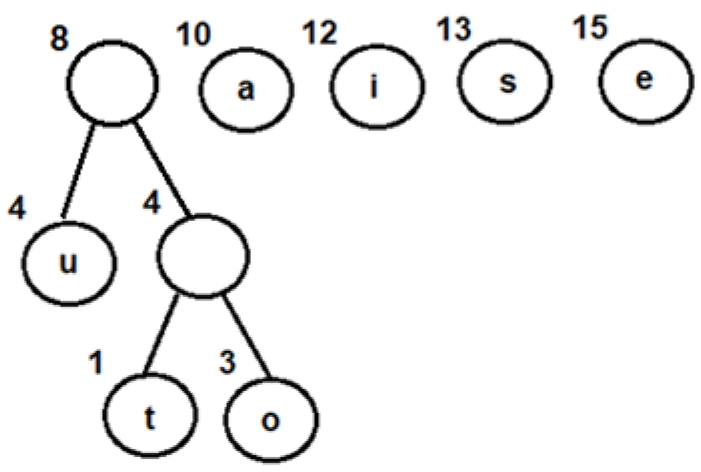


Fig 3: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s
ii. Add the new internal node to priority queue-

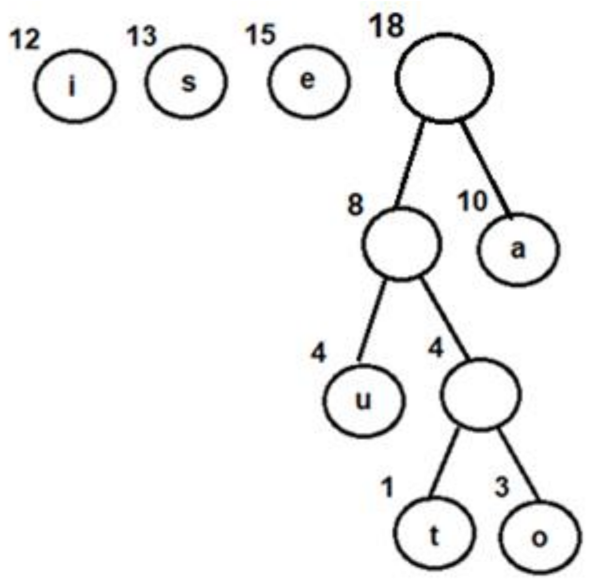


Fig 4: Combining node u withan internal node having 4 as frequency

i. Extract and Combine nodes i and s
ii. Add the new internal node to priority queue-

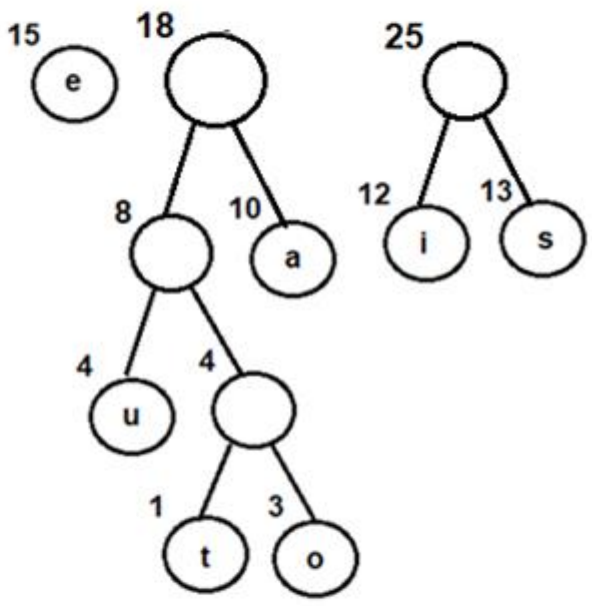


Fig 5: Combining nodes i and s

i. Extract and Combine node ewith an internal node having 18 as the frequency
ii. Add the new internal node to priority queue-

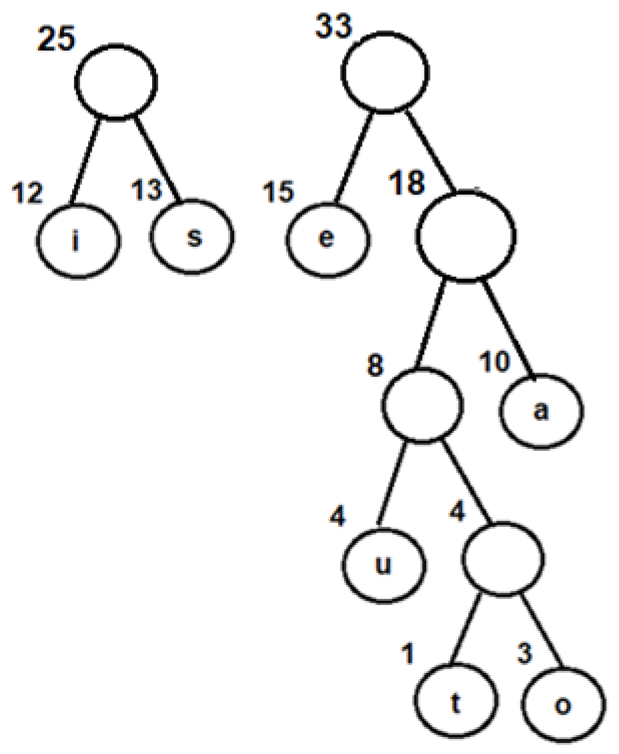


Fig 6: Combining node e with an internal node having 18 as frequency

i. Finally, Extract and Combine internal nodes having 25 and 33 as the frequency
ii. Add the new internal node to priority queue-

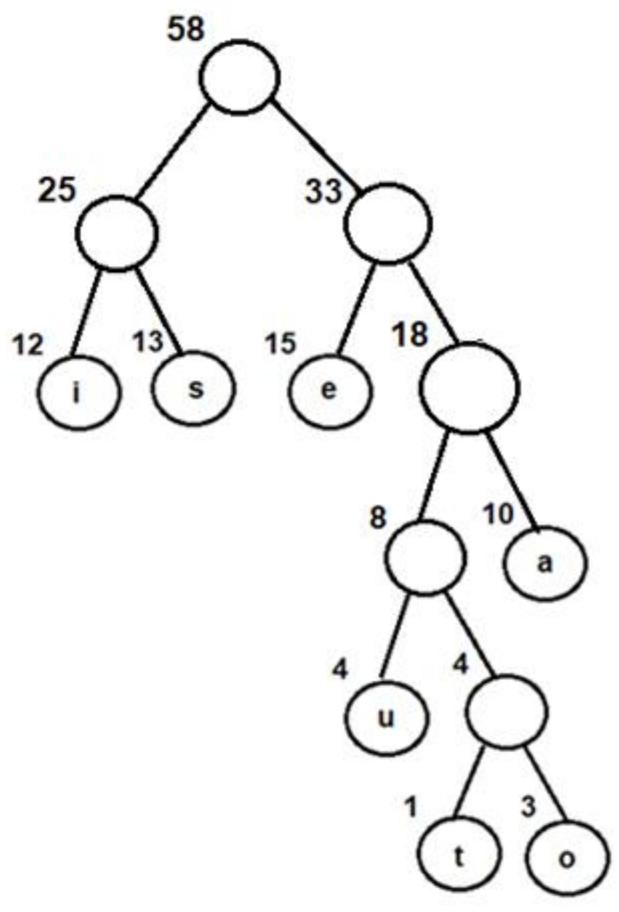


Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

Now, since we have only one node in the queue, the control will exit out of the loop

Step C- Since internal node with frequency 58 is the only node in the queue, it becomes the root of **Huffman tree**.

Step 6- Last node in the heap is the root of Huffman tree

Steps for traversing the Huffman Tree

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Add 0 to arraywhile traversing the left child and add 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

Following the above steps for Huffman Tree generated above, we get prefix-free and variable-length binary codes with minimum expected codeword length-

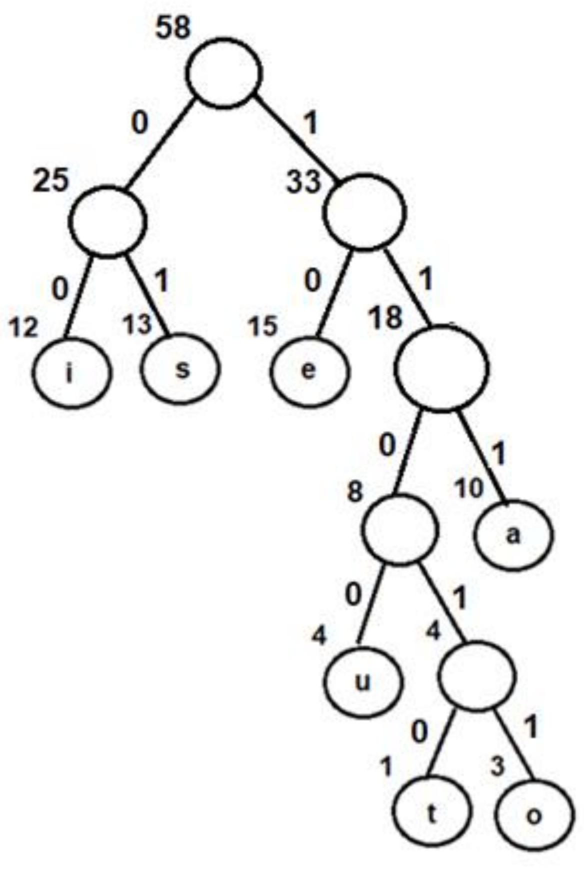


Fig 8: Assigning binary codes to Huffman tree

Characters	Binary Codes
i	00
s	01
e	10
u	1100
t	11010
o	11011
a	111

Using the above binary codes-
Suppose the string "staeiout" needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding, the string gets encoded to **"0111010111100011011110011010" (01 | 11010 | 111 | 10 | 00 | 11011 | 1100 | 11010)** at the sender side.

Once received at the receiver's side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A ‘1’ or ‘0’ in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node.

Thus for the above bit stream

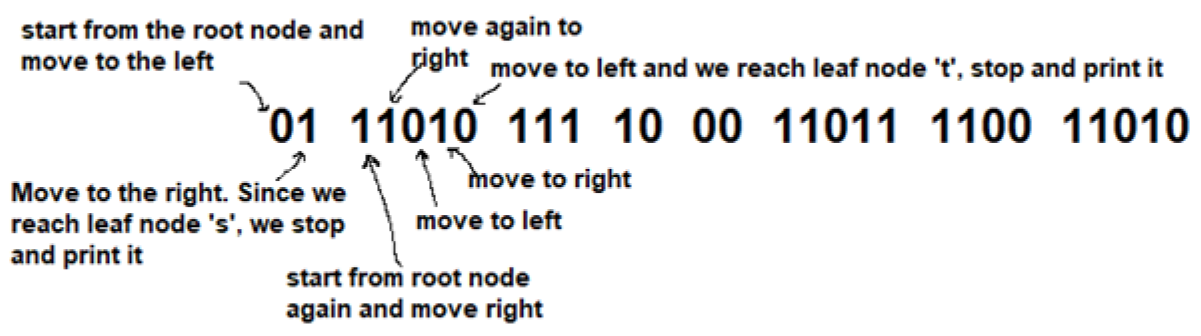

start from the root node and move to the left
move again to right
move to left and we reach leaf node 't', stop and print it
01 11010 111 10 00 11011 1100 11010
Move to the right. Since we reach leaf node 's', we stop and print it
move to right
move to left
start from root node again and move right

Fig 9: Decoding the bit stream

On similar lines-

- 111 gets decoded to ‘a’
- 10 gets decoded to ‘e’
- 00 gets decoded to ‘i’
- 11011 gets decoded to ‘o’
- 1100 gets decoded to ‘u’
- And finally, 11010 gets decoded to ‘t’, thus returning the string “staeiout” back

Implementation-
Following is the C++ implementation of Huffman coding. The algorithmcan be mapped to any programming language as per the requirement.

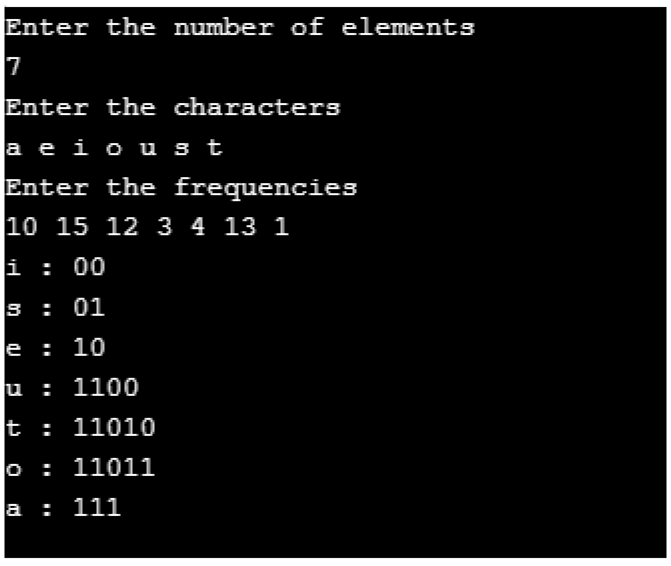
```
#include <iostream>

#include <vector>

#include <queue>

#include <string>


using namespace std;


class Huffman_Codes

{

 struct New_Node

 {

  char data;

  size_t freq;

  New_Node* left;
  New_Node* right;

  New_Node(char data, size_t freq) : data(data),

                                 freq(freq),

left(NULL),

right(NULL)

                                 {}

 ~New_Node()

 {

   delete left;

   delete right;

 }

 };


 struct compare

 {

  bool operator()(New_Node* l, New_Node* r)

  {

    return (l->freq > r->freq);

  }

};


New_Node* top;


void print_Code(New_Node* root, string str)

{

if(root == NULL)

   return;


 if(root->data == '$')

 {
  print_Code(root->left, str + "0");

  print_Code(root->right, str + "1");

 }


 if(root->data != '$')

 {

   cout << root->data <<" : " << str << "\n";

   print_Code(root->left, str + "0");

   print_Code(root->right, str + "1");

 }

}


public:

  Huffman_Codes() {};

  ~Huffman_Codes()

  {

    delete top;

  }

  void Generate_Huffman_tree(vector<char>& data, vector<size_t>& freq,
size_t size)

  {

   New_Node* left;

   New_Node* right;

   priority_queue<New_Node*, vector<New_Node*>, compare > minHeap;


for(size_t i = 0; i < size; ++i)

   {

      minHeap.push(new New_Node(data[i], freq[i]));

   }


while(minHeap.size() != 1)
```

```
    {
      left = minHeap.top();
minHeap.pop();


      right = minHeap.top();
minHeap.pop();


      top = new New_Node('$', left->freq + right->freq);
      top->left  = left;
      top->right = right;
      minHeap.push(top);
     }
    print_Code(minHeap.top(), "");
 }
};


 int main()
 {
  int n, f;
  char ch;
  Huffman_Codes set1;
  vector<char> data;
  vector<size_t> freq;
  cout<<"Enter the number of elements \n";
  cin>>n;
  cout<<"Enter the characters \n";
  for (int i=0;i<n;i++)
  {
      cin>>ch;
data.insert(data.end(), ch);
```

```
  }

  cout<<"Enter the frequencies \n";

  for (int i=0;i<n;i++)

  {

      cin>>f;

freq.insert(freq.end(), f);

  }


  size_t size = data.size();

  set1.Generate_Huffman_tree(data, freq, size);


  return 0;

}
```

Copy

The program is executed using same inputs as that of the example explained above. This will help in verifying the resultant solution set with actual output.



```
Enter the number of elements
7
Enter the characters
a e i o u s t
Enter the frequencies
10 15 12 3 4 13 1
i : 00
s : 01
e : 10
u : 1100
t : 11010
o : 11011
a : 111
```

Fig 10: Output

Time Complexity Analysis-

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

- Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).
- Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

Advantages of Huffman Encoding-

- This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length
- It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string
- The binary codes generated are prefix-free

Disadvantages of Huffman Encoding-

- Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.
- Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
- Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

Real-life applications of Huffman Encoding-

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.

Dijkstra's Algorithm

Dijkstra's algorithm, published in 1959, is named after its discoverer Edsger Dijkstra, who was a Dutch computer scientist. This algorithm aims to find the shortest-path in a directed or undirected graph with non-negative edge weights.

Before, we look into the details of this algorithm, let's have a quick overview about the following:

- **Graph**: A graph is a non-linear data structure defined as G=(V,E) where V is a finite set of vertices and E is a finite set of edges, such that each edge is a line or arc connecting any two vertices.
- **Weighted graph**: It is a special type of graph in which every edge is assigned a numerical value, called weight
- **Connected graph**: A path exists between each pair of vertices in this type of graph
- **Spanning tree** for a graph G is a subgraph G′ including all the vertices of G connected with minimum number of edges. Thus, for a graph G with n vertices, spanning tree G′ will have n vertices and maximum n-1 edges.

Problem Statement

Given a weighted graph G, the objective is to find the shortest path from a given source vertex to all other vertices of G. The graph has the following characteristics-

- Set of vertices `V`
- Set of weighted edges `E` such that `(q,r)` denotes an **edge** between **vertices** q and r and cost(q,r) denotes its weight

Dijkstra's Algorithm:

- This is a single-source shortest path algorithm and aims to find solution to the given problem statement
- This algorithm works for both directed and undirected graphs
- It works only for connected graphs
- The graph should not contain negative edge weights
- The algorithm predominantly follows Greedy approach for finding locally optimal solution. But, it also uses Dynamic Programming approach for building globally optimal solution, since the previous solutions are stored and further added to get final distances from the source vertex
- The main logic of this algorithm is basedon the following formula- `dist[r]=min(dist[r], dist[q]+cost[q][r])`

This formula states that distance vertex r, which is adjacent to vertex q, will be updated if and only if the value of `dist[q]+cost[q][r] is less than dist[r]`. Here-

- dist is a 1-D array which, at every step, keeps track of the shortest distance from source vertex to all other vertices, and
- cost is a 2-D array, representing the cost adjacency matrix for the graph

- This formula uses both Greedy and Dynamic approaches. The Greedy approach is used for finding the minimum distance value, whereas the Dynamic approach is used for combining the previous solutions (**dist[q]** is already calculated and is used to calculate **dist[r]**)

Algorithm-

Input Data-

- Cost Adjacency Matrix for Graph G, say cost
- Source vertex, say s

Output Data-

- Spanning tree having shortest path from s to all other vertices in G

Following are the steps used for finding the solution-

Step 1; Set dist[s]=0, S=ϕ // s is the source vertex and S is a 1-D array having all the visited vertices

Step 2: For all nodes v except s, set dist[v]= ∞

Step 3: find q not in S such that dist[q] is minimum // vertex q should not be visited

Step 4: add q to S // add vertex q to S since it has now been visited

Step 5: update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited dist[r]=min(dist[r], dist[q]+cost[q][r]) //Greedy and Dynamic approach

Step 6: Repeat Steps 3 to 5 until all the nodes are in S // repeat till all the vertices have been visited

Step 7: Print array dist having shortest path from the source vertex u to all other vertices

Step 8: Exit

Let's try and understand the working of this algorithm using the following example-

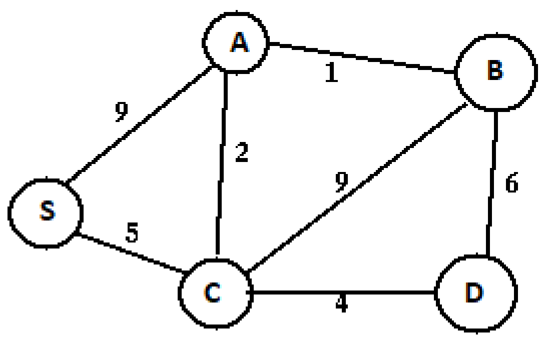


Fig 1: Input Graph (Weighted and Connected)

Given the above weighted and connected graph and source vertex s, following steps are used for finding the tree representing shortest path between s and all other vertices-

Step A- Initialize the distance array (dist) using the following steps of algorithm –

- **Step 1**- Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the visited vertices
- **Step 2**- For all nodes v except s, set dist[v]= ∞

Set of visited vertices (S)	S	A	B	C	D

	0	∞	∞	∞	∞

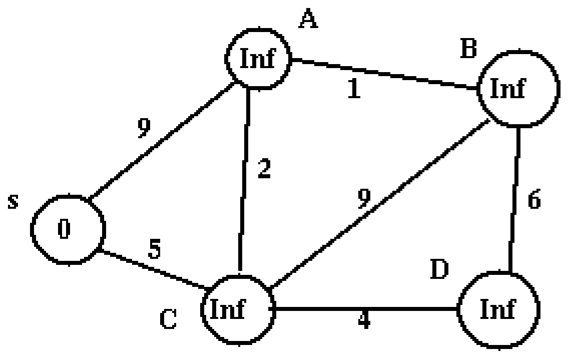


Fig 2: Graph after initializing dist[]

Step B- a)Choose the source vertex s as dist[s] is minimum and s is not in S.

Step 3- find q not in S such that dist[q] is minimum // vertex should not be visited

Visit s by adding it to S

Step 4- add q to S // add vertex q to S since it has now been visited

Step c) For all adjacent vertices of s which have not been visited yet (are not in S) i.e A and C, update the distance array using the following steps of algorithm -

Step 5- update dist[r] for all r adjacent to q such that r is not in S //vertex r should not be visited dist[r]=min(dist[r], dist[q]+cost[q][r]) //Greedy and Dynamic approach

dist[A]= min(dist[A], dist[s]+cost(s, A)) = min(∞, 0+9) = 9 dist[C] = min(dist[C], dist[s]+cost(s, C)) = min(∞, 0+5) = 5

Thus dist[] gets updated as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. Updating the distance array for adjacent vertices of C i.e. A, B and D

Step 6- Repeat Steps 3 to 5 until all the nodes are in S

dist[A]=min(dist[A], dist[C]+cost(C,A)) = min(9, 5+2)= 7

dist[B]= min(dist[B], dist[C]+cost(C,B)) = min(∞, 5+9)= 14

dist[D]= min(dist[D], dist[C]+cost(C,D))= min((∞,5+4)=9

This updates dist[] as follows-

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

[s,C]	0	7	14	5	9

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

The last updation of dist[] gives the shortest path values from s to all other vertices

The resultant shortest path spanning tree for the given graph is as follows-

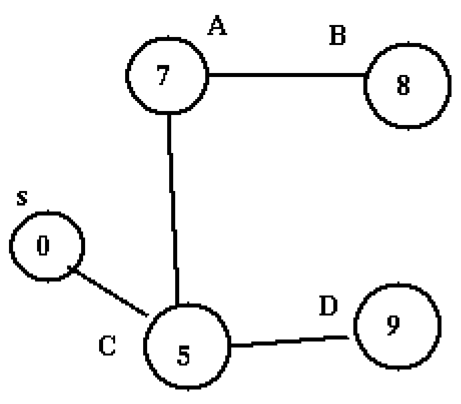


Fig 3: Shortest path spanning tree

Note-

- There can be multiple shortest path spanning trees for the same graph depending on the source vertex

Implementation-
Following is the C++ implementation for Dijkstra's Algorithm

Note : The algorithm can be mapped to any programming language as per the requirement.

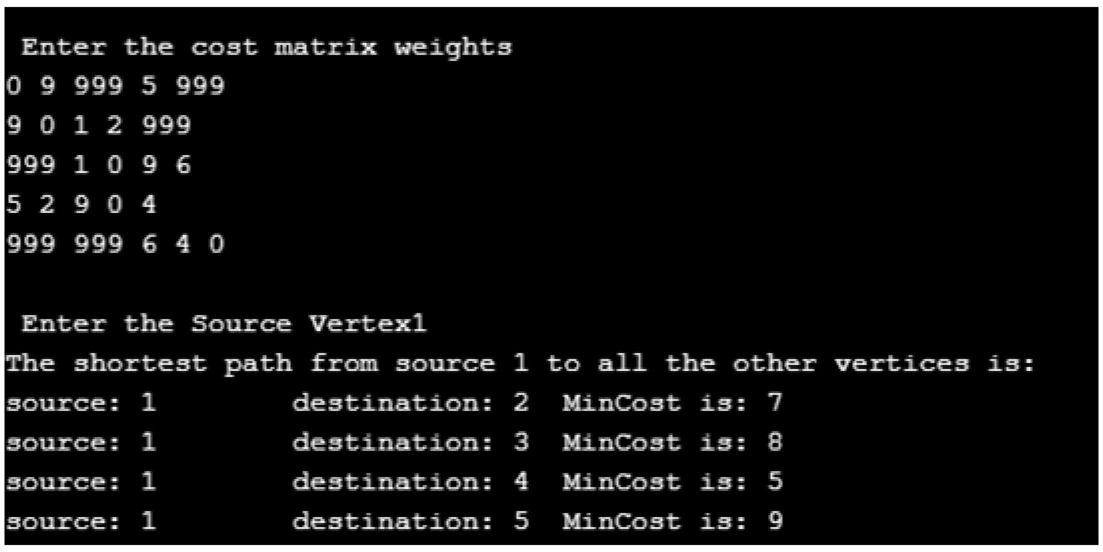
```
#include<iostream>

using namespace std;

#define V 5  //Defines total number of vertices in the graph

#define INFINITY 999
```

```
int min_Dist(int dist[], bool visited[])

//This method used to find the vertex with minimum distance and is not
yet visited

{

     int min=INFINITY,index;                 //Initialize min with
infinity

     for(int v=1;v<=V;v++)

     {

          if(visited[v]==false &&dist[v]<=min)

          {

               min=dist[v];

               index=v;

          }

     }

     return index;

}

void Dijkstra(int cost[V][V],int src) //Method to implement shortest
path algorithm

{

     int dist[V];

     bool visited[V];

     for(int i=1;i<=V;i++)                    //Initialize dist[] and
visited[]

     {

          dist[i]=INFINITY;

          visited[i]=false;

     }

     //Initialize distance of the source vertec to zero

     dist[src]=0;

     for(int c=2;c<=V;c++)

     {

          //u is the vertex that is not yet included in visited and is
having minimum
```

```
          int u=min_Dist(dist,visited);            distance

          visited[u]=true;                          //vertex u is now
visited

          for(int v=1;v<=V;v++)

//Update dist[v] for vertex v which is not yet included in visited[]
and

//there is a path from src to v through u that has smaller distance
than

// current value of dist[v]

          {

               if(!visited[v] && cost[u][v]
&&dist[u]+cost[u][v]<dist[v])

               dist[v]=dist[u]+cost[u][v];

          }

     }

      //will print the vertex with their distance from the source

     cout<<"The shortest path  "<<src<<" to all the other vertices is:
\n";

     for(int i=1;i<=V;i++)

     {

        if(i!=src)

        cout<<"source:"<<src<<"\t destination:"<<i<<"\t MinCost
is:"<<dist[i]<<"\n";

     }

}

int main()

{

     int cost[V][V], i,j, s;

     cout<<"\n Enter the cost matrix weights";

     for(i=1;i<=V;i++)      //Indexing ranges from 1 to n

          for(j=1;j<=V;j++)

          {

cin>>cost[i][j];
```

```
                //Absence of edge between vertices i and j is
represented by INFINITY

             if(cost[i][j]==0)

               cost[i][j]=INFINITY;

           }

cout<<"\n Enter the Source Vertex";

cin>>s;



     Dijkstra(cost,s);

     return 0;

}
```

Copy
The program is executed using same input graph as in Fig.1.This will help in verifying the resultant solution set with actual output.



```
 Enter the cost matrix weights
0 9 999 5 999
9 0 1 2 999
999 1 0 9 6
5 2 9 0 4
999 999 6 4 0

 Enter the Source Vertex1
The shortest path from source 1 to all the other vertices is:
source: 1        destination: 2  MinCost is: 7
source: 1        destination: 3  MinCost is: 8
source: 1        destination: 4  MinCost is: 5
source: 1        destination: 5  MinCost is: 9
```

Fig 4: Output

Time Complexity Analysis-

Following are the cases for calculating the time complexity of Dijkstra's Algorithm-

- **Case1**- When graph G is represented using an adjacency matrix -This scenario is implemented in the above **C++** based program. Since the implementation contains two nested for loops, each of complexity **O(n)**, the complexity of Dijkstra's algorithm is **O(n2)**. Please note that n here refers to total number of vertices in the given graph
- **Case 2**- When graph G is represented using an adjacency list - The time complexity, in this scenario reduces to **O(|E| + |V| log |V|)** where |E|represents number of edges and |V| represents number of vertices in the graph

Disadvantages of Dijkstra's Algorithm-

Dijkstra's Algorithm cannot obtain correct shortest path(s)with weighted graphs having negative edges. Let's consider the following example to explain this scenario-

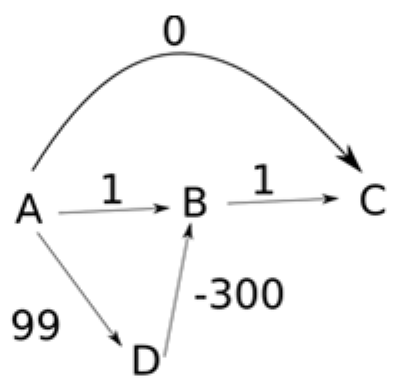


Fig 5: Weighted graph with negative edges

Choosing source vertex as A, the algorithm works as follows-

Step A- Initialize the distance array (dist)-

Set of visited vertices (S)	A	B	C	D
	0	∞	∞	∞

Step B- Choose vertex A as dist[A] is minimum and A is not in S. Visit A and add it to S. For all adjacent vertices of A which have not been visited yet (are not in S) i.e C, B and D, update the distance array

dist[C]= min(dist[C], dist[A]+cost(A, C)) = min(∞, 0+0) = 0

dist[B] = min(dist[B], dist[A]+cost(A, B)) = min(∞, 0+1) = 1

dist[D]= min(dist[D], dist[A]+cost(A, D)) = min(∞, 0+99) = 99

Thus dist[] gets updated as follows-

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99

Step C- Repeat Step B by

a. Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
b. The distance array does not get updated since there are no adjacent vertices of C

Set of visited vertices (S)	A	B	C	D
[A]	0	1	0	99
[A, C]	0	1	0	99

Continuing on similar lines, Step B gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	A	B	C	D

[A]	0	1	0	99
[A, C]	0	1	0	99
[A, C, B]	0	1	0	99
[A, C, B, D]	0	1	0	99

Thus, following are the shortest distances from A to B, C and D-

A->C = 0 A->B = 1 A->D = 99

But these values are not correct, since we can have another path from **A** to **C, A->D->B->C** having **total cost= -200** which is smaller than 0. This happens because once a vertex is visited and is added to the set S, it is never "looked back" again. Thus, Dijkstra's algorithm does not try to find a shorter path to the vertices which have already been added to S.

- It performs a blind search for finding the shortest path, thus, consuming a lot of time and wasting other resources

Applications of Dijkstra's Algorithm-

- Traffic information systems use Dijkstra's Algorithm for tracking destinations from a given source location
- **Open Source Path First (OSPF)**, an Internet-based routing protocol, uses Dijkstra's Algorithm for finding best route from source router to other routers in the network
- It is used by **Telephone and Cellular networks** for routing management
- It is also used by **Geographic Information System (GIS)**, such as Google Maps, for finding shortest path from point A to point B