Unit - III
Greedy Algorithms

Contents

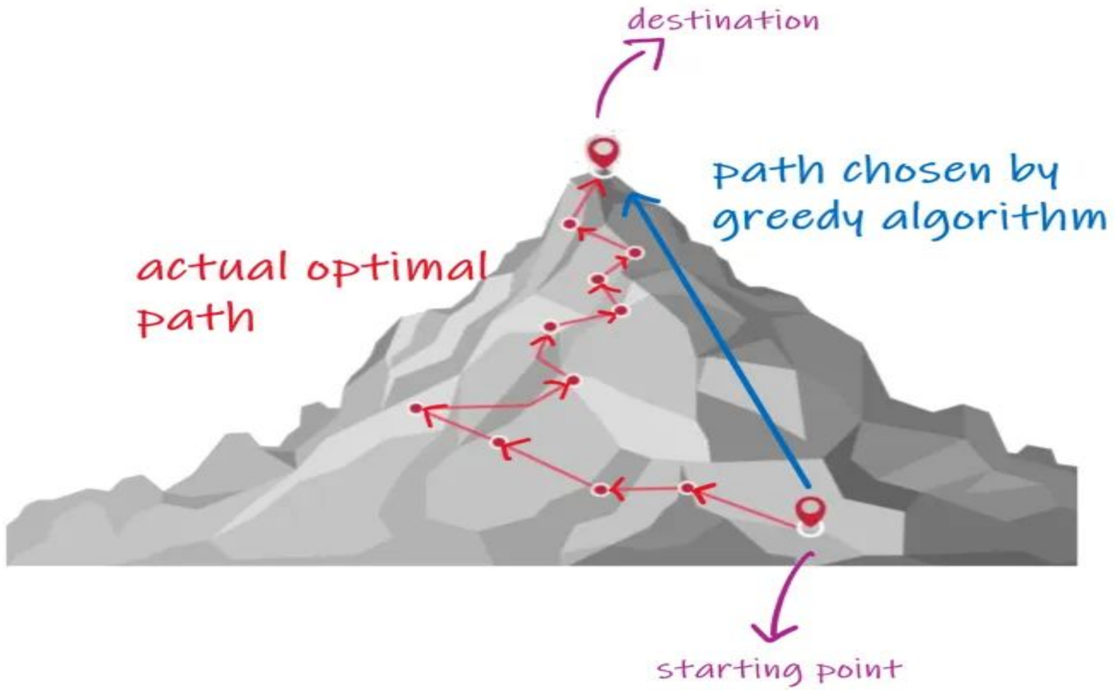
Introduction to Greedy Technique

- "Greedy Method finds out of many options, but you have to **choose the best option**."
- In this method, we have to find out the **best method/option out of many present ways**.
- In this approach/method we focus on the first stage and decide the output, **don't think about the future**.

Greedy Method

- Greedy Algorithm solves problems by **making the best choice** that seems best at the particular moment.
- Many **optimization problems** can be determined using a greedy algorithm.
- **Some issues have no efficient solution**, but a greedy algorithm may **provide a solution that is close to optimal**.
- A greedy algorithm works if a problem exhibits the following **two properties**:
  a. **Greedy Choice Property**: A **globally optimal solution** can be reached at by creating a **locally optimal solution**. In other words, an optimal solution can be obtained by creating "greedy" choices.
  b. **Optimal substructure**: Optimal solutions **contain optimal subsolutions**. In other words, answers to subproblems of an optimal solution are optimal.

Example 1



An optimal path may be a little twisted and longer, but it may take up less time as it was easy to complete.

Example 2

Physics lesson
(1 hour)

Tennis lesson
(3 hours)

Chemistry class
(2 hour)

Cooking class
(4 hours)

Cricket coaching
(5 hours)

“If I want to do maximum lessons, I should take up lessons that take minimum amount of time.”

Applications

1. Optimal Merge Patterns
2. Huffman Coding
3. Knapsack Problem
4. Activity Selection Problem
5. Job Sequencing with Deadline
6. Minimum Spanning Tree
7. Single-Source Shortest Path Algorithm

Components of Greedy Algorithm

Greedy algorithms have the following **five** components -

1. **A candidate set** - A solution is created from this set.
2. **A selection function** - Used to choose the best candidate to be added to the solution.
3. **A feasibility function** - Used to determine whether a candidate can be used to contribute to the solution.
4. **An objective function** - Used to assign a value to a solution or a partial solution.
5. **A solution function** - Used to indicate whether a complete solution has been reached.

Steps for achieving a Greedy Algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
3. **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

Dijkstra's Algorithm

Single-Source Shortest Path Algorithm

- It is a greedy algorithm that solves the single-source shortest path problem for a **directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.**
- Dijkstra's Algorithm maintains a **set S of vertices** whose final shortest-path weights **from the source s** have already been determined.That's for all vertices **v ∈ S;**

  we have d [v] = δ (s, v).

- The algorithm repeatedly **selects the vertex u ∈ V - S with the minimum shortest** - path estimate, **insert u into S and relaxes all edges leaving u.**
- **Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the greedy strategy.**

Steps of the Algorithm

1. Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity.

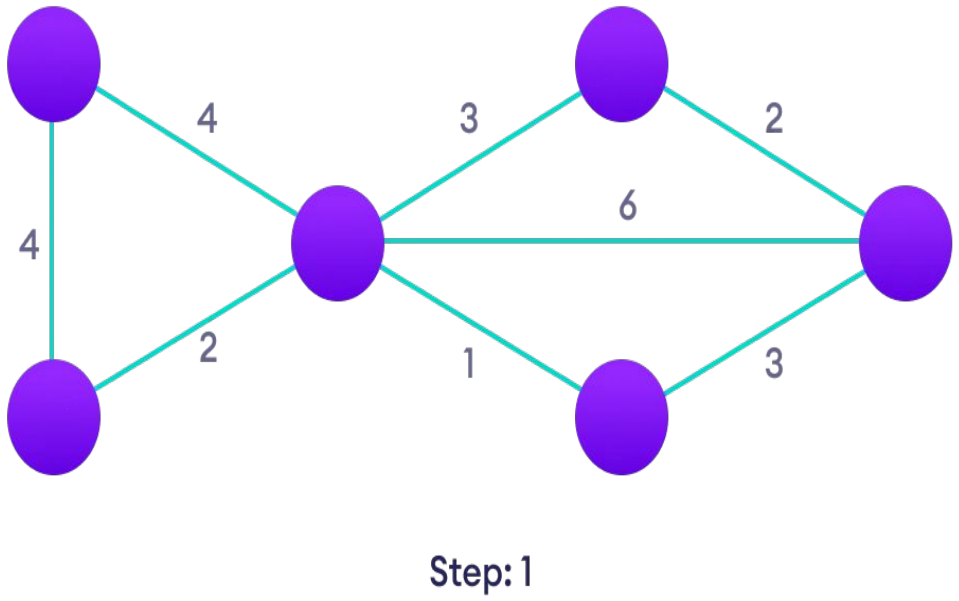
2. Array visited[ ] is initialized to zero.

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.

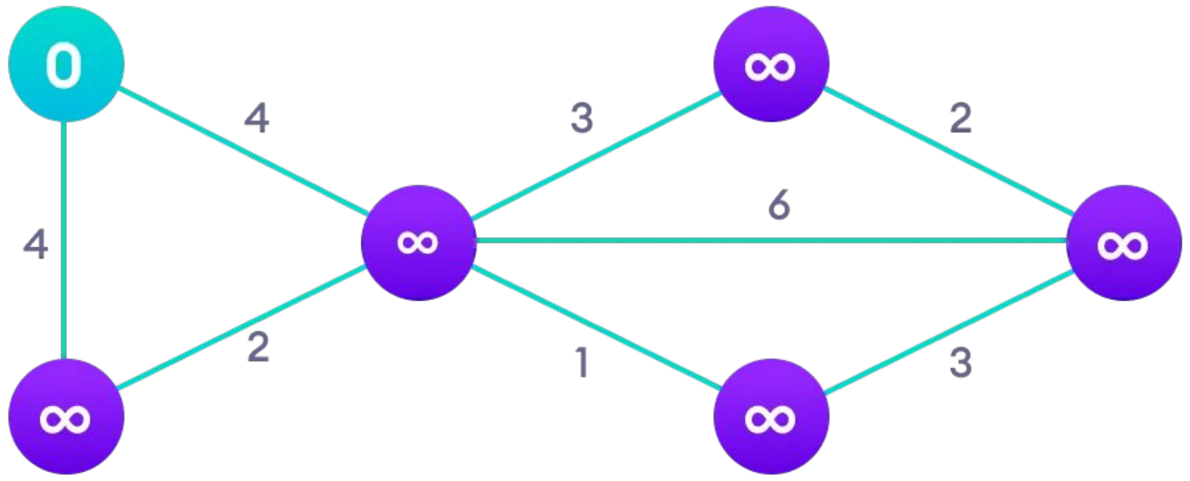
4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex 0. Initially, distance of source vertex is taken as 0. i.e. distance[0]=0;

5. for(i = 1; i < n; i++)
– Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.
– Recalculate the shortest distance of remaining vertices from the source.
– Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v
if(visited[v]==0) distance[v]=min(distance[v], distance[w]+cost[w][v])

Example of Dijkstra's algorithm

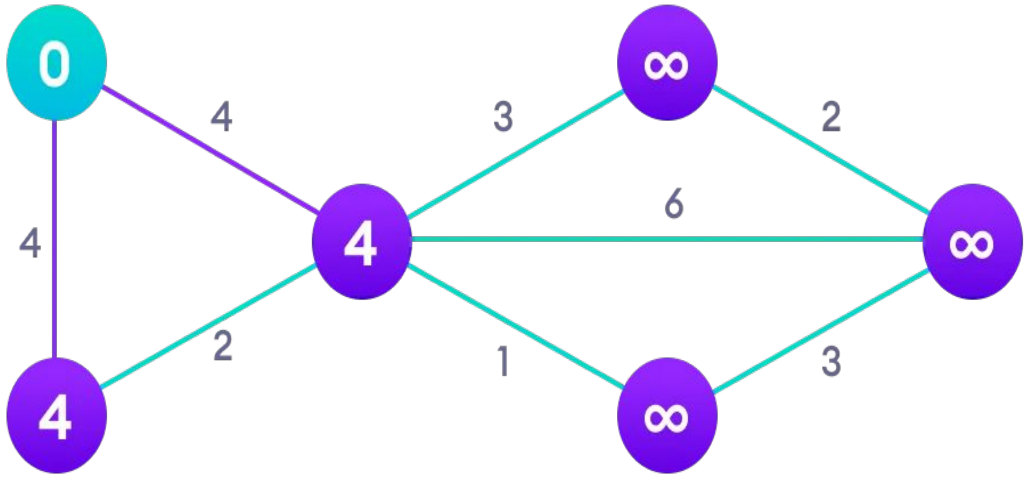


Start with a weighted graph



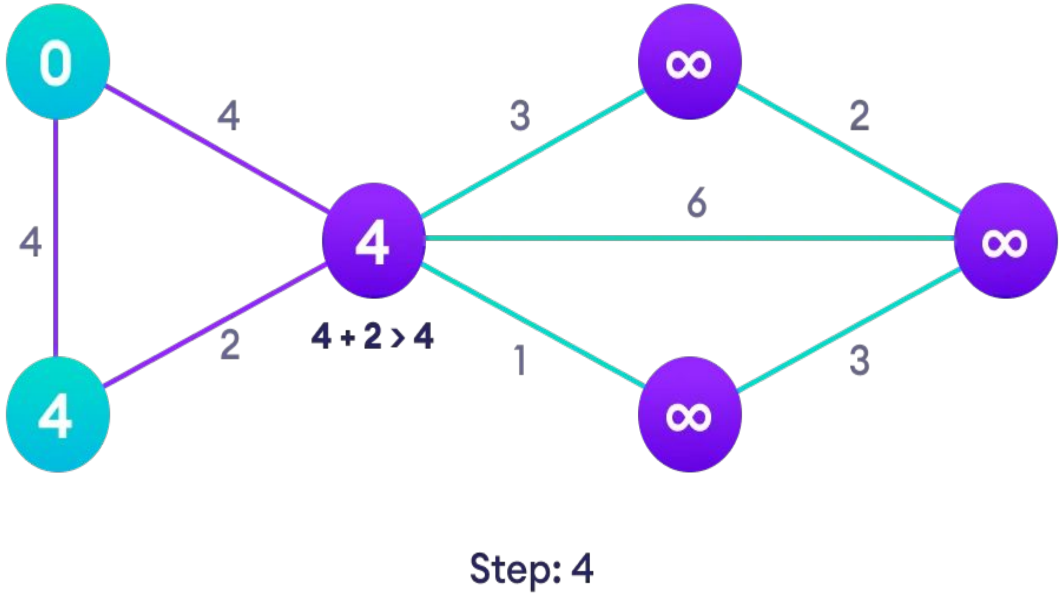
Step: 2

Choose a starting vertex and assign infinity path values to all other devices

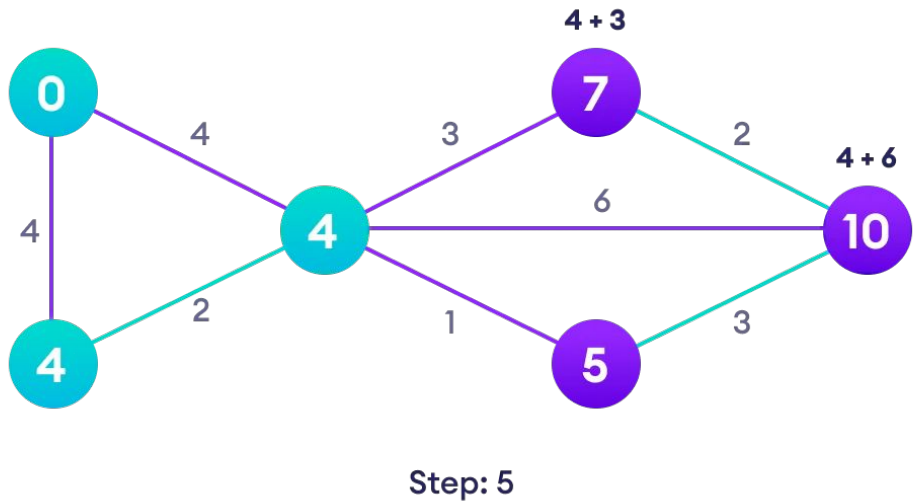


Step: 3

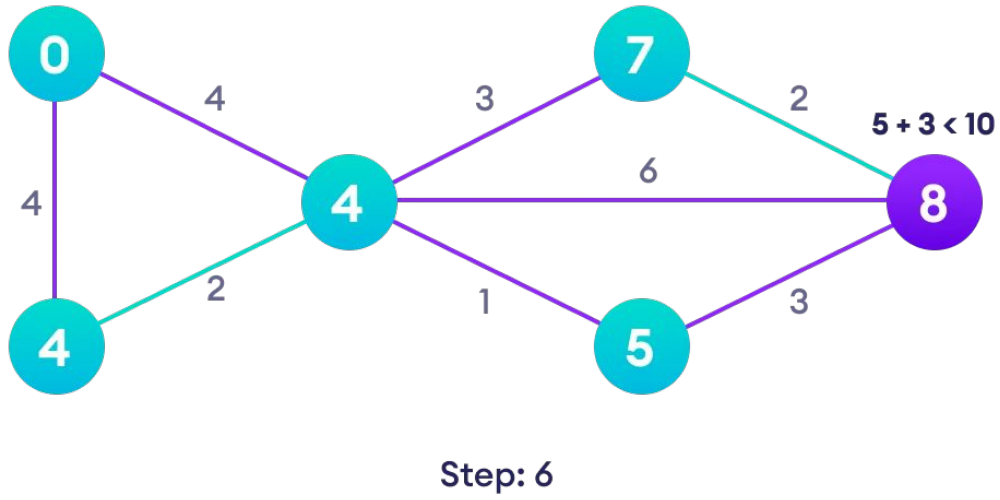
Go to each vertex and update its path length



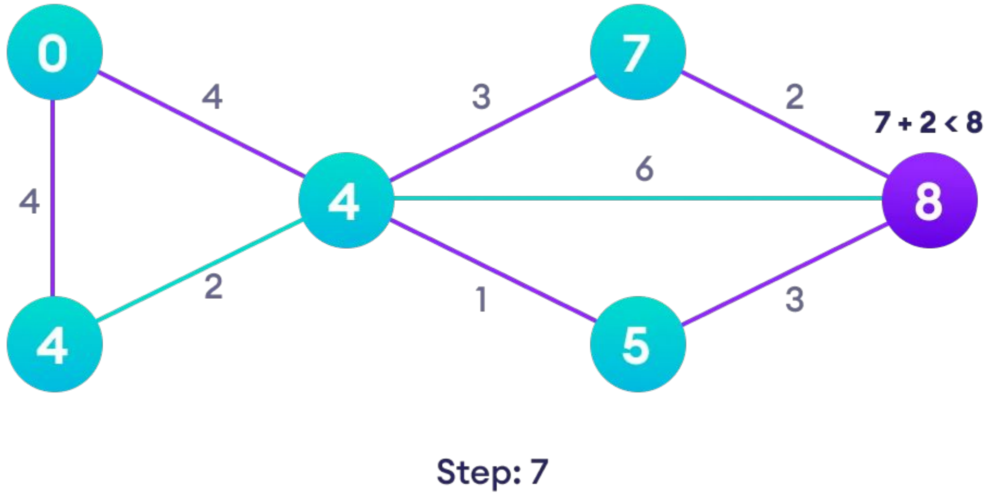
If the path length of the adjacent vertex is lesser than new path length, don't update it



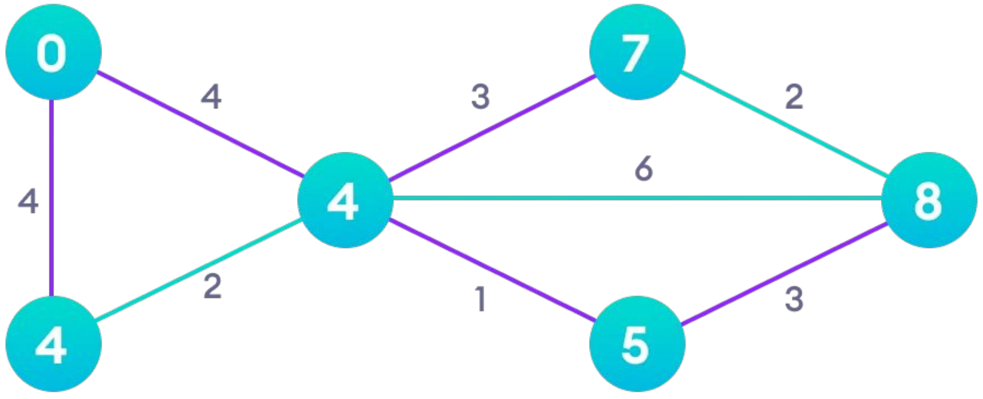
Avoid updating path lengths of already visited vertices



After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Time Complexity Analysis of Dijkstra’s Algorithm

Case-01:

This case is valid when-

- The given graph G is represented as an adjacency matrix.
- Priority queue Q is represented as an unordered list.

Here,

- A[i,j] stores the information about edge (i,j).
- Time taken for selecting i with the smallest dist is O(V).
- For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.
- Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.
- Thus, total time complexity becomes **O(V^2)**.

Time Complexity Analysis of Dijkstra’s Algorithm

Case-02:

This case is valid when-

- The given graph G is represented as an adjacency list.
- Priority queue Q is represented as a binary heap.

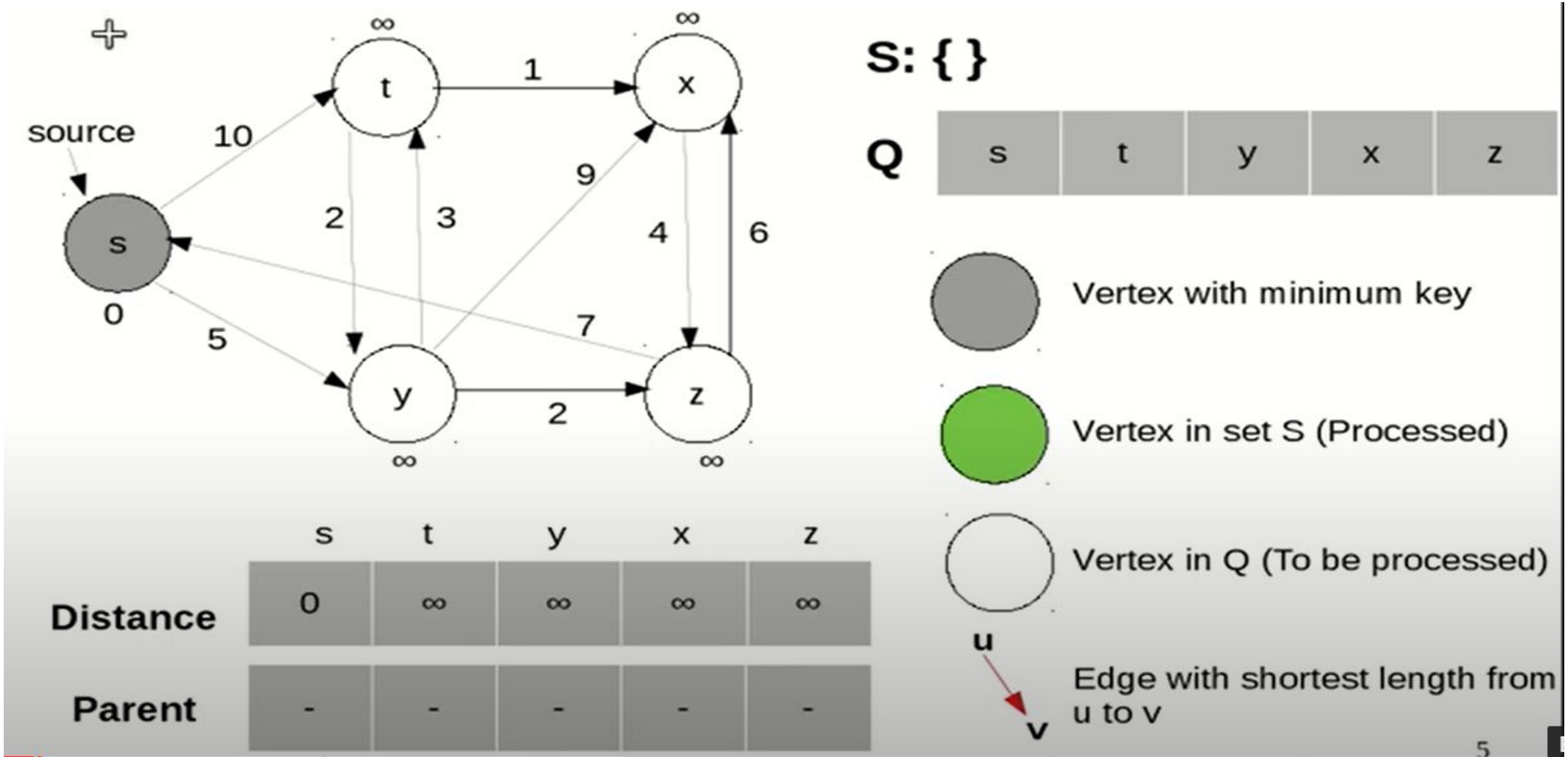
Here,

- With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.
- In min heap, operations like extract-min and decrease-key value takes O(logV) time.
- So, overall time complexity becomes O(E+V) x O(logV) which is O((E + V) x logV) = **O(ElogV)**

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.
- For example, The element with the highest value is considered as the highest priority element.
- However, in other cases, we can assume the element with the lowest value as the highest priority element.
- In other cases, we can set priorities according to our needs.

Initial Step



S: { }

Q				
s	t	y	x	z

● Vertex with minimum key

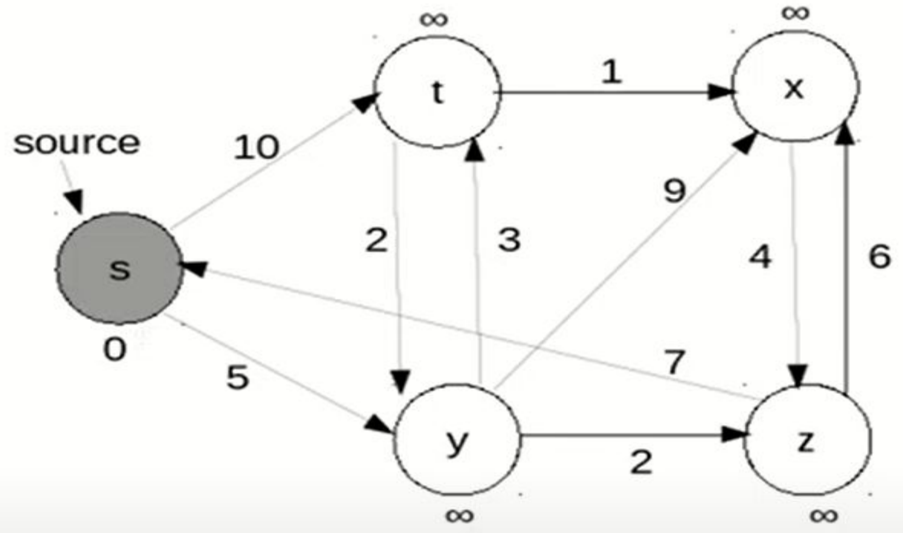
🟢 Vertex in set S (Processed)

○ Vertex in Q (To be processed)

$u \to v$ Edge with shortest length from u to v

	s	t	y	x	z
Distance	0	∞	∞	∞	∞
Parent	-	-	-	-	-

Working



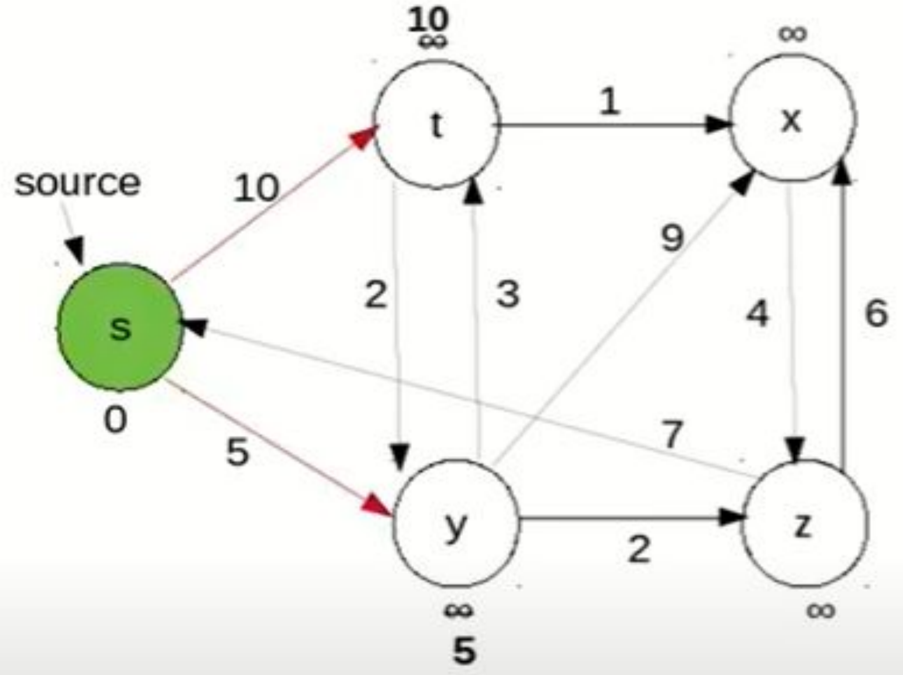
S: { }

Q

s	t	y	x	z

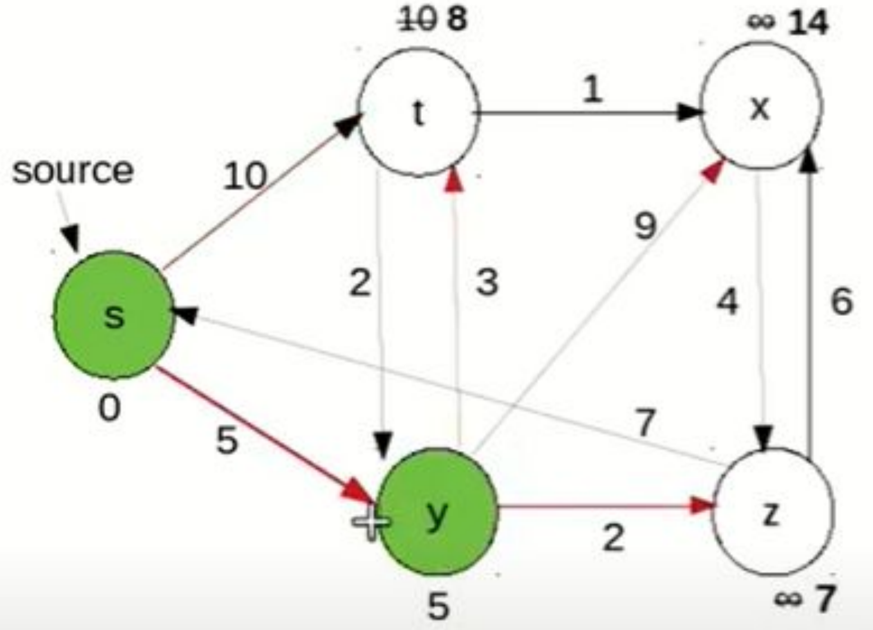
Idea:
1. Extract Minimum from Q (*say u*)
2. For each neighbour (v) of u
   Update distance of v from u

	s	t	y	x	z
Distance	0	∞	∞	∞	∞
Parent	-	-	-	-	-



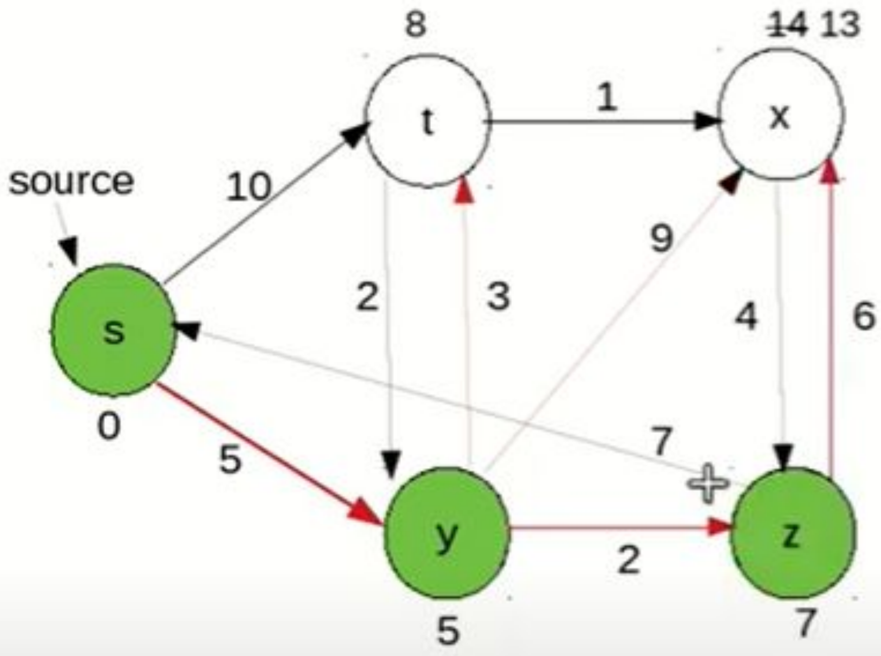
S	s			
Q	t	y	x	z

	s	t	y	x	z
Distance	0	10	5	∞	∞
Parent	-	s	s	-	-



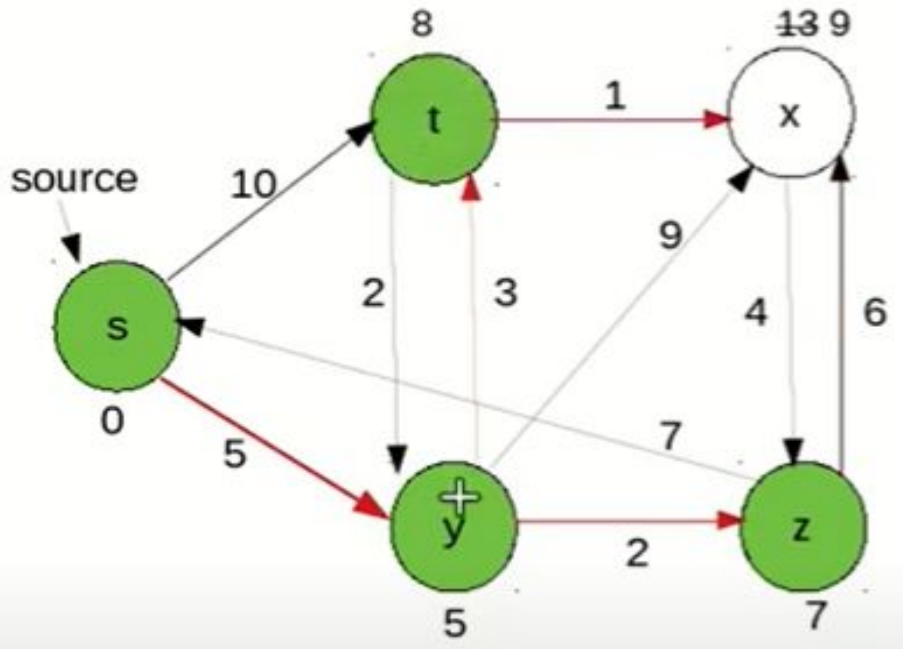
S	s	y	
Q	t	x	z

	s	t	y	x	z
Distance	0	8	5	14	7
Parent	-	y	s	y	y



S	s	y	z
Q	t	x	

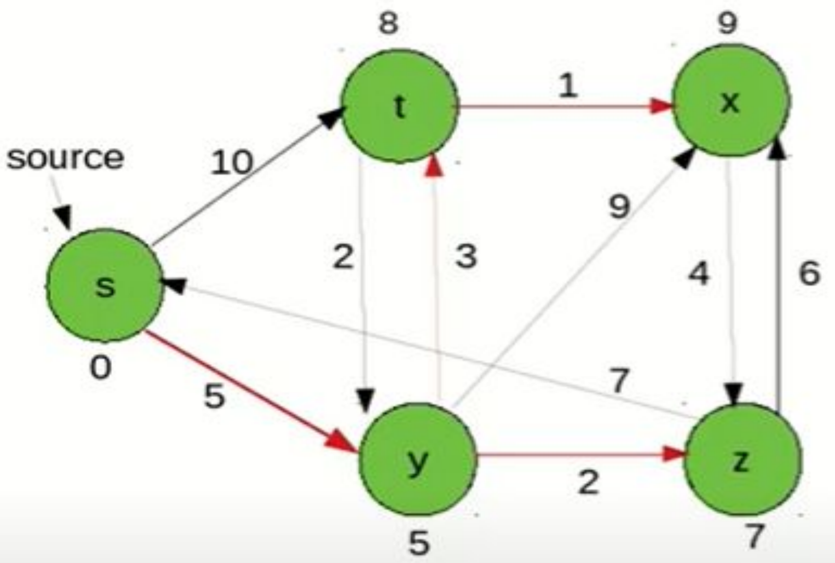
	s	t	y	x	z
Distance	0	8	5	13	7
Parent	-	y	s	z	y



S	s	y	z	t
Q	x			

	s	t	y	x	z
Distance	0	8	5	9	7
Parent	-	y	s	t	y

Working of Dijkstra's



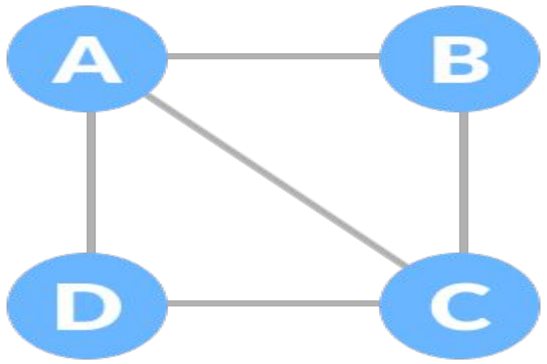
S	s	y	z	t	x

Q: { }

	s	t	y	x	z
Distance	0	8	5	9	7
Parent	-	y	s	t	y

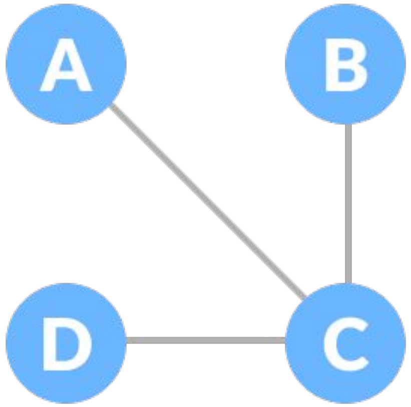
Spanning Tree and Minimum Spanning Tree

- Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.
- An undirected graph is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Connected graph

A connected graph is a graph in which there is always a path from a vertex to any other vertex.

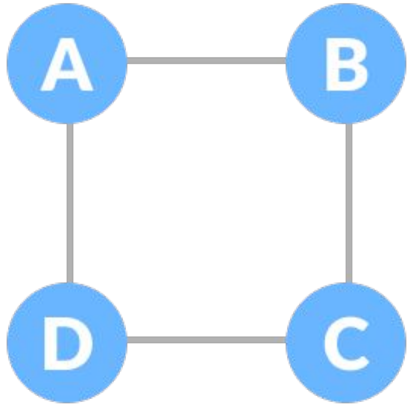


Spanning tree

- 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.
- The total number of spanning trees with n vertices that can be created from a **complete graph is equal to n^(n-2).**
- If we have n = 4, the maximum number of possible spanning trees is equal to **4^(4-2) = 16.**
- Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

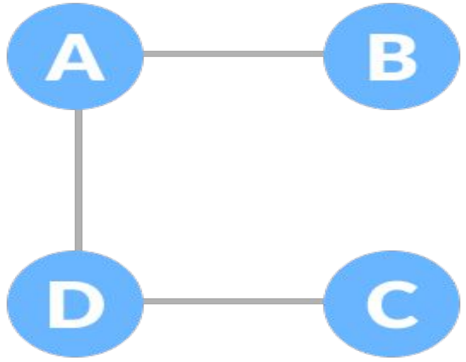
Example of a Spanning Tree

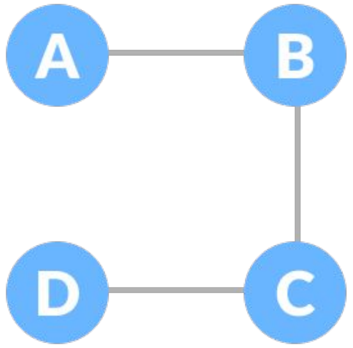
`Let the original graph be:`

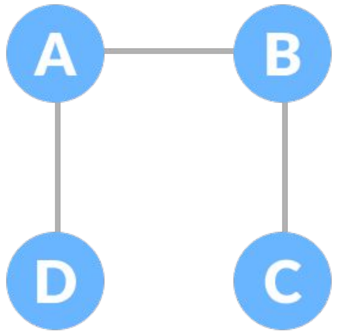


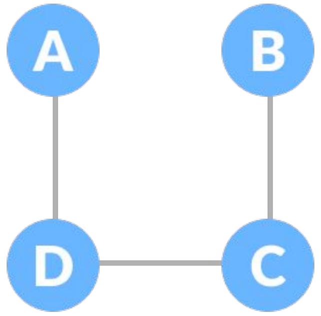
Normal graph

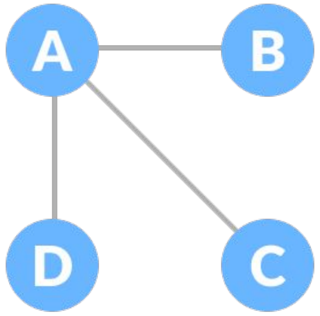
Some of the possible spanning trees that can be created from the above graph are:

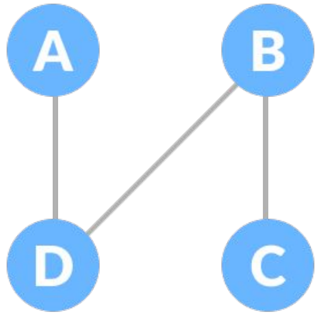






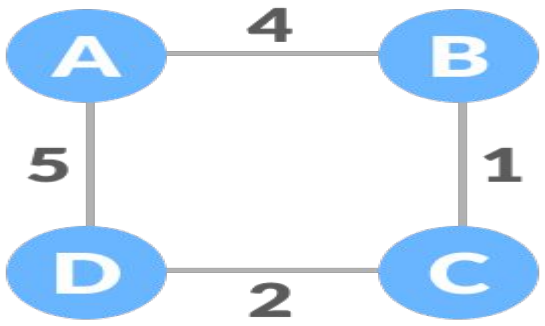






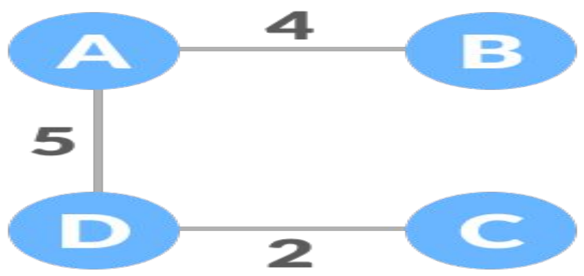
Minimum Spanning Tree

- A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.
- Example of a Spanning Tree
- Let's understand the above definition with the help of the example below.
- The initial graph is:

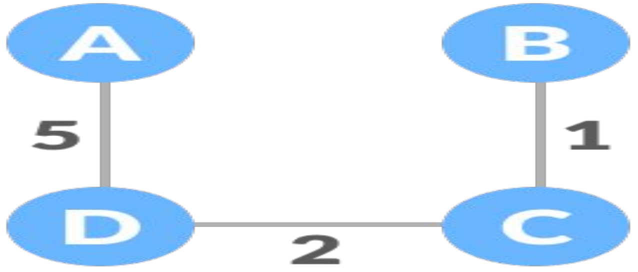


Weighted graph

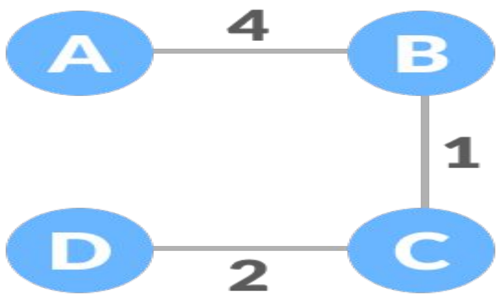
The possible spanning trees from the above graph are:



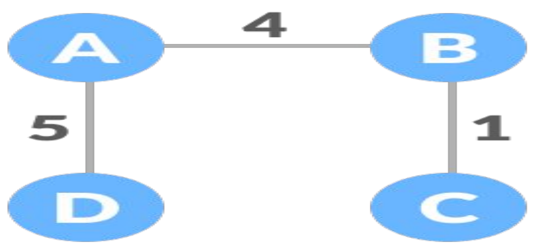
sum = 11



sum = 8

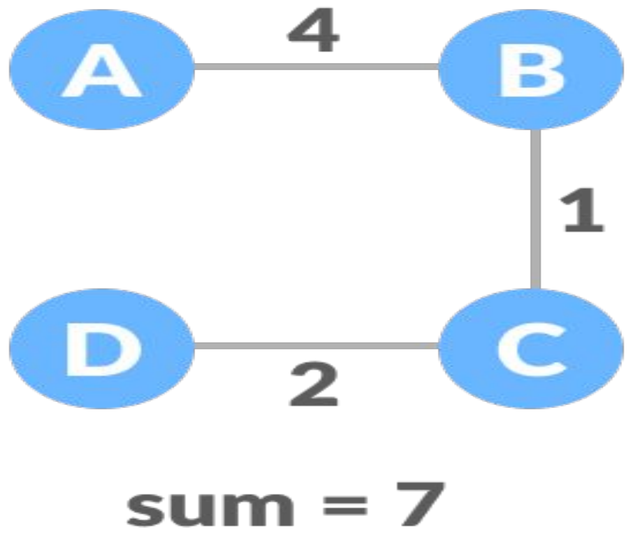


sum = 7



sum = 10

The minimum spanning tree from the above spanning trees is:



Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum **e- n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum **n^n-2** number of spanning trees.
- Thus, we can conclude that spanning trees are a **subset** of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are -

1. Civil Network Planning
2. Computer Network Routing Protocol
3. Cluster Analysis

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here

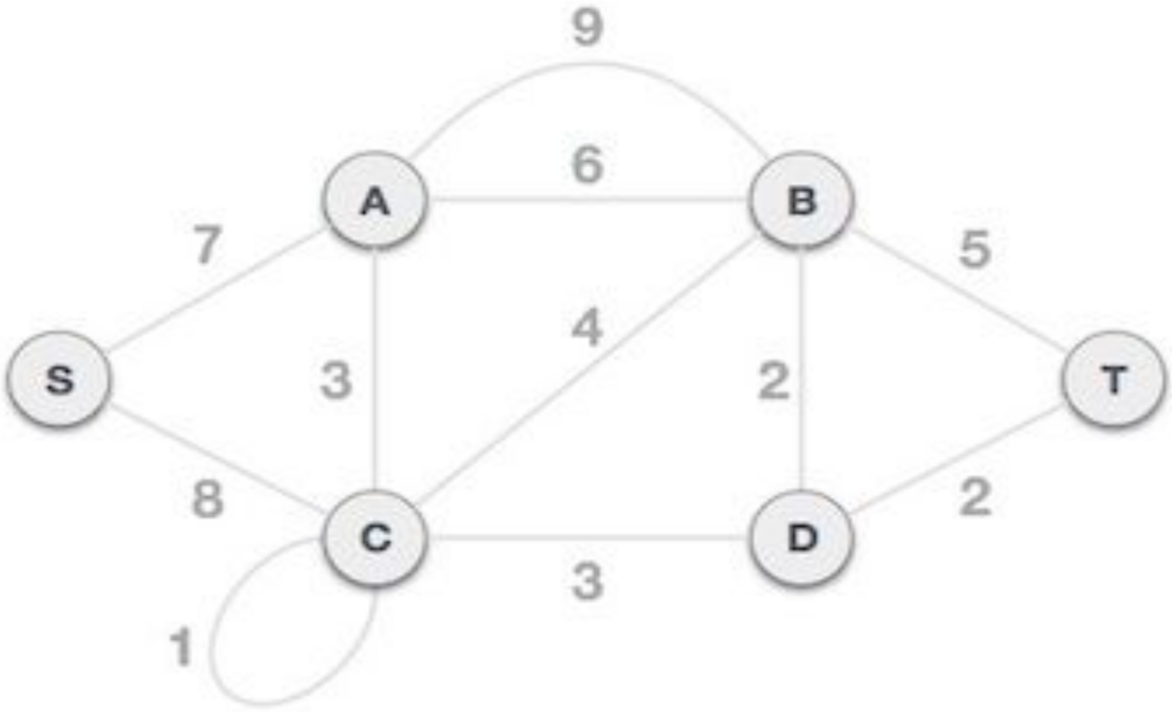
1. Kruskal's Algorithm
2. Prim's Algorithm

Both are greedy algorithms.

Kruskal's algorithm

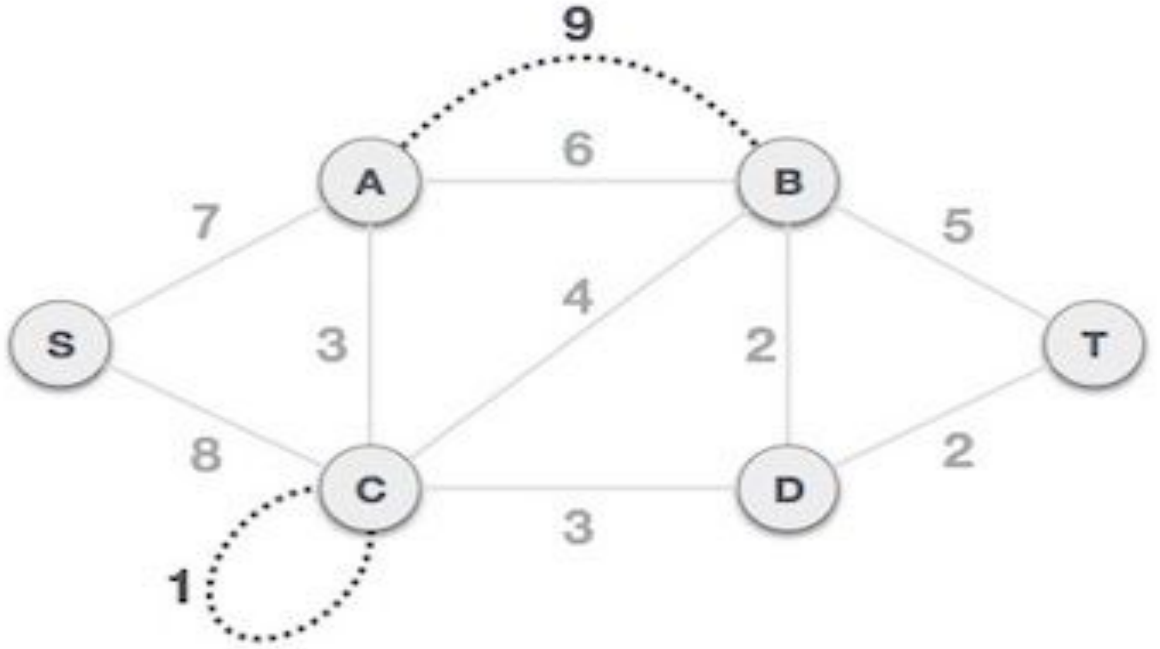
- Kruskal's algorithm to find the **minimum cost spanning tree** uses the greedy approach.
- This algorithm treats the graph as a forest and every node it has as an individual tree.
- A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.
- It takes a graph as input and finds the subset of the edges of that graph which
- **Form a tree that includes every vertex**
- Has the minimum **sum of weights among all the trees that can be formed from the graph**

Example

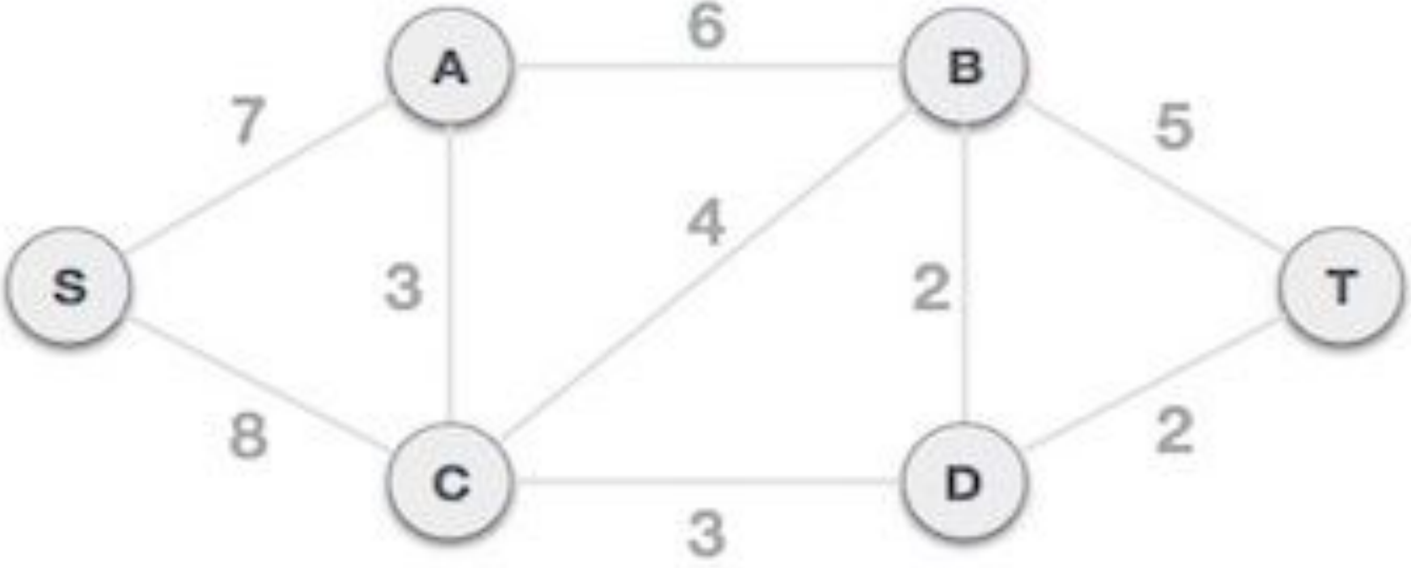


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.

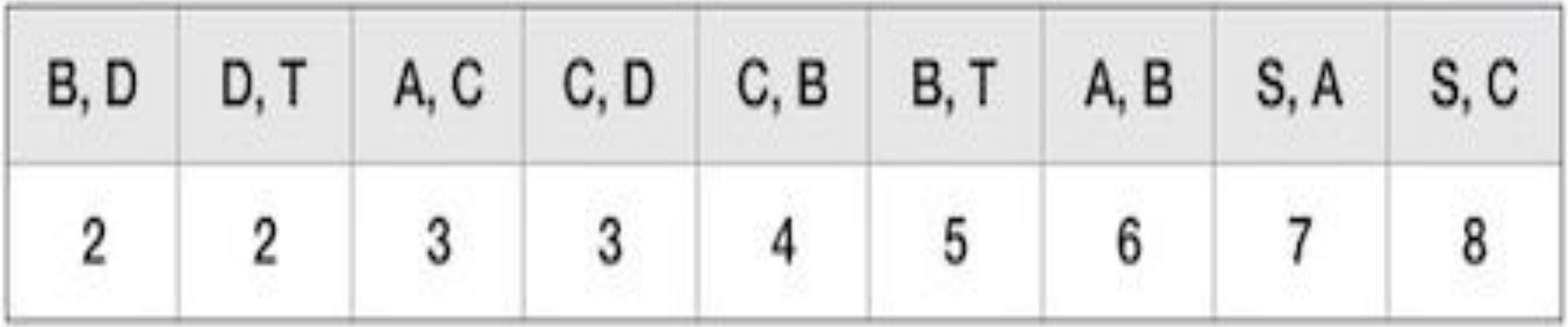


In case of parallel edges, keep the one which has the **least cost associated and remove all others.**


A
B
S
C
D
T
6
7
5
4
3
2
8
2
3

Step 2 - Arrange all edges in their increasing order of weight

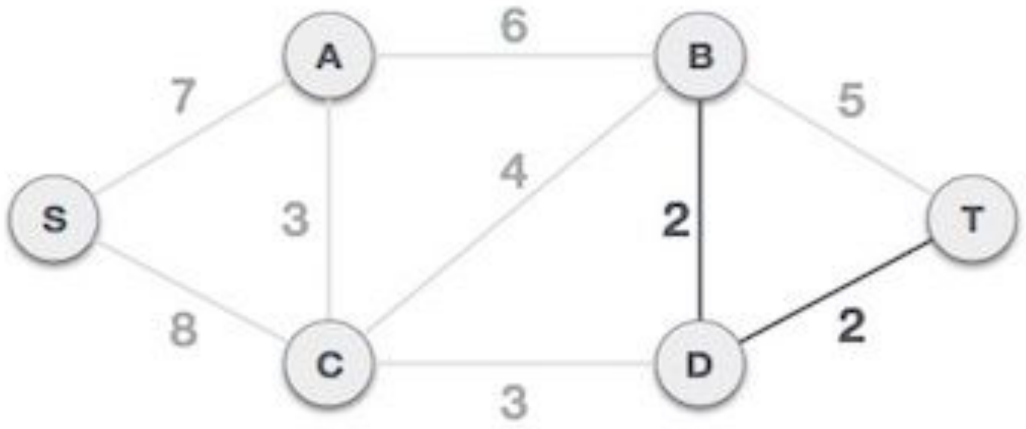
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



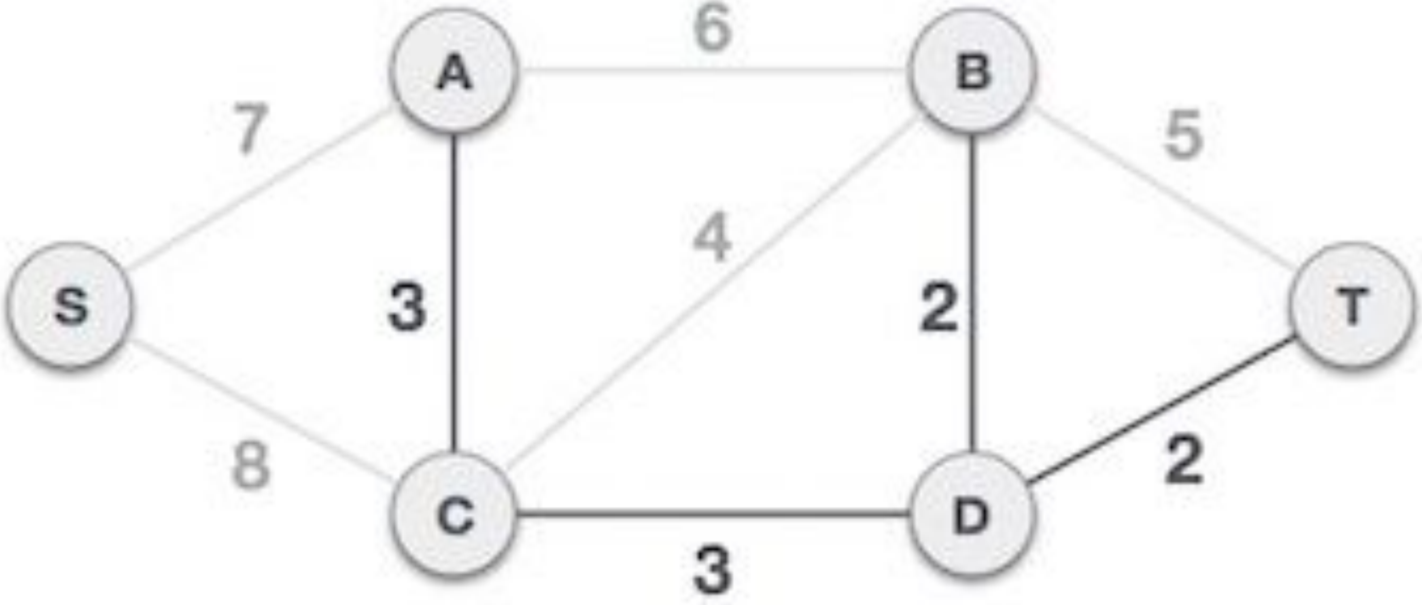
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

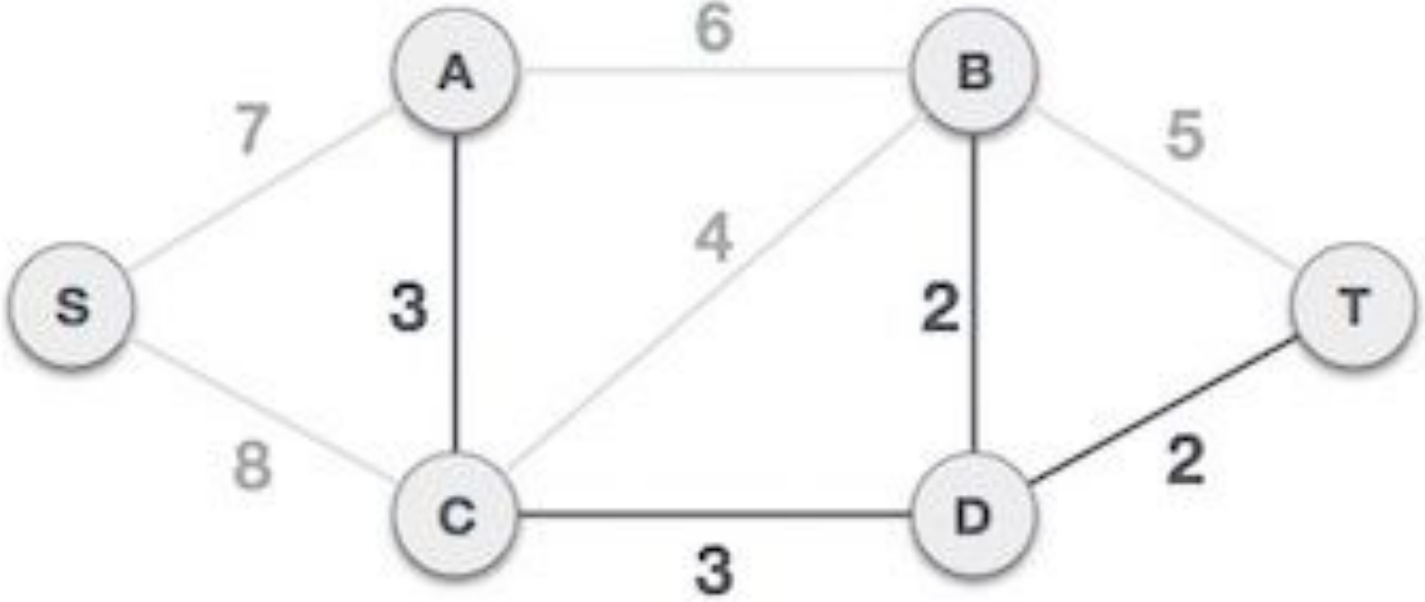
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



Next cost is 3, and associated edges are A,C and C,D. We add them again –

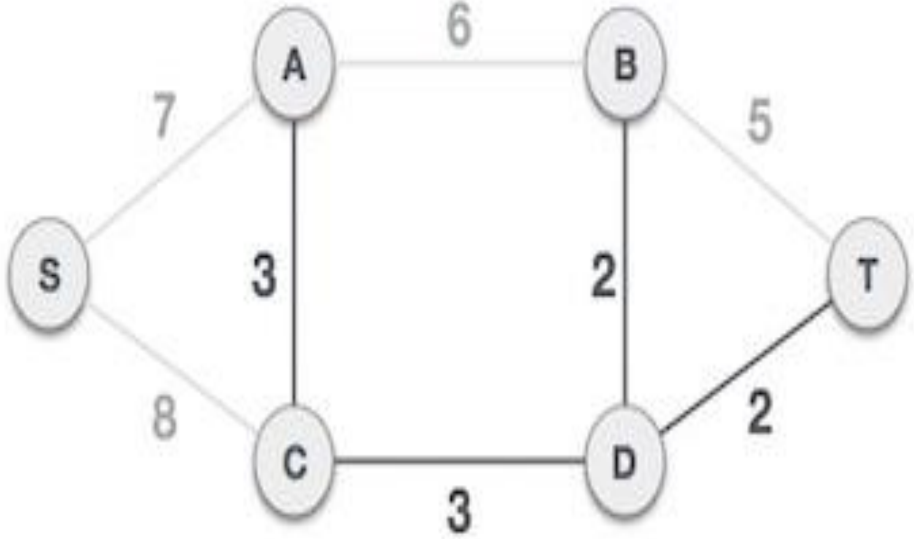


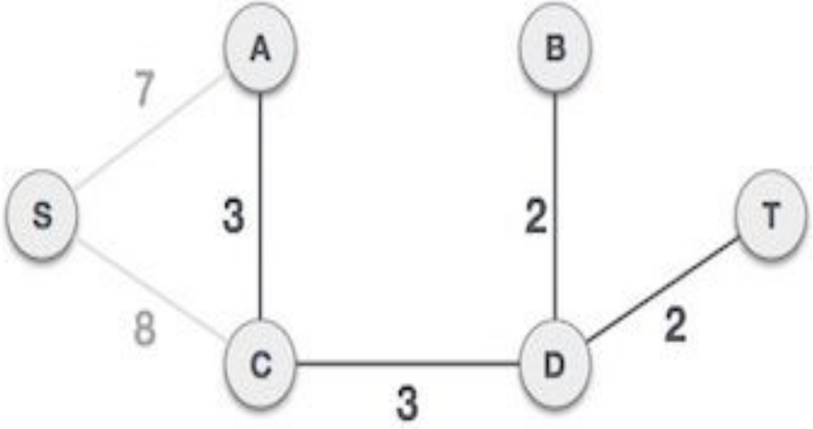
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



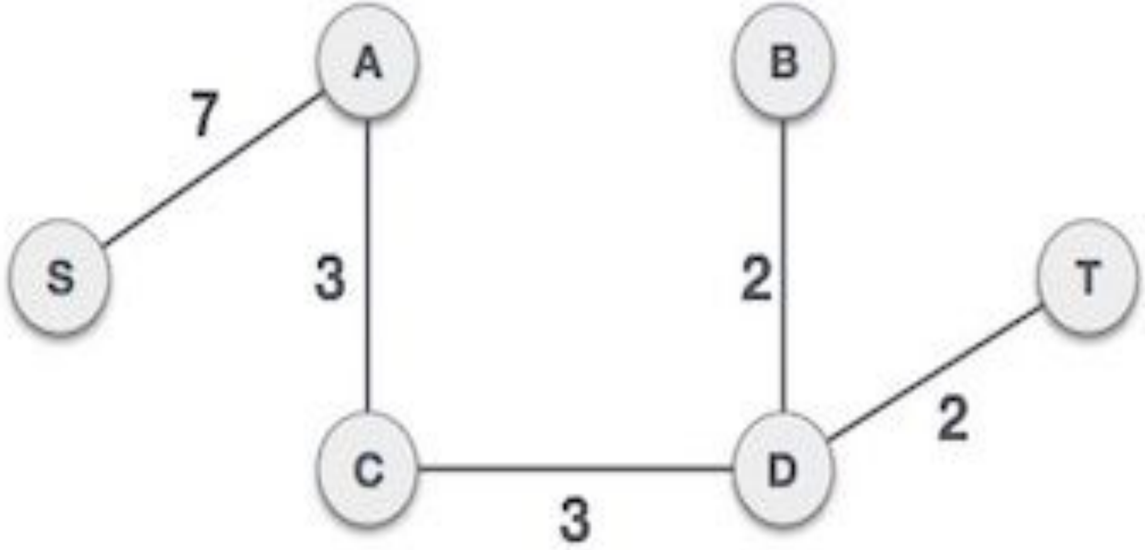
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.





Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



Kruskal's Algorithm Applications

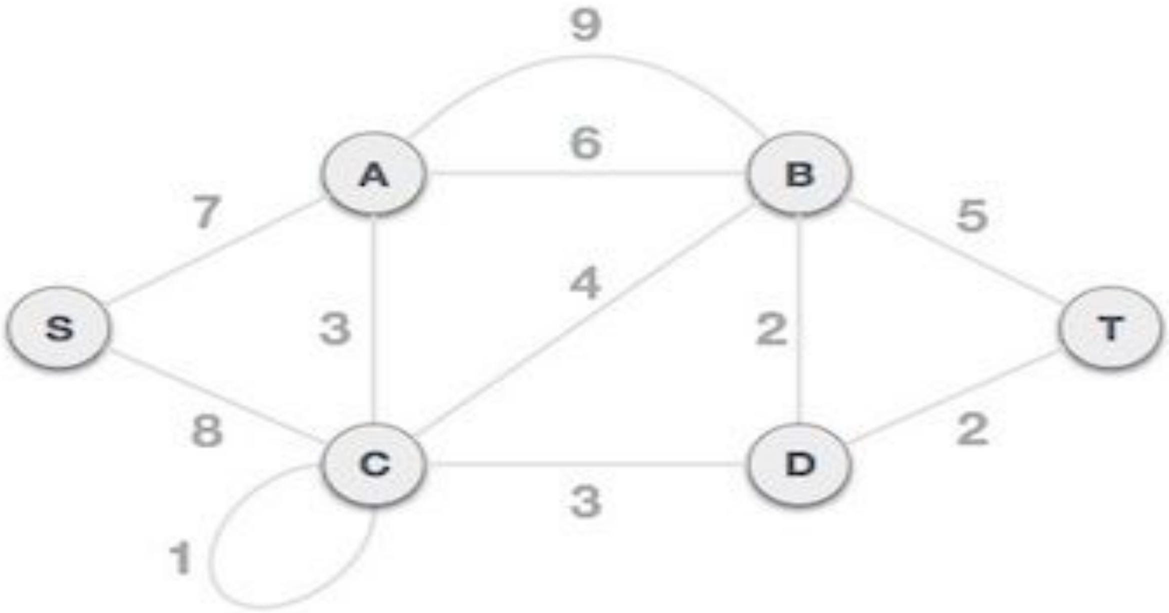
1. In order to layout electrical wiring
2. In computer network (LAN connection)

Kruskal's Algorithm Complexity

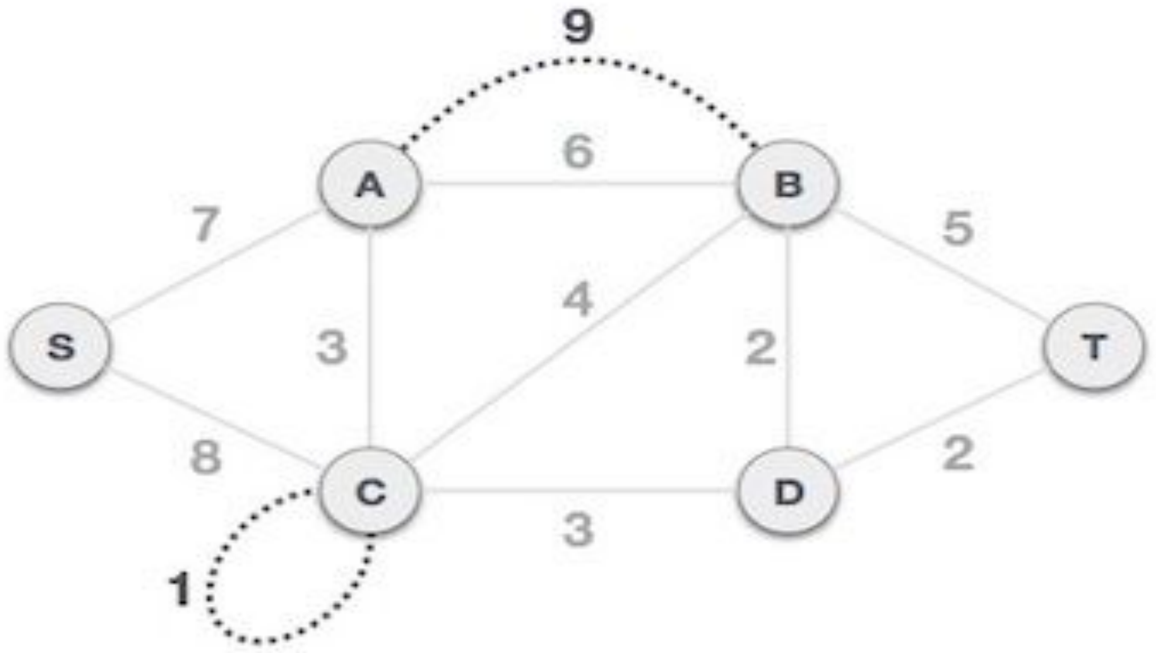
The time complexity Of Kruskal's Algorithm is: O(E log E).

Prim's algorithm

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the **greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.**
- Prim's algorithm, in contrast with Kruskal's algorithm, **treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.**
- To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −


9
6
7
5
4
3
2
8
2
3
1
S
A
B
C
D
T

Step 1 - Remove all loops and Parallel Edges
Remove all loops and parallel edges from the given graph.



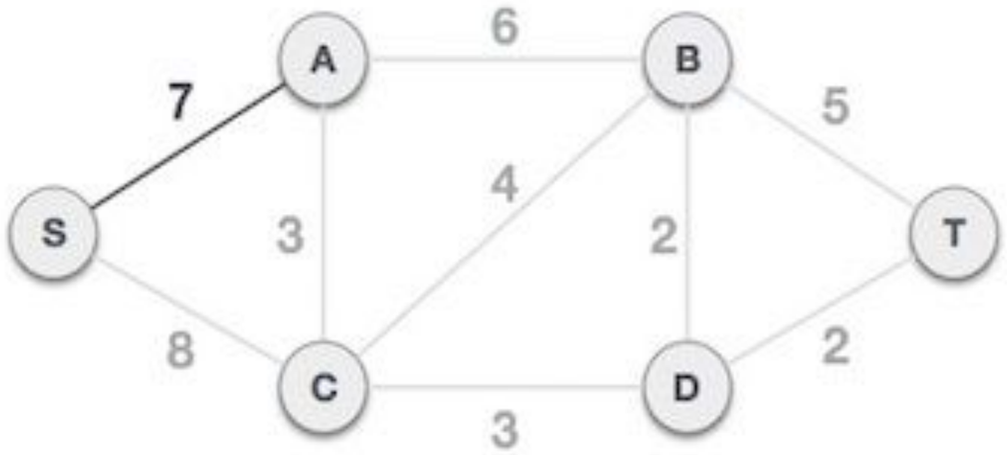
In case of parallel edges, keep the one which has the **least cost associated and remove all others.**

Step 2 - Choose any arbitrary node as root node

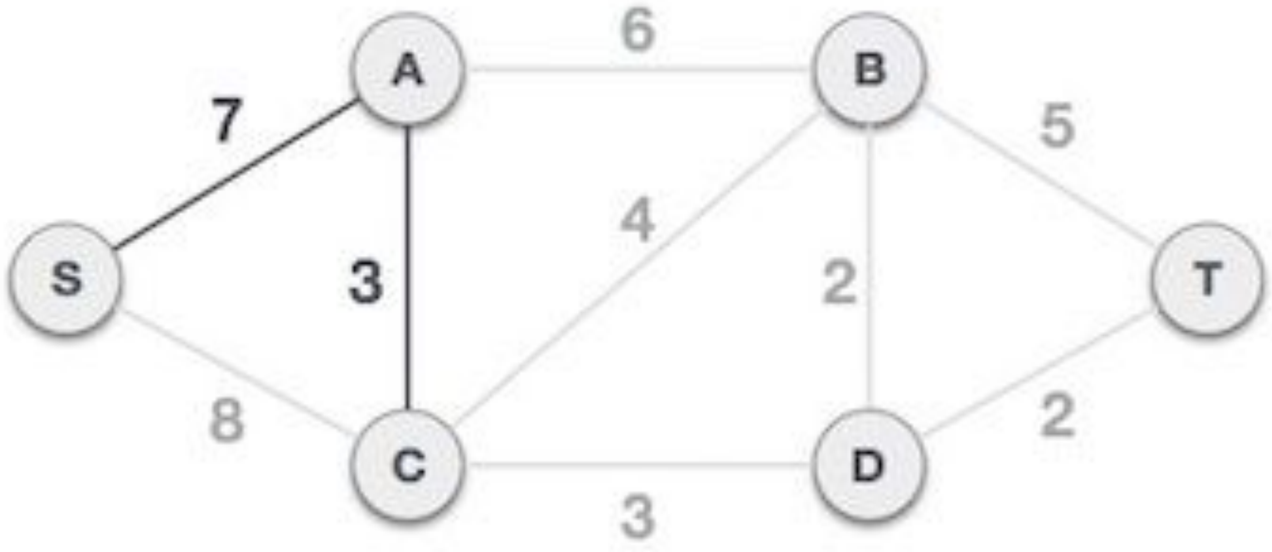
- In this case, we choose S node as the root node of Prim's spanning tree.
- This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node.
- So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

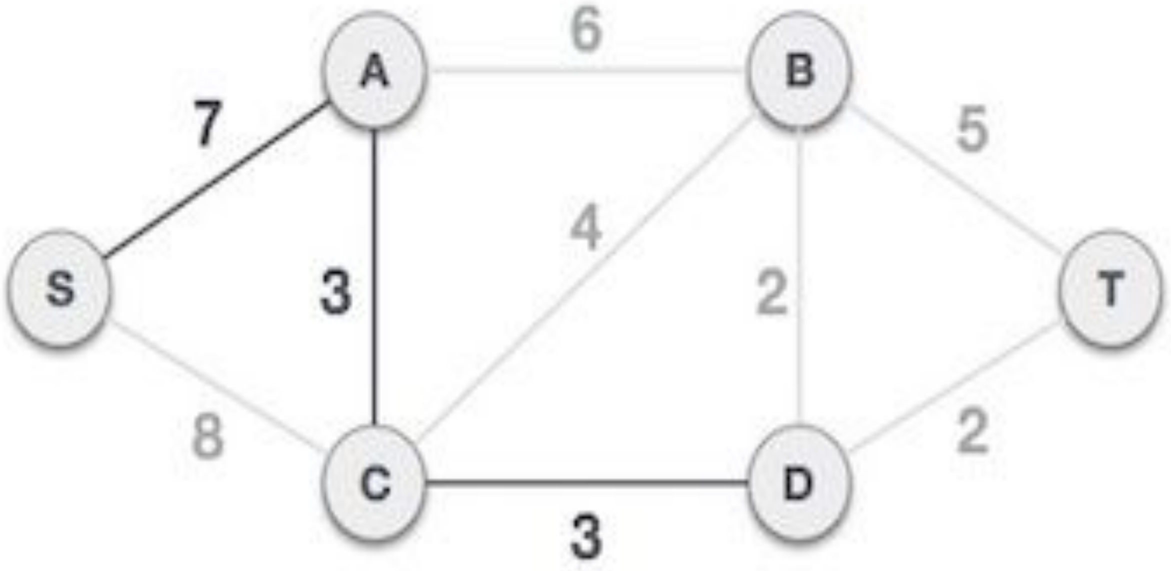
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



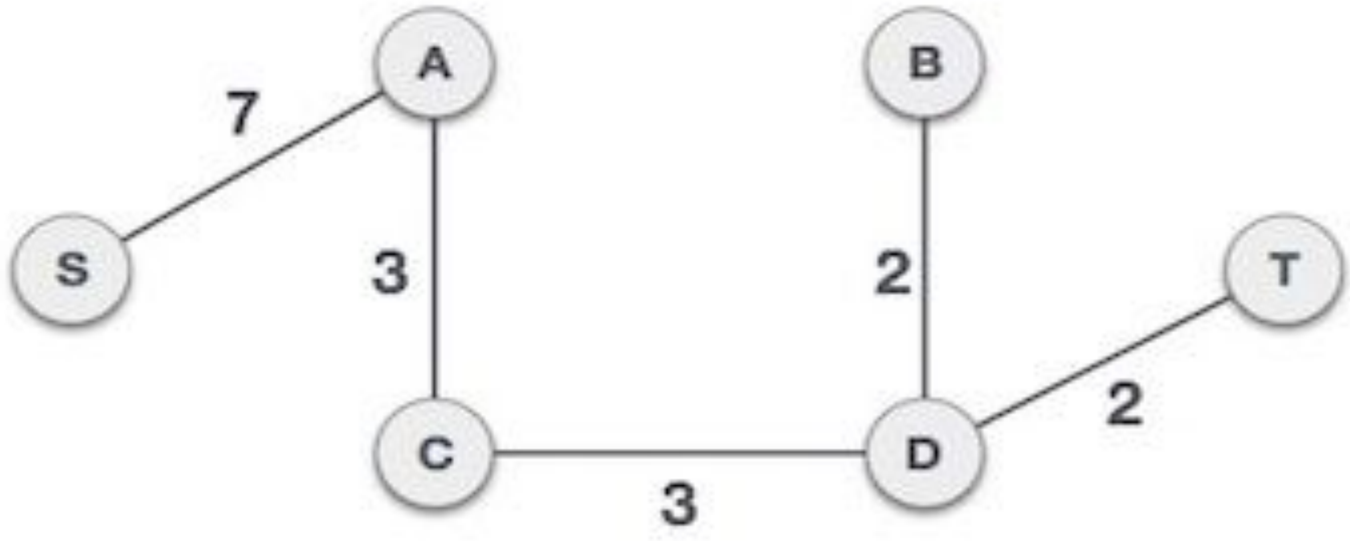
After Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

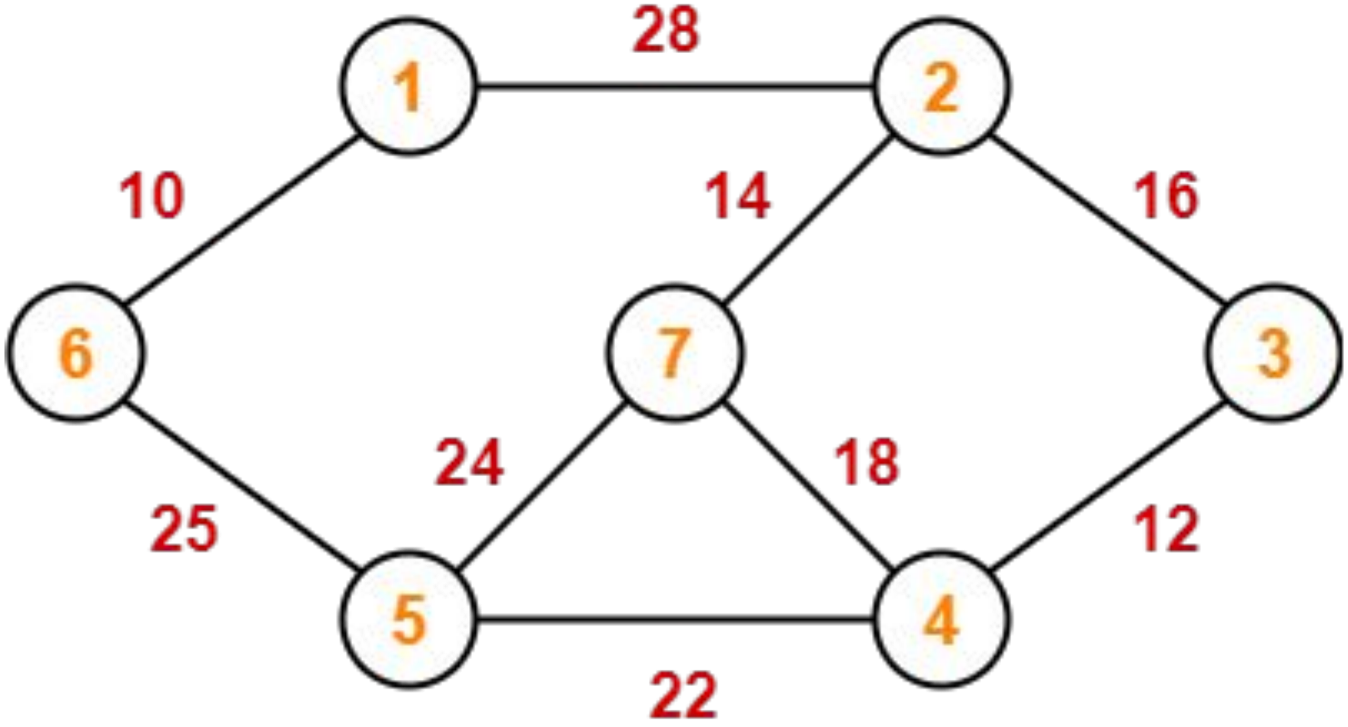


Prim's Algorithm Complexity

The time complexity of Prim's algorithm is **O(E log V)**.

Prim's Algorithm Application

1. Laying cables of electrical wiring
2. In network designed
3. To make protocols in network cycles


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

Optimal Merge Patterns

- Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file.
- Here, we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time O (n+m).
- The formula of external merging cost is:
- Where, **f (i)** represents the number of records in each file and **d (i)** represents the depth.

$$\sum_{i=1}^{n} f(i)d(i)$$

- If more than **2 files need to be merged** then it can be done in pairs.
- For example, **if need to merge 4 files A, B, C, D. First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.**
- An optimal merge pattern corresponds to a **binary merge tree with minimum weighted external path length.**
- The function tree algorithm uses the **greedy rule to get a two- way merge tree** for n files.
- For Ex. **Given 3 files with size 2,3,4 units.**Find optimal way to combine these files then,**pick two smallest numbers** and repeat this until we left with only one number.

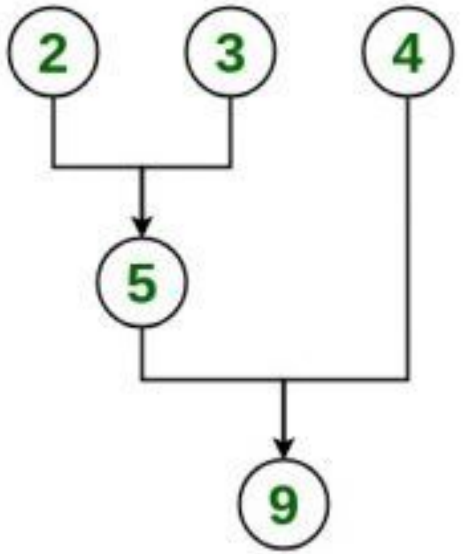
Example:1
Given 3 files with size 2, 3, 4 units.Find optimal way to combine these files

Input: n = 3, size = {2, 3, 4}

Output: 14

Explanation: There are different ways to combine these files:

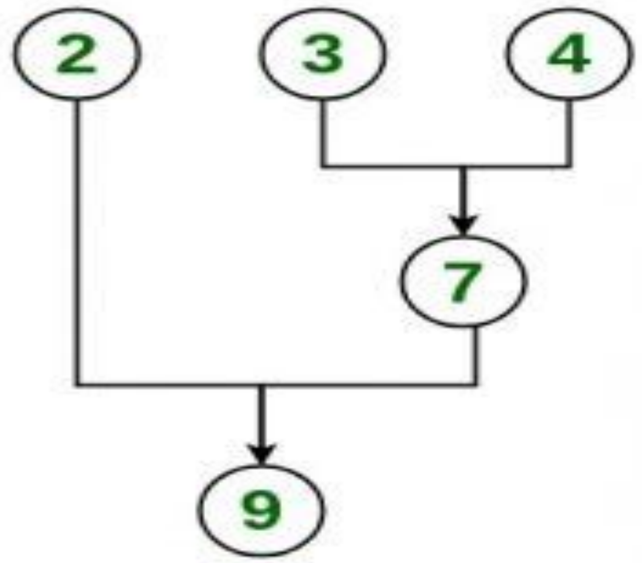
Method 1: Optimal method



Complexity - Cost = 5 + 9 = 14

Example:1
Given 3 files with size 2, 3, 4 units.Find optimal way to combine these files

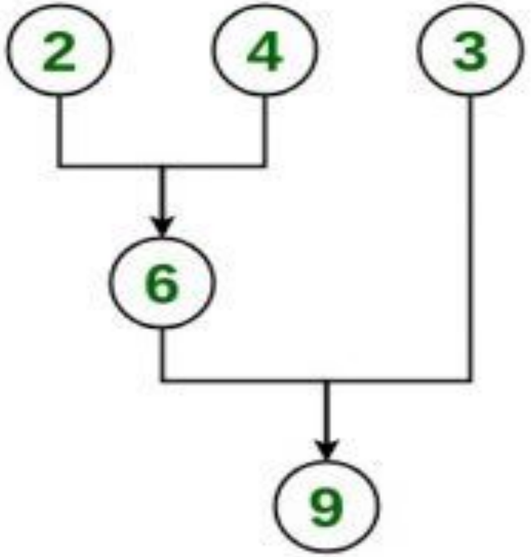
Method 2:



Complexity – Cost = 7 + 9 = 16

Example:2
Given 3 files with size 2, 3, 4 units.Find optimal way to combine these files

Method 3:



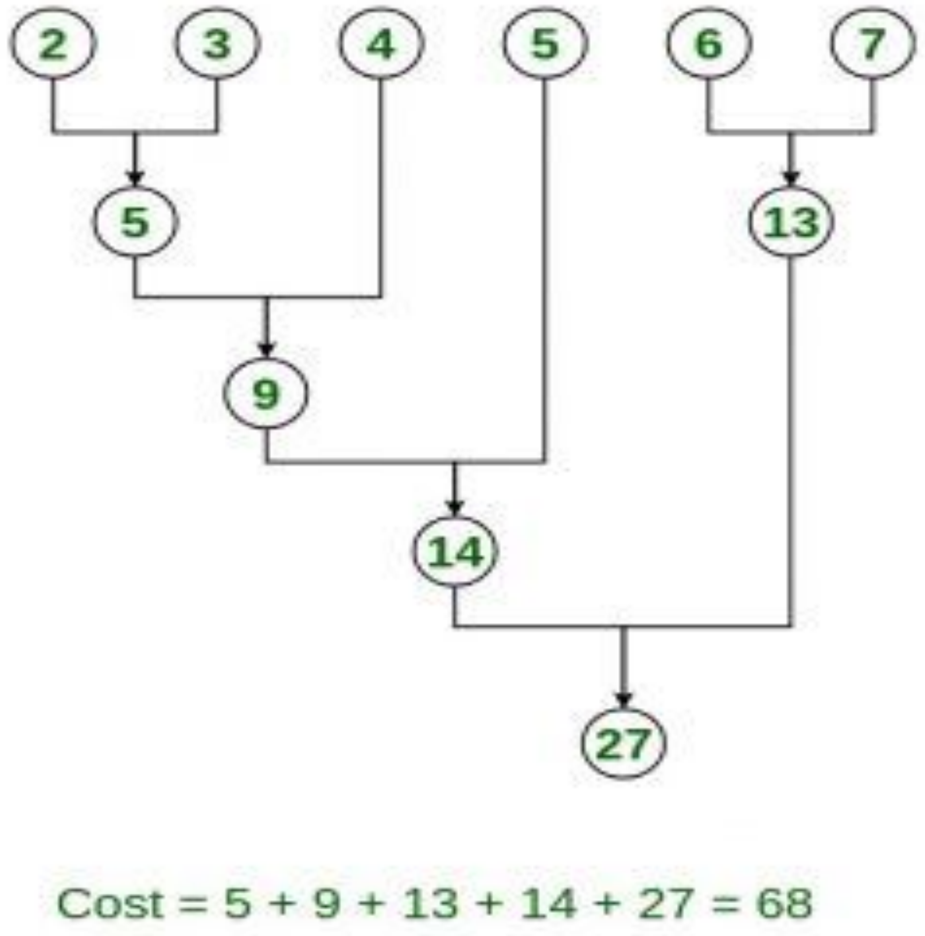
Complexity - Cost = 6 + 9 = 15

Example: 2

Input: n = 6, size = {2, 3, 4, 5, 6, 7}

Output: 68

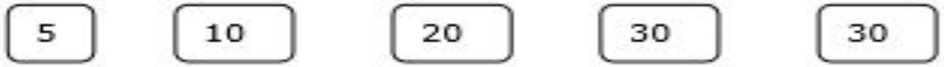
Explanation: Optimal way to combine these files



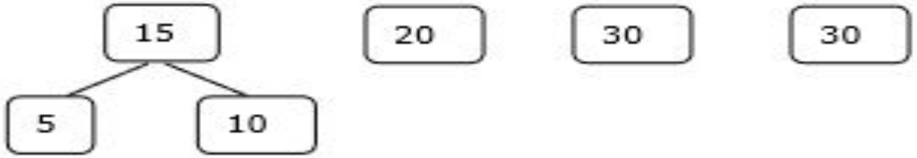
Algorithm

```
Algorithm: TREE (n)
for i := 1 to n - 1 do
   declare new node
   node.leftchild := least (list)
   node.rightchild := least (list)
   node.weight) := ((node.leftchild).weight) +
((node.rightchild).weight)
   insert (list, node);
return least (list);
```

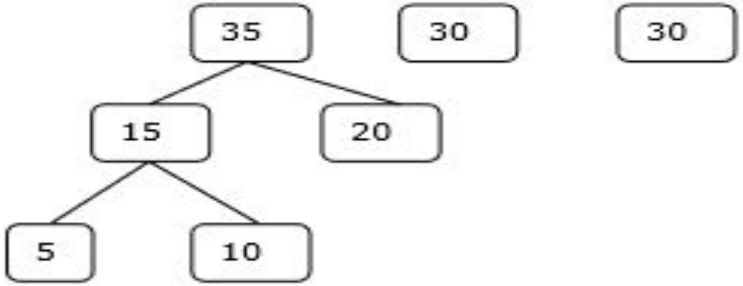
Initial Set



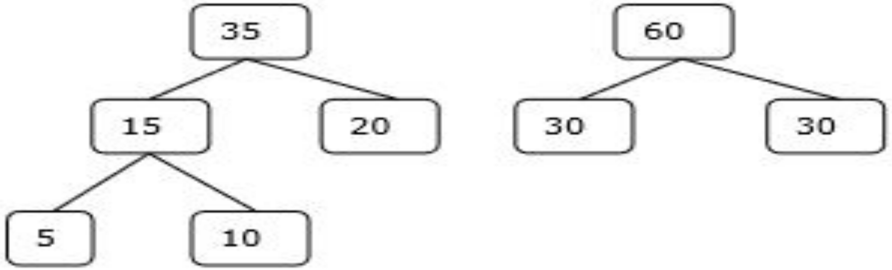
Step-1



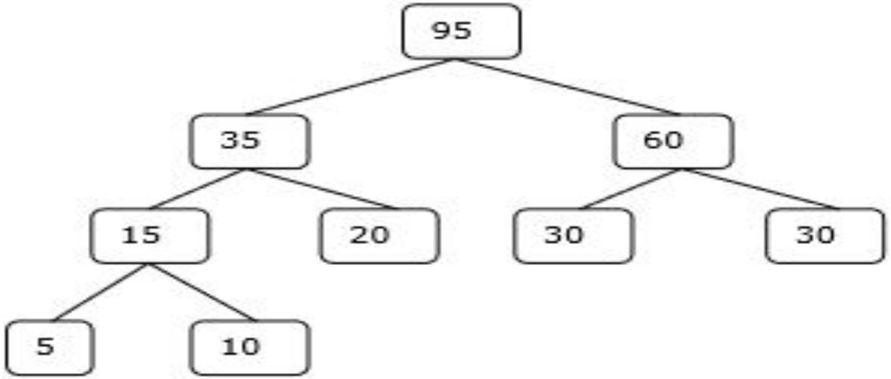
Step-2



Step-3



Step-4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Time Complexity

- If we implement **Heap** to get the minimum sized file, the time complexity is **O(nlogn)**,

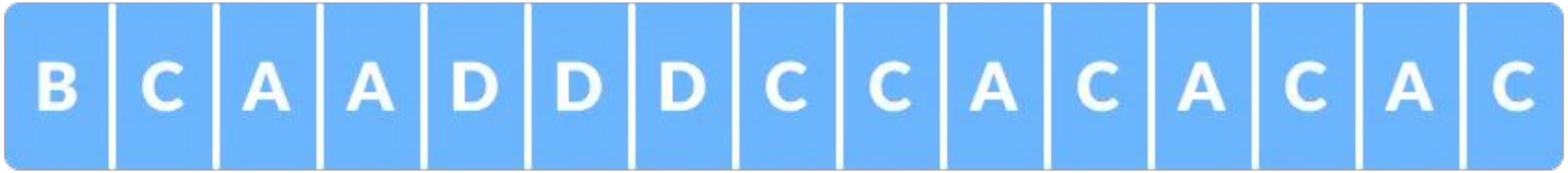
- If we use **simple list and perform linear search** to get the minimum sized file, the time complexity is **O(n^2)**.

Optimal Merge Patterns-Huffman Coding

- Huffman Coding is a technique of **compressing data to reduce its size without losing any of the details**. It was first developed by David Huffman.
- Huffman Coding is generally useful to compress the data in which there are **frequently occurring characters**.
- Huffman coding is a **lossless data compression** algorithm.
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

How Huffman Coding works?

- Suppose the string below is to be sent over a network.
- Each character occupies **8 bits**. There are a total of 15 characters in the given string.
- Thus, a total of **8 * 15 = 120 bits** are required to send this string.
- Using the Huffman Coding technique, we can compress the string to a smaller size.

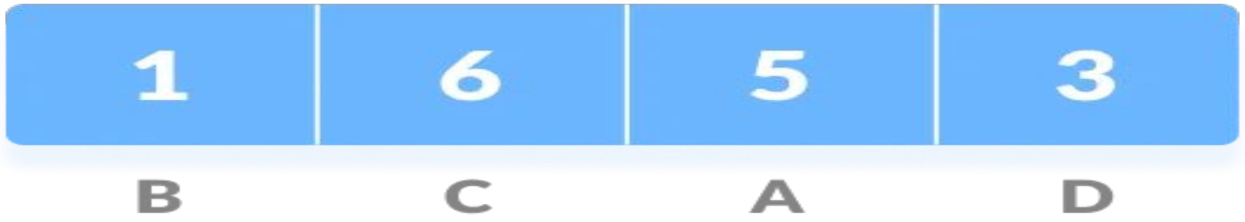


Initial string

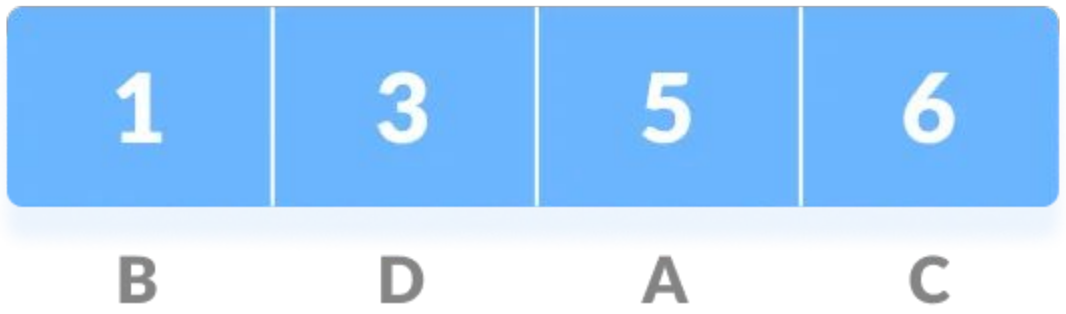
Huffman Coding **prevents any ambiguity** in the **decoding** process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The tree created helps in maintaining the property.

Huffman coding is done with the help of the following steps.

1. **Calculate the frequency of each character in the string.**

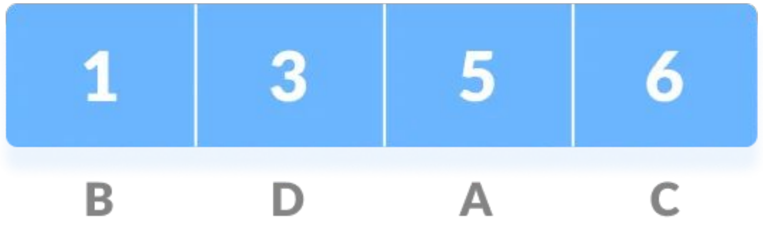


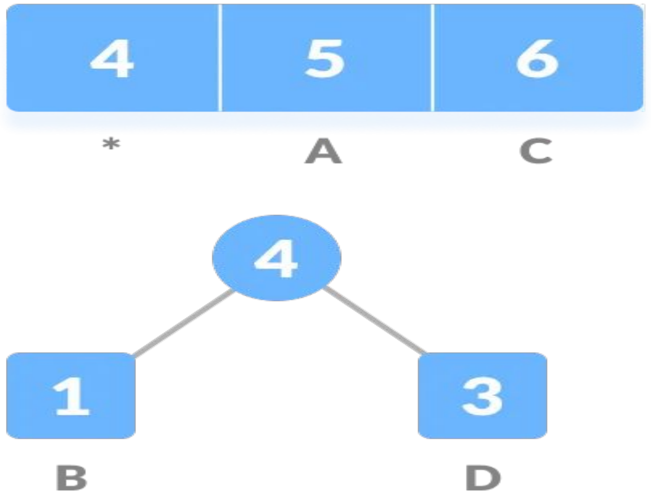
2. Sort the characters in increasing order of the frequency. These are **stored in a priority queue Q**.



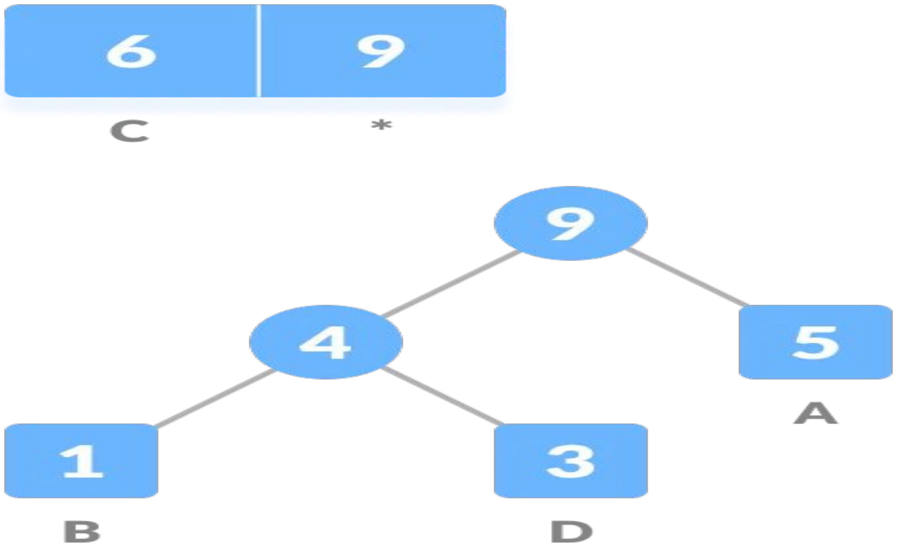
3. Make each **unique character as a leaf node**.

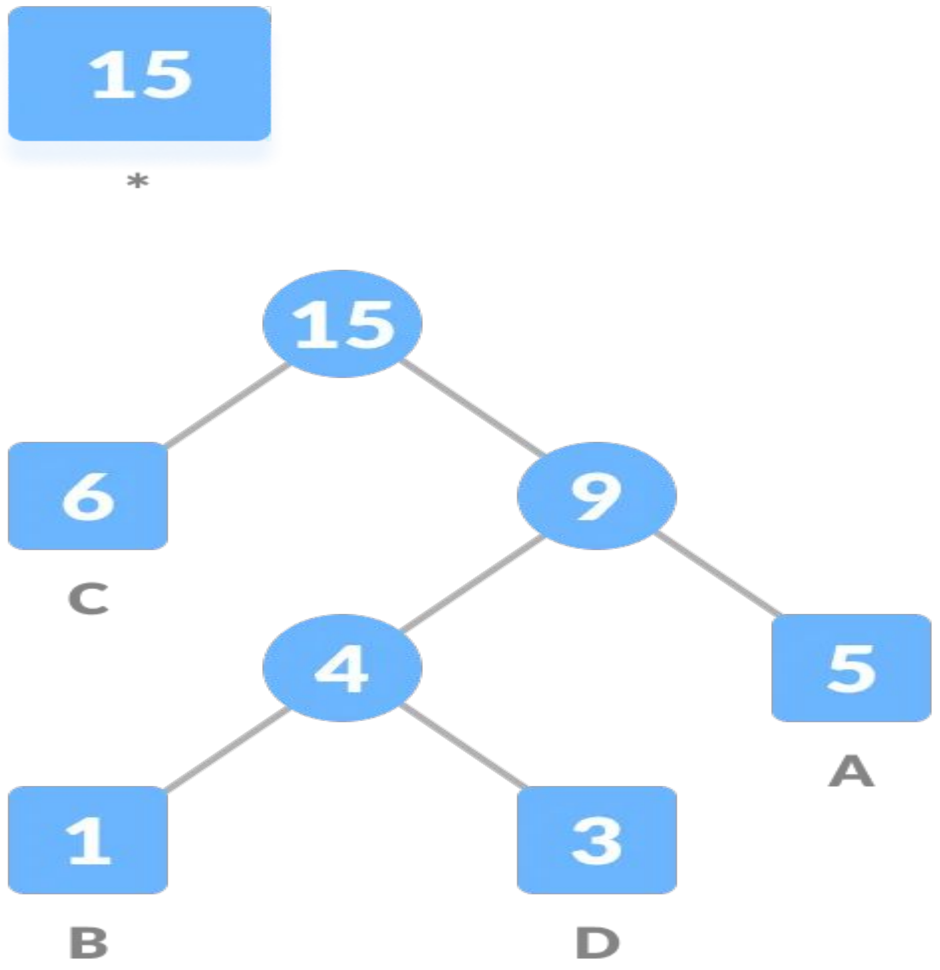
4. Create an **empty node z**. Assign the **minimum frequency to the left child of z and assign the second minimum frequency to the right child of z**. Set the value of the z as the **sum** of the above two minimum frequencies.



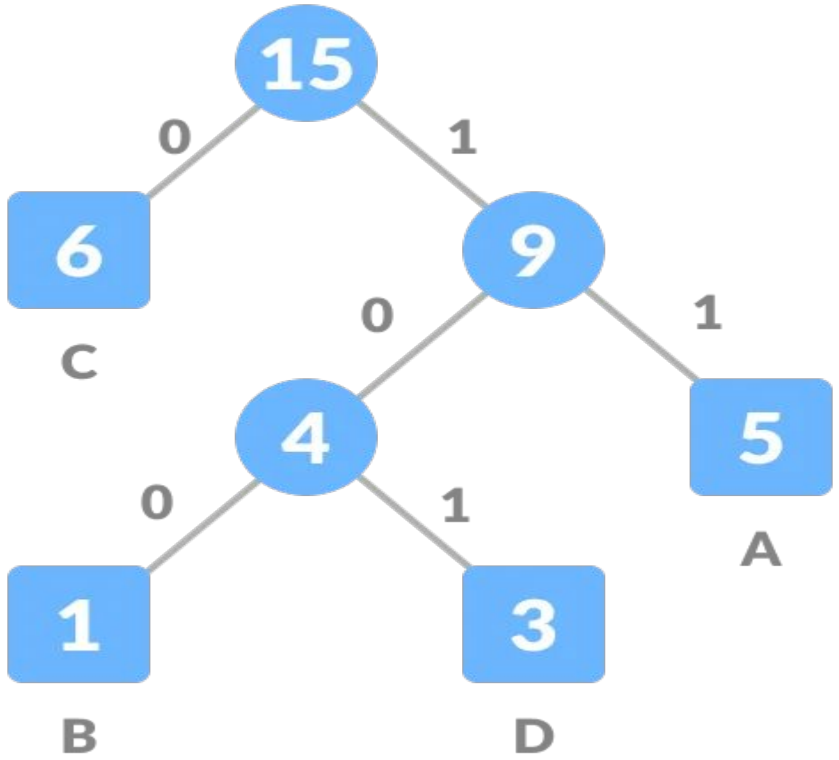


5. **Remove** these **two minimum** frequencies from Q and **add the sum** into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.

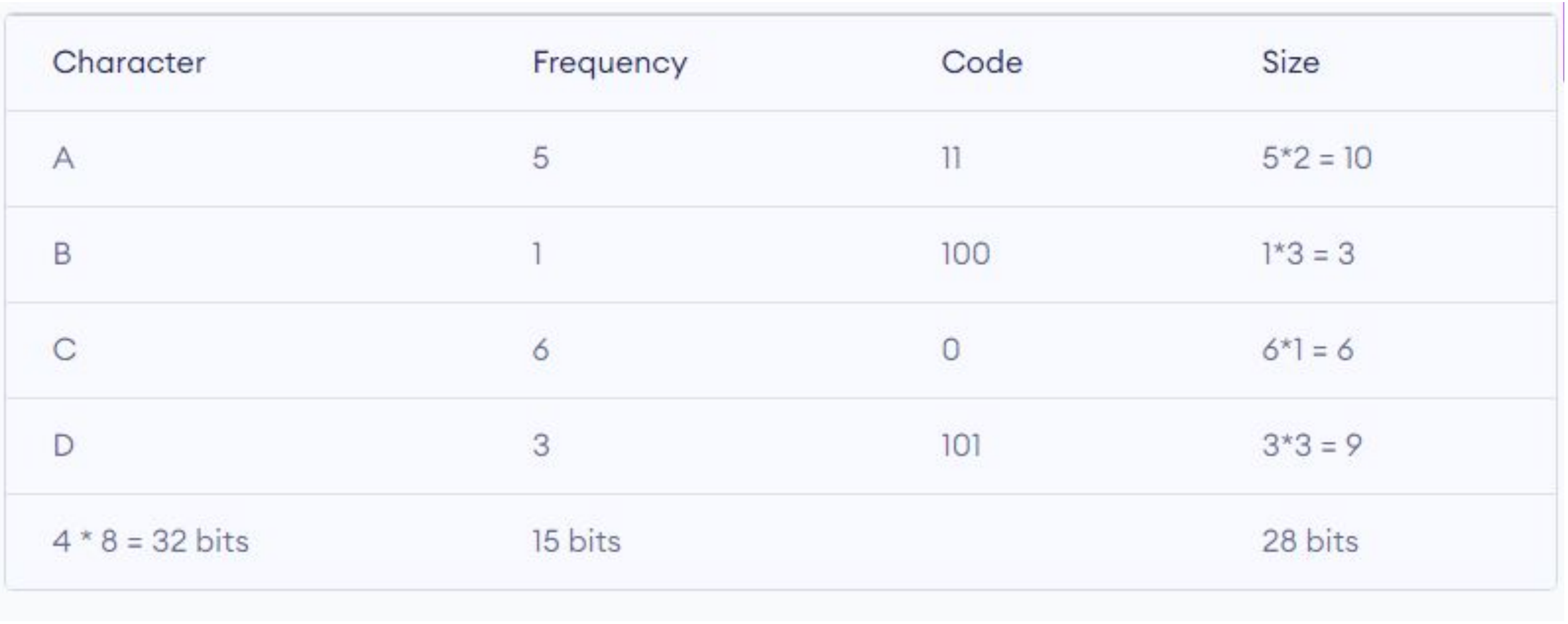



15
*
15
6
C
9
4
5
A
1
B
3
D

8. For each non-leaf node, **assign 0 to the left edge and 1 to the right edge.**



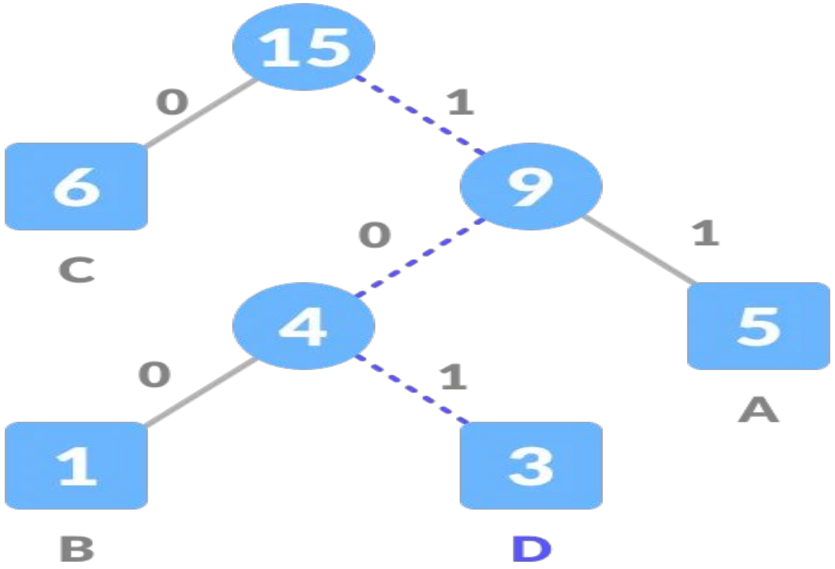
- For sending the above string over a network, we have to send the tree as well as the given compressed-code.
- **Without** encoding, the total size of the string was **120 bits.** **After** encoding the size is reduced to **32 + 15 + 28 =75 bits.**



Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Decoding the code

- For decoding the code, we can take the code and traverse through the tree to find the character.
- Let 101 is to be decoded, we can traverse from the root as in the figure below.



Huffman Coding Algorithm

```
• create a priority queue Q consisting of each unique
  character.
• sort then in ascending order of their frequencies.
• for all the unique characters:
•     create a newNode
•     extract minimum value from Q and assign it to leftChild
  of newNode
•     extract minimum value from Q and assign it to
  rightChild of newNode
•     calculate the sum of these two minimum values and
  assign it to the value of newNode
•     insert this newNode into the tree
• return rootNode
```

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the **least frequent character is at root**)
2. **Extract two nodes with the minimum** frequency from the min heap.
3. Create a new internal node with a frequency equal to the **sum of the two nodes frequencies.** Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Huffman Coding Complexity

The time complexity for encoding each unique character based on its frequency is **O(nlog n)**.

Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is **O(nlog n)**.

Huffman Coding Applications

Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.

For text and fax transmissions.

Job Sequencing With Deadlines

- The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.
- In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

- Let us consider, a set of n given **jobs** which are associated with **deadlines** and **profit** is earned, if a job is completed by its deadline.
- These jobs need to be ordered in such a way that there is **maximum profit**.
- It may happen that all of the given jobs may not be completed within their deadlines.
- Assume, **deadline of ith job Ji is di** and the **profit received from this job is pi**. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.
- Thus, **D(i)>0 for 1⩽i⩽n.**
- Initially, these jobs are ordered according to profit, i.e. **p1⩾p2⩾p3⩾...⩾pn⩾pn**.

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	**J1**	**J2**	**J3**	**J4**	**J5**
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J2	J1	J4	J3	J5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

- From this set of jobs, first we select **J2**, as it can be completed within its deadline and contributes maximum profit.
- Next, **J1** is selected as it gives more profit compared to J4.
- In the next clock, **J4** cannot be selected as its deadline is **over**, hence **J3** is selected as it executes within its deadline.
- The job **J5** is discarded as it **cannot** be executed within its deadline.
- Thus, the solution is the sequence of jobs **(J2, J1, J3)**, which are being executed within their deadline and gives maximum profit.
- **Total profit of this sequence is 100 + 60 + 20 = 180.**

Example 2

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

Answer the following questions-

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

Solution

Step-01:

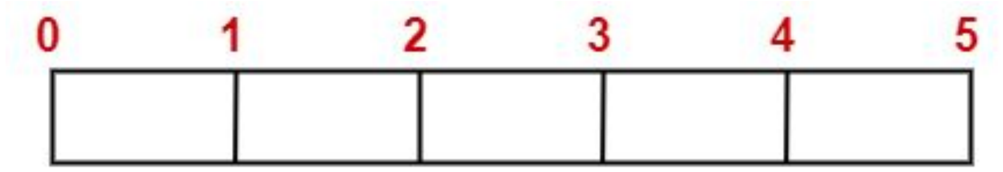
Sort all the given jobs in decreasing order of their profit-

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

Step-02:

Value of maximum deadline = 5.

So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



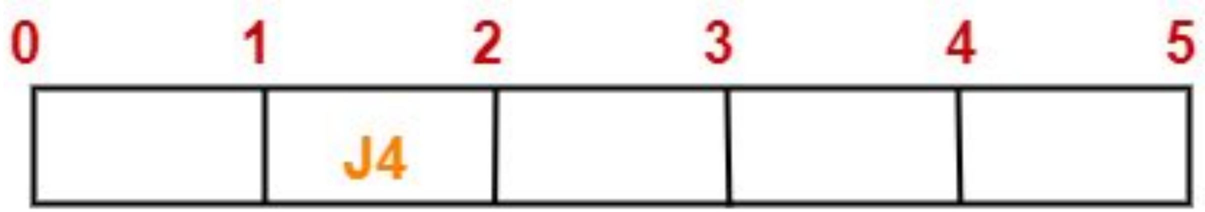
Gantt Chart

Now,

We take each job one by one in the order they appear in Step-01.

We place the job on Gantt chart as far as possible from 0.

Step-03:



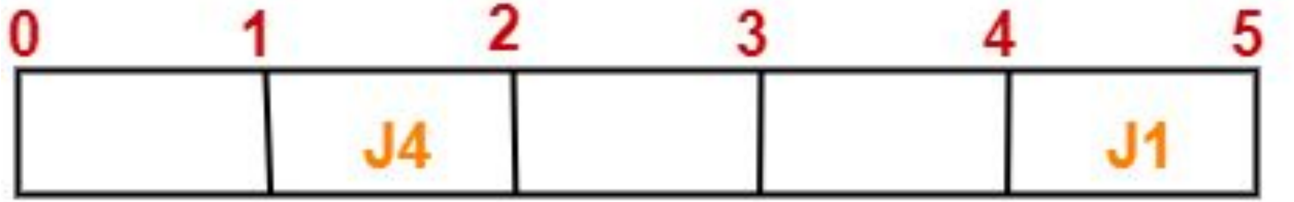
We take job J4.

Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

Step-04:

We take job J1.

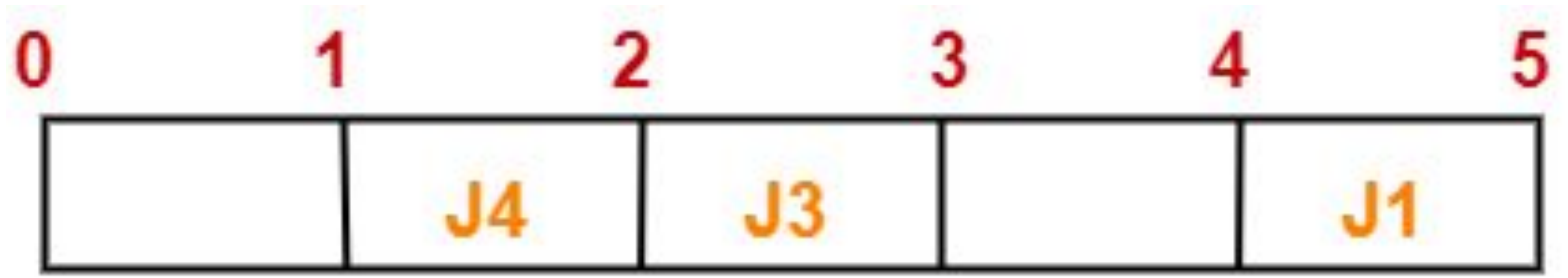
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



Step-05:

We take job J3.

Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

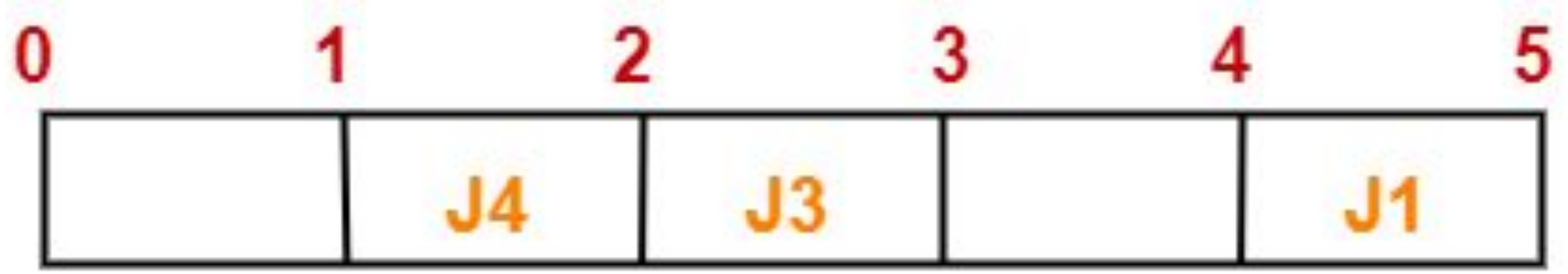


Step-06:

We take job J2.

Since its deadline is 3, so we place it in the first empty cell before deadline 3.

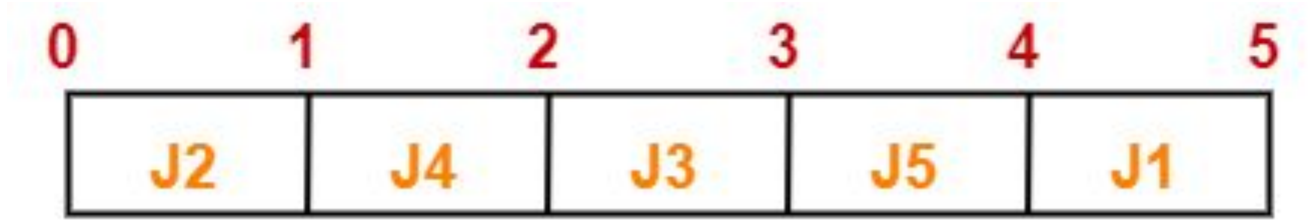
Since the second and third cells are already filled, so we place job J2 in the first cell as-



Step-07:

Now, we take job J5.

Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

The only job left is job J6 whose deadline is 2.

All the slots before deadline 2 are already occupied.

Thus, job J6 can not be completed.

1. **The optimal schedule** is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

2. **All the jobs are not completed in optimal schedule.**

This is because job J6 could not be completed within its deadline.

3. **Maximum earned profit**

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200 = **990 units**

Algorithm

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

Step-01:

Sort all the given jobs in decreasing order of their profit.

Step-02:

Check the value of maximum deadline.

Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:

Pick up the jobs one by one.

Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is **O(n^2)**.

Knapsack Problem

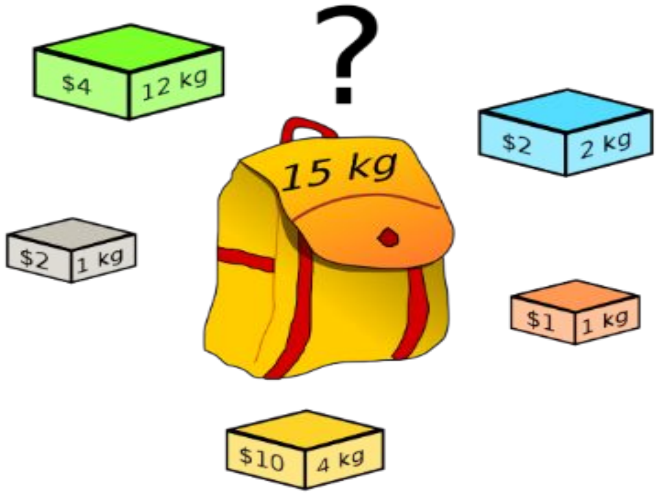
- Its is also called as **Fractional Knapsack Problem**
- You are given the following-

1. A knapsack (kind of shoulder bag) with limited **weight** capacity.
2. Few items each having some **weight and value**.

The problem states-

Which items should be placed into the knapsack such that-

1. The value or profit obtained by putting the items into the knapsack is maximum.
2. And the weight of the knapsack should not exceed



Knapsack Problem

Fractional Knapsack Problem-

In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Problem Scenario Example

- A thief is robbing a store and can carry a maximal weight of **W** into his knapsack.
- There are n items available in the store and weight of ith item is **wi** and its profit is **pi**.
- What items should the thief take?
- In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit.
- Hence, the objective of the thief is to **maximize the profit**.

Based on the nature of the items, Knapsack problems are categorized as

1. Fractional Knapsack
2. Knapsack

There are n items in the store

- Weight of ith item **wi>0**
- Profit for ith item **pi>0** and
- Capacity of the Knapsack is **W**
- In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a f**raction xi of ith item**.

0 ⩽ xi ⩽1

- The **i th** item contributes the weight **xi.wi** to the total weight in the knapsack and profit xi.pi to the total profit.
- Hence, the objective of this algorithm is to

Maximize ∑n=(xi.pi) ...(n=1 to n)

- subject to constraint,

∑n=(xi.wi)⩽W ....(n=1 to n)

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\textstyle\sum n(xi.wi)=W \quad \ldots(n=1 \text{ to } n)$$

In this context, first we need to sort those items according to the value of **pi/wi**, so that

$(pi+1)/(wi+1) \leq pi/wi$.

Here, x is an array to store the fraction of items.

Fractional Knapsack Problem Using Greedy Method-

Step-01:

For each item, compute its **profit/ weight ratio**.

Step-02:

Arrange all the items in **decreasing order of their profit/ weight ratio**.

Step-03:

Start putting the items into the knapsack beginning from the item with the **highest ratio**.

Put as many items as you can into the knapsack.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

Example

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

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

Solution-

Step-01:

Compute the profit/ weight ratio for each item-

Items	Profit	Weight	Ratio
1	30	5	6
2	40	10	4
3	45	15	3
4	77	22	3.5
5	90	25	3.6

Step-02:

Sort all the items in decreasing order of their **Profit / weight** ratio-

I1	I2	I5	I4	I3
(6)	(4)	(3.6)	(3.5)	(3)

Step-03:

Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	Items in Knapsack	Cost
60	Ø	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.

Since in fractional knapsack problem, even the fraction of any item can be taken.

So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

Total cost of the knapsack

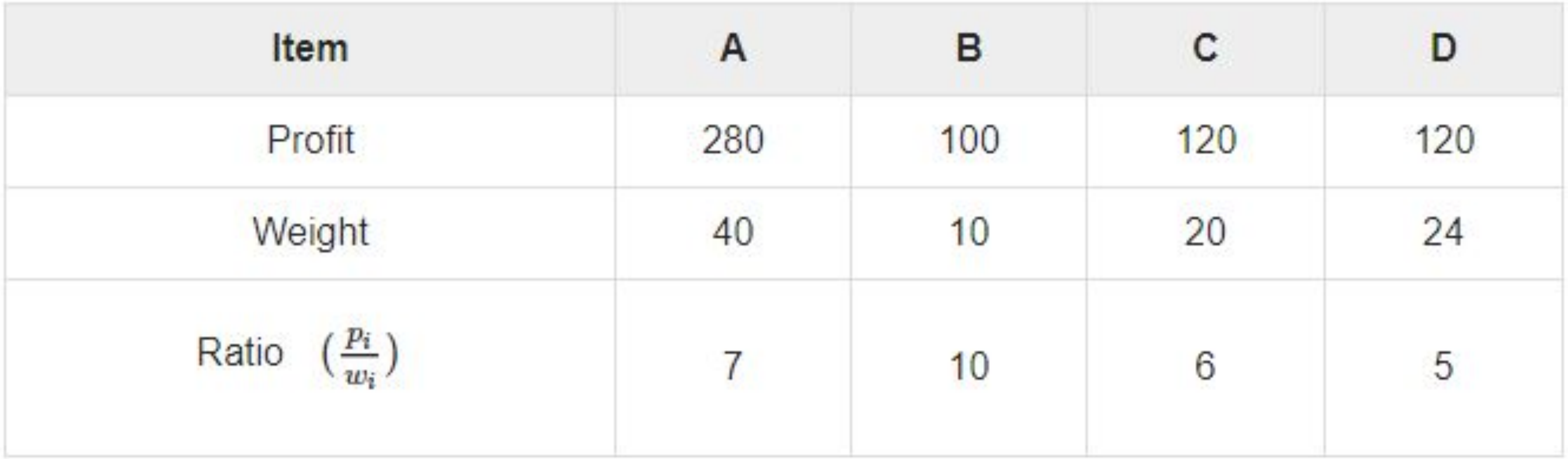
= 160 + (20/22) x 77

= 160 + 70

= 230 units

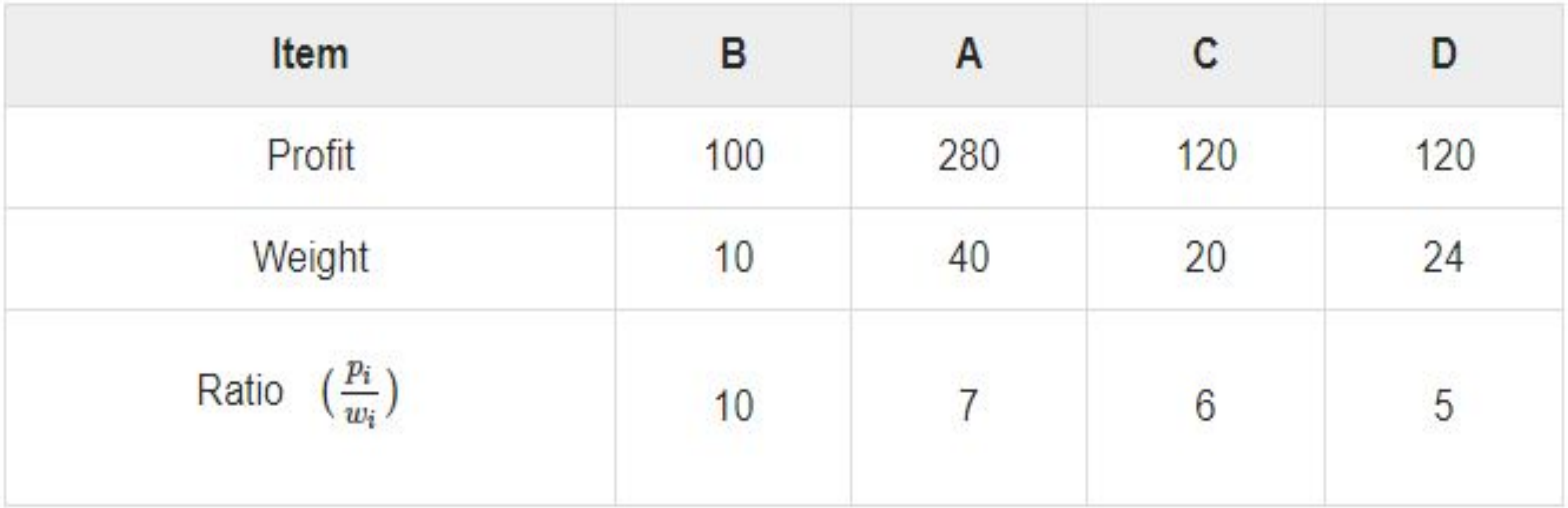
Example

Let us consider that the capacity of the knapsack **W = 60** and the list of provided items are shown in the following table –



Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(\frac{p_i}{w_i})$	7	10	6	5

As the provided items are not sorted based on **piwi.** After sorting, the items are as shown in the following table.



Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(\frac{p_i}{w_i})$	10	7	6	5

Solution

- After sorting all the items according to **piwi**.
- First all of **B** is chosen as weight of B is less than the capacity of the knapsack.
- Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of A.
- Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of C.
- Hence, **fraction of C (i.e. (60 − 50)/20)** is chosen.
- Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
- The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**
- And the **total profit is**

  100 + 280 + 120 * (10/20) = 380 + 60 = 440

Analysis

- If the provided items are already sorted into a decreasing order of **pi/wi**, then the while loop takes a time in **O(n)**;
- Therefore, the total time including the sort is in

  O(n logn).

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

1. Finding the least wasteful way to cut raw materials
2. portfolio optimization
3. Cutting stock problems

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?

Some points to note here:

It might not be possible to complete all the activities, since their **timings can collapse**.

1. **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.**
2. Greedy approach can be used to find the solution since we want to **maximize the count of activities** that can be executed.
3. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

1. **act[] array** containing all the activities.
2. **s[]** array containing the **starting time** of all the activities.
3. **f[]** array containing the **finishing time** of all the activities.

Output Data from the Algorithm:

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

Algorithm

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.

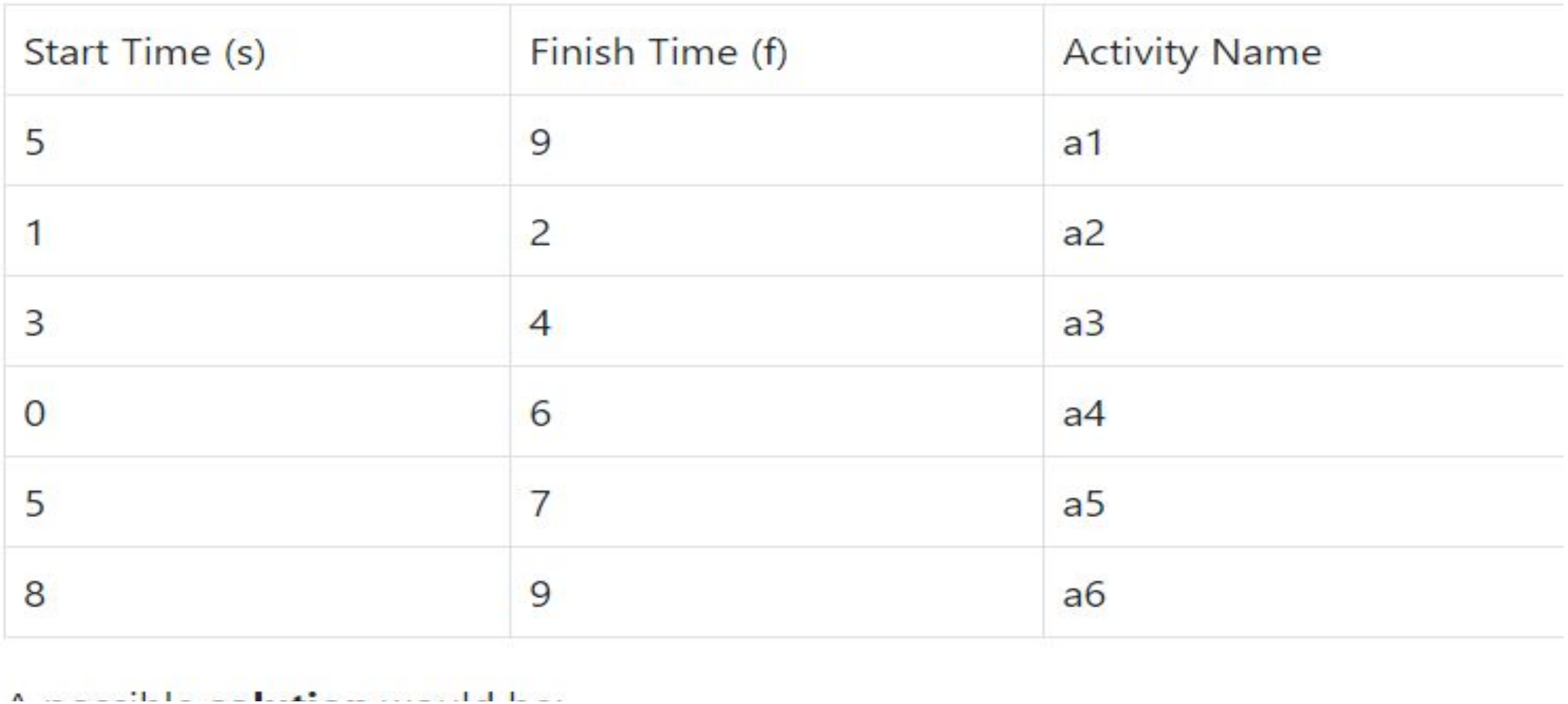
Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

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.

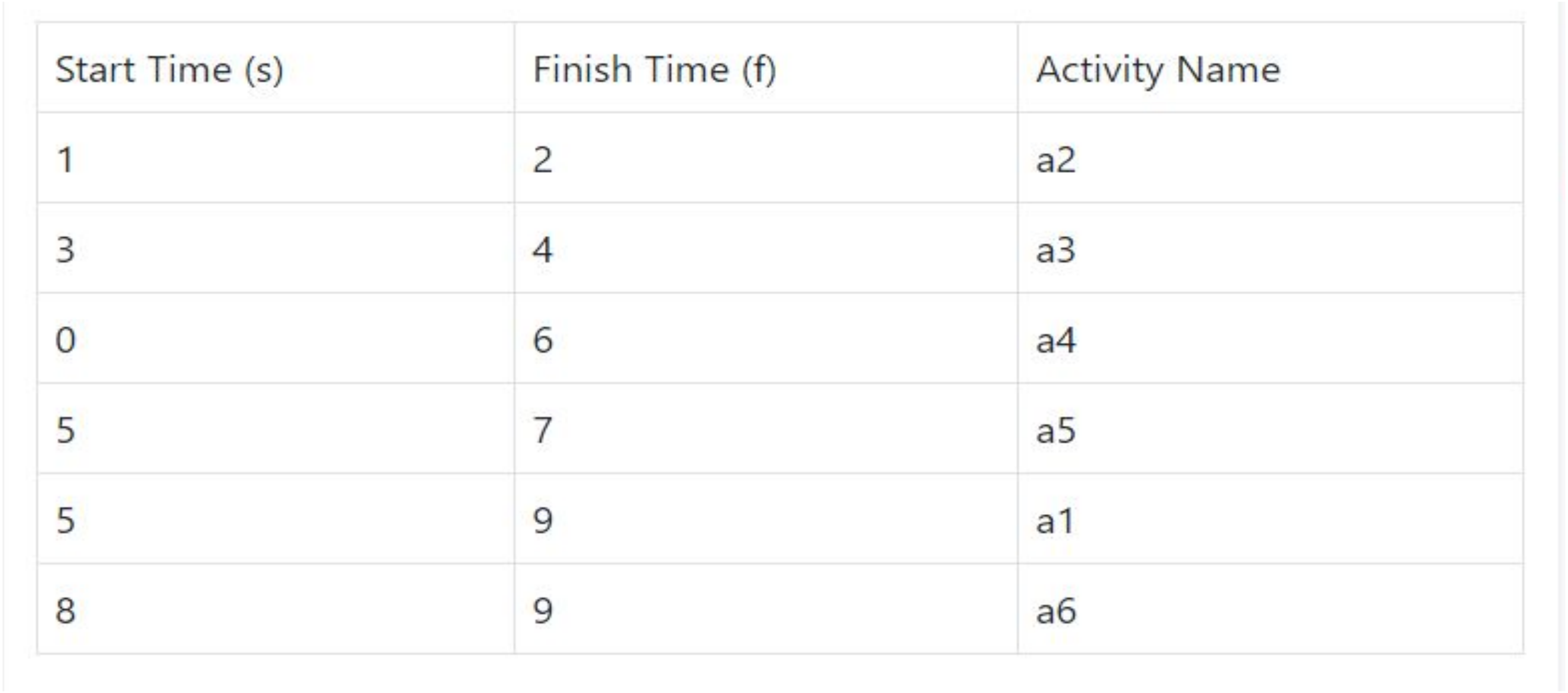
Activity Selection Problem Example



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**.



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[]

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:

OUTPUT :

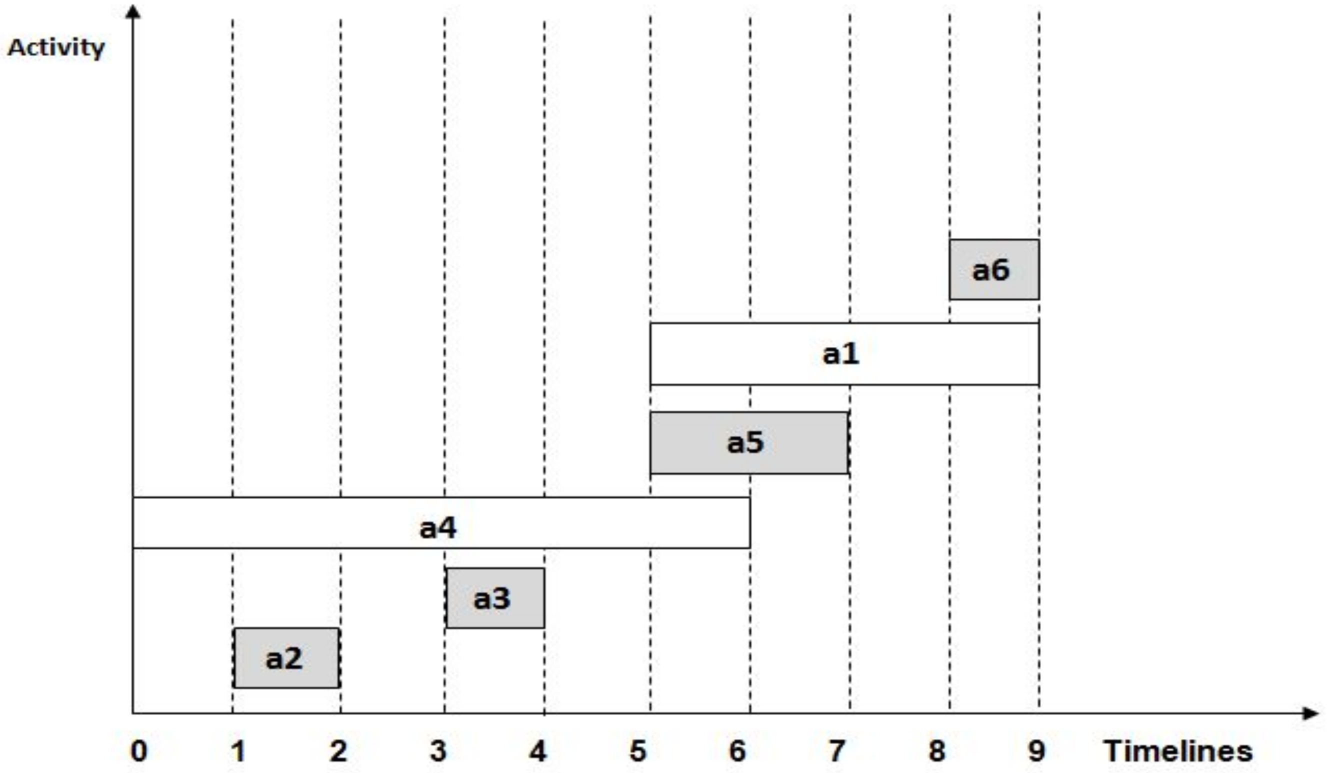
```
(1,2)

(3,4)

(5,7)

(8,9)
```

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



Time Complexity Analysis

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** . The time complexity of this method will be O(nlogn), which also defines complexity of the algorithm.

Applications

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

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

Thank You