# MODULE-2

## MERGE SORT

Merge Sort is a Divide & conquer algorithm. It divides the input array into 2 halves, call itself for the 2 halves & then merges the 2 sorted halves. The merge () function is used for merging 2 halves: a[1] .... a[⌊n/2⌋] and a[⌊n/2+1⌋] .... a[n]. Each set is individually sorted & the resulting sorted sequences are merged to produce and single sorted sequence of n elements.

Algorithm:

Step 1 : START

Step 2 : Algorithm MergeSort(arr[], l, r)

Step 3 : If r > l

Step 4 : find the middle point to divide the array into 2 halves.

Middle, m = (l+r)/2

Step 5 : Call Merge Sort for first half :

Merge sort (arr, l, m)

Step 6 : Call Merge Sort for second half :

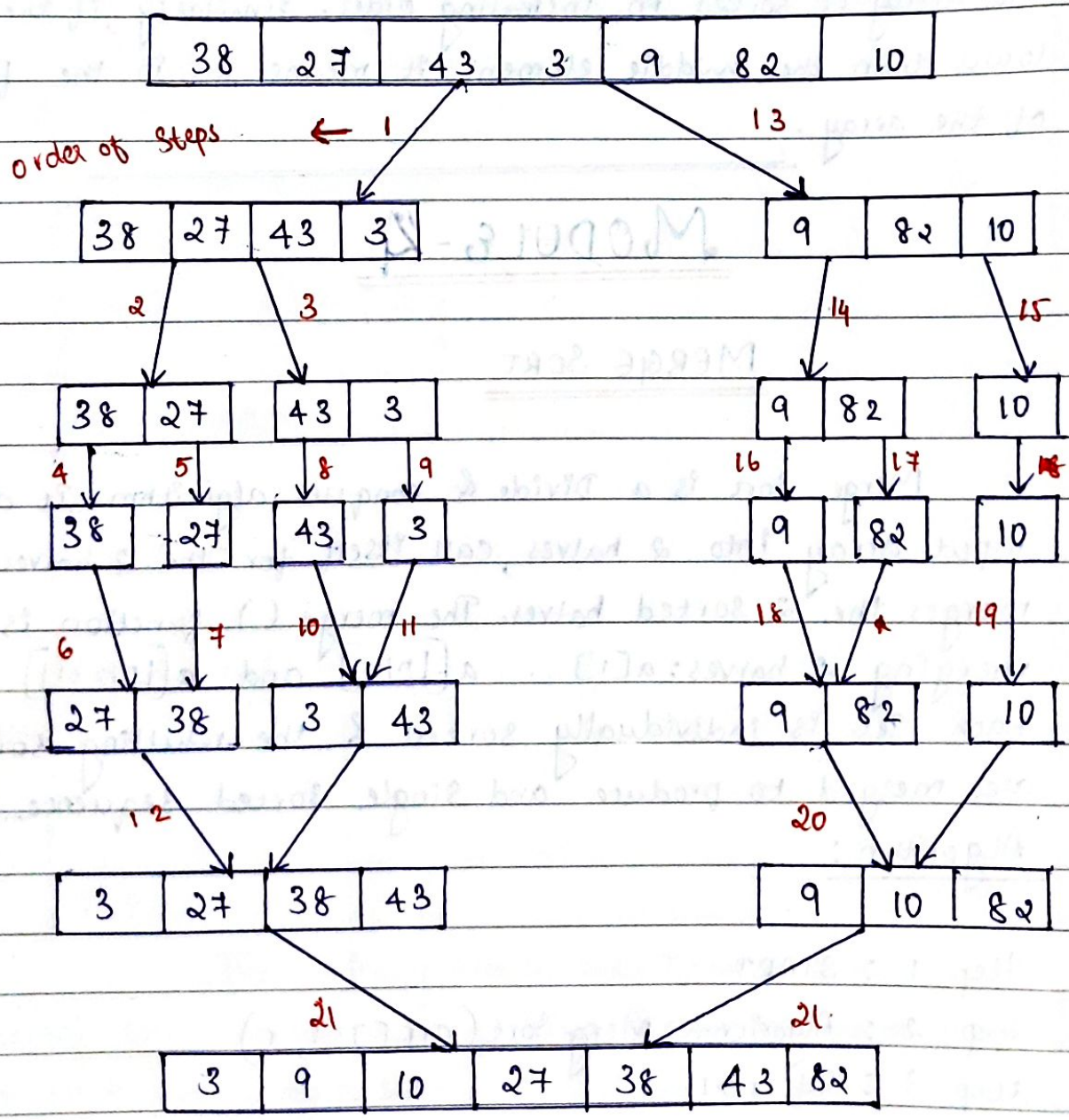
Mergesort (arr, m+1, r)

Step 7 : Merge the 2 halves sorted in step 2 and step 3 :

Merge (arr, l, m, r)

Step 8 : STOP

Example : Array : 38, 27, 43, 3, 9, 82, 10.



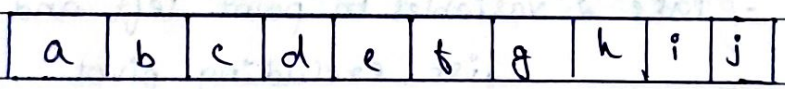

Time complexity of Merge sort - $\theta(n \log n)$

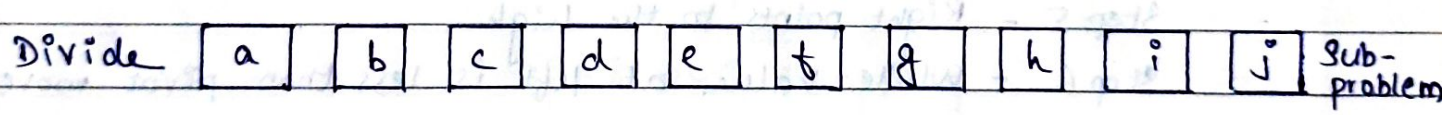
# Divide & Conquer Method

This technique can be divided into the following 3 parts.

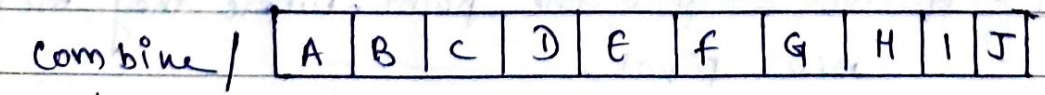
1) **Divide** : This involves dividing the problem into some sub problem.
2) **Conquer** : sub problem by calling recursively until sub problem solved.
3) **Combine** : The sub problem solved so that we will get find problem solution.

In this approach, the problem in hand, is divided into smaller sub problems & then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.







# Quick Sort

* Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays, one of which holds values smaller than the specified value, say pivot based on which the partition is made and another array holds values greater than the pivot value.

* Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

* This algorithm is quite efficient for large-sized data sets.

Algorithm

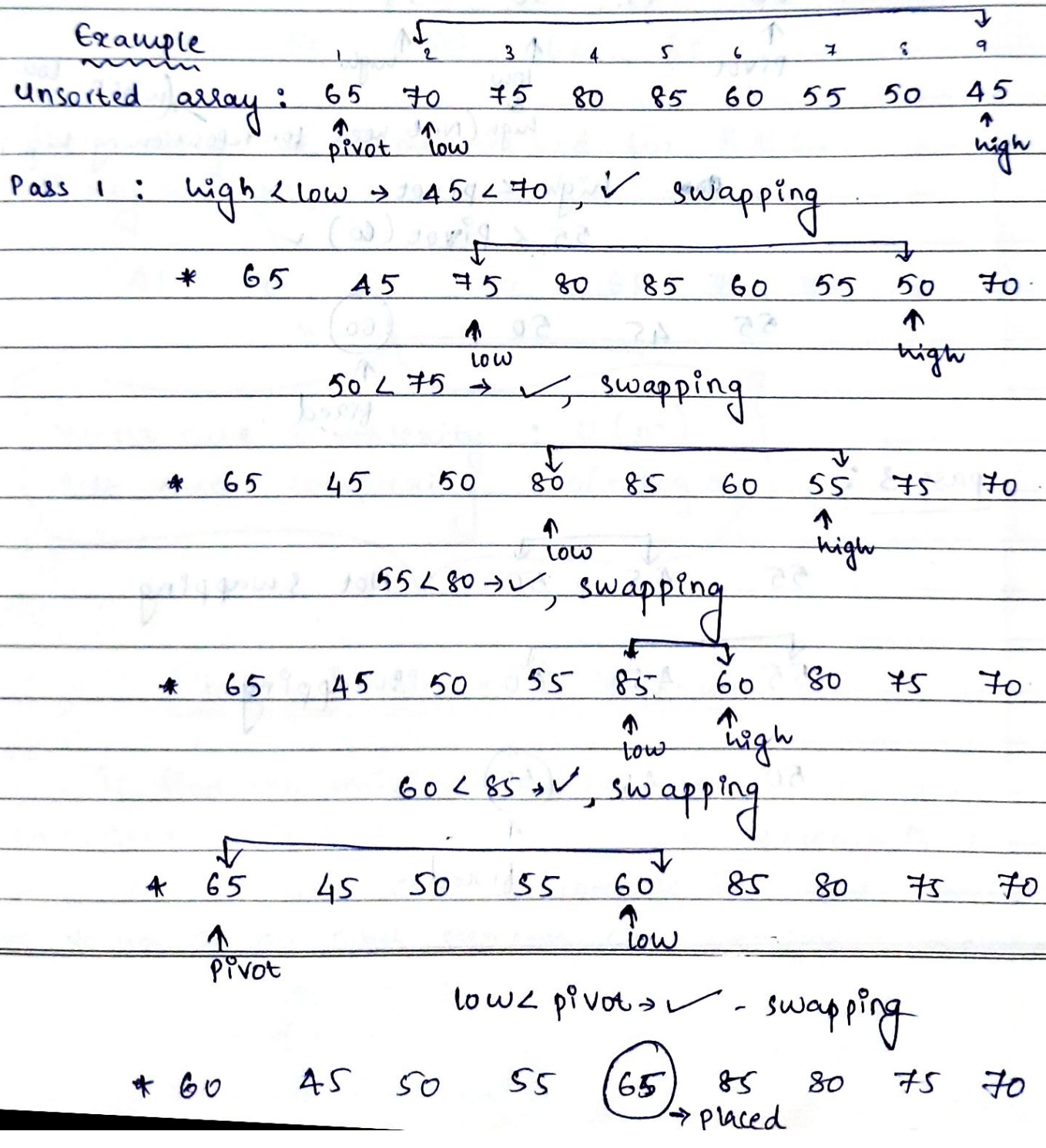
1) Quick sort pivot algorithm

Step 1 - START
Step 2 - Choose the highest index value has pivot
Step 3 - Take 2 variables to point left and right of the list excluding pivot.
Step 4 - Left points to the low index
Step 5 - Right points to the high
Step 6 - While value at left is less than pivot move right
Step 7 - While value at right is greater than pivot move left
Step 8 - If both step 6 & step 7 doesnot match swap left and right
Step 9 - If left ≥ right, the point where they met is new pivot.
Step 10 - STOP

## 2) Quick Sort Algorithm

Step 1 - START
Step 2 - Make the right most index value / left most or mid point set as pivot element
Step 3 - Partition the array using pivot value
Step 4 - Quick sort left partition recursively
Step 5 - Quick sort right partition recursively
Step 6 - STOP

Example



Pass 2 : Consider the left array.

60 45 50 55

↑ Pivot ↑ low ↑ high

55 < 45 → condition false, No swapping

Repeat the same step

60 45 50 55

↑ Pivot ↑ low ↑ high.

low & high (Not need for repositioning high). → (when low = high)

high < pivot

55 < pivot (60) ✓

55 45 50 (60)

↑ fixed.

pass 3 :

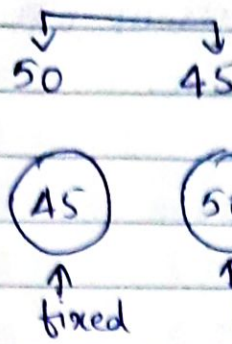
55 45 50 Not swapping.

55 45 50 swapping.

50 45 (55)

↑ fixed.

Pass 4




After first half, Resultant array →

45 50 55 60 65

Then apply the same method for R.H.S.
Finally we got,

45 50 55 60 65 70 75 80 85

Worst case complexity : $O(n^2)$
Best case complexity : $o(n \log n)$

## Finding Maximum & Minimum

To find the minimum value into an array of items isn't difficult. The most natural approach is to take the first item and to compare its value against the values of all other elements. Once we find a smaller

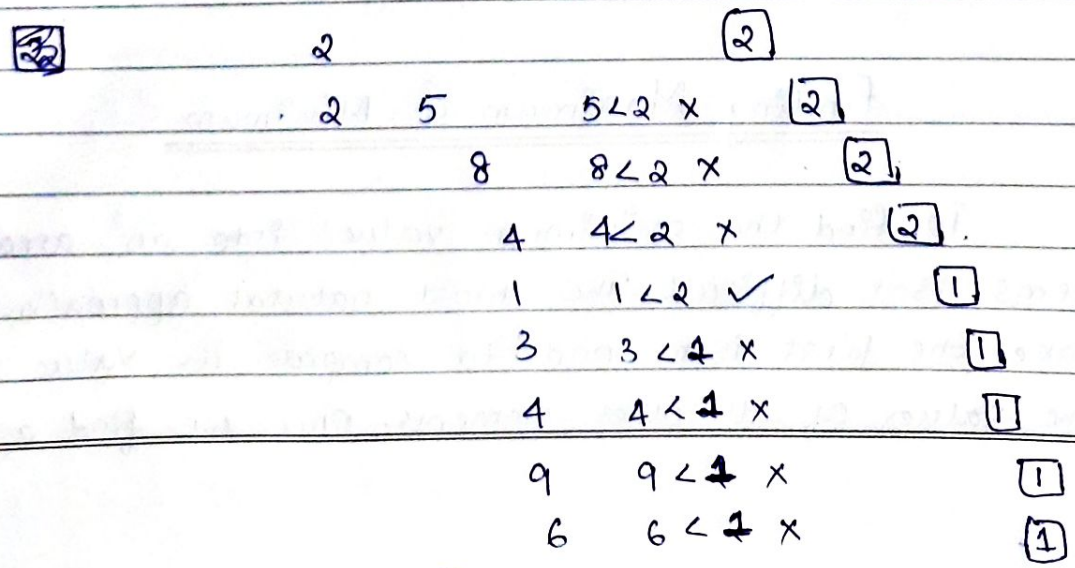
element, we continue the comparisons with its value. finally, we find the minimum.

## Algorithm

Step 1 : START
Step 2 : Algorithm Max-Min (numbers [])
Step 3 : max = numbers [1]
Step 4 : min = numbers [1]
Step 5 : for i = 2 to n do
Step 6 : if numbers [i] > max then
Step 7 : Max = numbers [i]
Step 8 : if numbers [i] < min then
Step 9 : Min = numbers [i]
Step 10 : Return (Max, Min)
Step 11 : STOP

## Example

Minimum: 2, 5, 8, 4, 1, 3, 4, 9, 6



Maximum: 2, 5, 8, 4, 1, 3, 4, 9, 6

| | | |
|---|---|---|
| 2 | | [2] |
| 5 | 5 > 2 ✓ | [5] |
| 8 | 8 > 5 ✓ | [8] |
| 4 | 4 > 8 x | [8] |
| 1 | 1 > 8 x | [8] |
| 3 | 3 > 8 x | [8] |
| 4 | 4 > 8 x | [8] |
| 9 | 9 > 8 ✓ | [9] |
| 6 | 6 > 9 x | [9] |

Return - (9) → Maximum Element.

Complexity : O(n)

## Strassen's Matrix Multiplication

Let us consider two matrices 'X' & 'Y'. We want to calculate the resultant matrix 'Z' by multiplying X & Y.

In Strassen's matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's matrix multiplication can be performed only on square matrices where 'n' is a power of 2. Order of both of the matrices are n x n.

Consider 2 matrices

$$A = \begin{bmatrix} A_{11} & A_{12} \\ 1 & 2 \\ 3 & 4 \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ 5 & 6 \\ 7 & 8 \\ B_{21} & B_{22} \end{bmatrix}$$

Normal matrix multiplication ⇒

$$\text{Resultant matrix, } C = \begin{bmatrix} 1\times5+2\times7 & 1\times6+2\times8 \\ 3\times5+4\times7 & 3\times6+4\times8 \end{bmatrix}$$

$$= \begin{bmatrix} 5+14 & 6+16 \\ 15+28 & 18+32 \end{bmatrix}$$

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Using Strassen's matrix multiplication ⇒

$S_1 = B_{12} - B_{22} = 6 - 8 = -2$

$S_2 = A_{11} + A_{12} = 1 + 2 = 3$

$S_3 = A_{21} + A_{22} = 3 + 4 = 7$

$S_4 = B_{21} - B_{11} = 7 - 5 = 2$

$S_5 = A_{11} + A_{22} = 1 + 4 = 5$

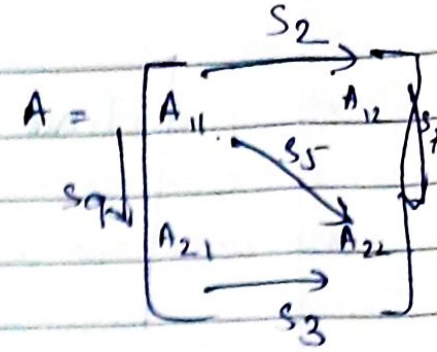
$S_6 = B_{11} + B_{22} = 5 + 8 = \underline{13}$

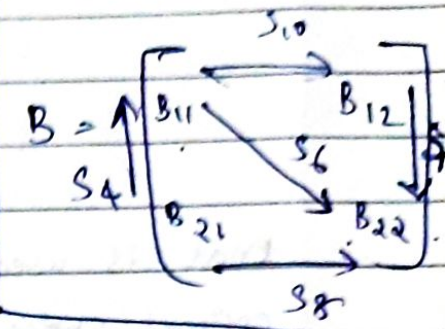
$S_7 = A_{12} - A_{22} = 2 - 4 = \underline{-2}$

$S_8 = B_{21} + B_{22} = 7 + 8 = \underline{15}$

$S_9 = A_{11} - A_{21} = 1 - 3 = \underline{-2}$

$S_{10} = B_{11} + B_{12} = 5 + 6 = \underline{11}$




$P_1 = A_{11} \times S_1 = 1 \times -2 = \underline{-2}$

$P_2 = B_{22} \times S_2 = 8 \times 3 = \underline{24}$

$P_3 = B_{11} \times S_3 = 5 \times 7 = \underline{35}$

$P_4 = A_{22} \times S_4 = 4 \times 2 = \underline{8}$

$P_5 = S_5 \times S_6 = 5 \times 13 = \underline{65}$

$P_6 = S_7 \times S_8 = -2 \times 15 = \underline{-30}$

$P_7 = S_9 \times S_{10} = -2 \times 11 = \underline{-22}$

$C_{11} \Rightarrow P_5 + P_4 - P_2 + P_6$

$C_{22} = P_5 + P_1 - P_3 - P_7$

Resultant Matrix, $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

$$\begin{cases} C_{11} = P_5 + P_4 - P_2 + P_6 = 65 + 8 - 24 + (-30) = \underline{19} \\ C_{12} = P_1 + P_2 = -2 + 24 = \underline{22} \\ C_{21} = P_3 + P_4 = 35 + 8 = \underline{43} \\ C_{22} = P_5 + P_1 - P_3 - P_7 = 65 + (-2) - 35 - (-22) = \underline{50} \end{cases}$$

Time complexity $\Rightarrow O(n^{\log 7})$, which is approximately $O(n^{2.8074})$

## Greedy Method

A Greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious & immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

## Knapsack Problem

* The knapsack problem is a problem in combinatorial optimization (finding an optimal object from a finite set of objects).
* Given a set of items, each with a weight and a value, determine the no: of each item to include in a collection so that the total weight is less than or equal

to a given limit & the total value is as large as possible.

* Knapsack is a sack which we need to place the given items according to the capacity of the knapsack.

## Algorithm

Step 1 : START
Step 2 : Bound (CP, CW, K)
Step 3 : {
Step 4 : b = CP, c = CW
Step 5 : for i = K+1 to n
Step 6 : {
Step 7 : c = c + w[i]
Step 8 : if (c < m) then
Step 9 : b = b + p[i]
Step 10 : else
Step 11 : Return $b + \left(1 - \left(\left(c - m\right) / w[i]\right)\right) * p[i]$
Step 12 : }
Step 13 : }
Step 14 : STOP

where, CP → current profit
CW → current weight
K → index
m → maximum capacity

Complexity → $O(n \log n)$

Example:

Profits → $P_1 = 10$, $P_2 = 10$, $P_3 = 12$, $P_4 = 18$.

Weights → $w_1 = 2$, $w_2 = 4$, $w_3 = 6$, $w_4 = 9$

$M = 15$ → capacity

$n = 4$

When $i = K+1 \rightarrow 0+1 = 1$

$c = c + w[i] \Rightarrow 0+2 = 2$

$2 < 15$ ✓

$b = b + P[i] = 0+10 = 10$

When $i = 2$, $c = 2 + w[2] \Rightarrow 2+4 = 6$

$6 < 15$ ✓

$b = b + P[i] = 10+10 = 20$

When $i = 3$, $c = 6+6 = 12$

$12 < 15$ ✓

$b = 20+12 = 32$

When $i = 4$, $c = 12+9 = 21$.

$21 < 15$ ✗

$b = b + (1 - ((c-m)/w[i])) * P[i]$

$= 32 + (1 - (21-15)/9) * 18$

$= 32 + \left(1 - \frac{6}{9}\right) * 18$

$= 32 + \left(\frac{9-6}{9}\right) * 18$

$= 38$ → Max profit

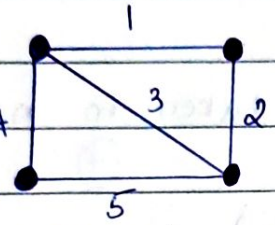
To obtain this maximum profit, we need to consider the item 4 first, then second, then first.

$18 + 10 + 10 = 38$ $(P_4 + P_2 + P_1)$.

# Minimum Cost Spanning Trees

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees.

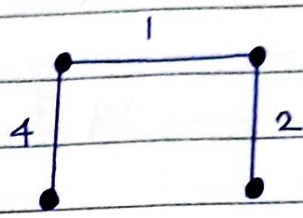
eg:




Undirected graph


Spanning Tree

Cost = 11 (4+5+2)




Minimum Spanning Tree (4+1+2) = 7

There are 2 famous algorithms for finding the Minimum Spanning Tree.

1) Kruskal's algorithm
2) Prim's algorithm

These 2 algorithms are Greedy algorithm. Each step of a greedy algorithm must make one of several possible choices. Making the choice that is the best at the moment.

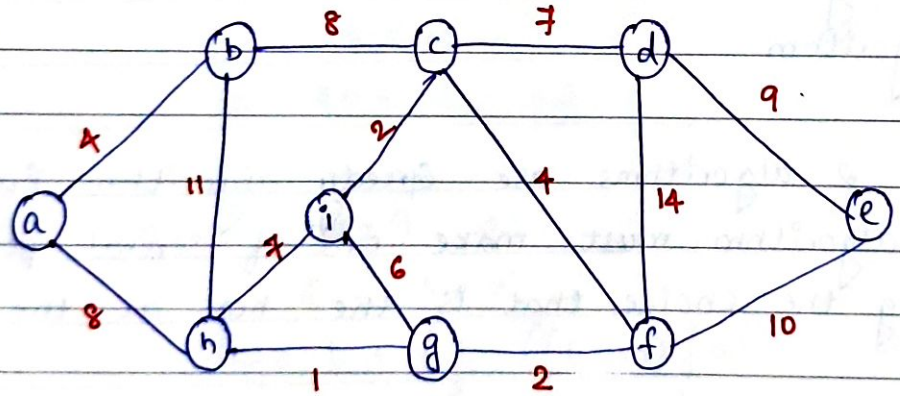
## Kruskal's Algorithm

This algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm.

## Algorithm

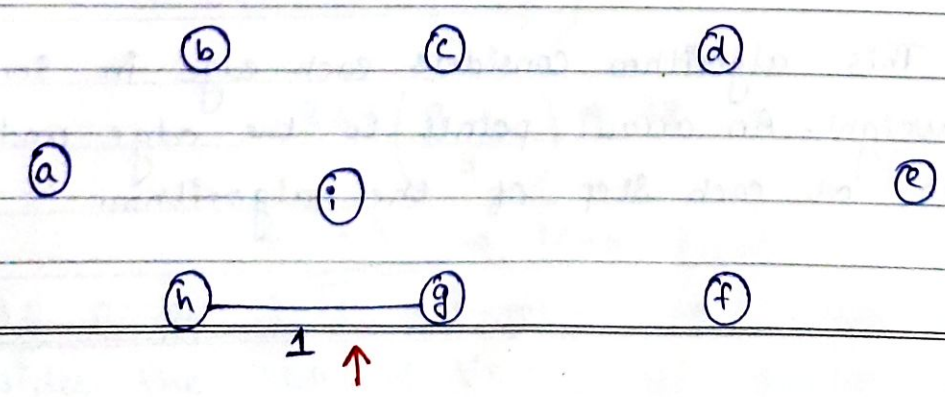
Step 1 : START
Step 2 : MST - KRUSKAL (G, w)
Step 3 : A = $\phi$
Step 4 : for each vertex $v \in G.V$
Step 5 : Make - set (v)
Step 6 : Sort the edges of G.E into non decreasing order by weight w.
Step 7 : for each edge $(u, v) \in G.E$ taken in non-decreasing order by weight.
Step 8 : If find-set (u) $\neq$ find-set (v)
Step 9 : $A = A \cup \{(u, v)\}$
Step 10 : Union (u, v)
Step 11 : Return A
Step 12 : STOP

Example.



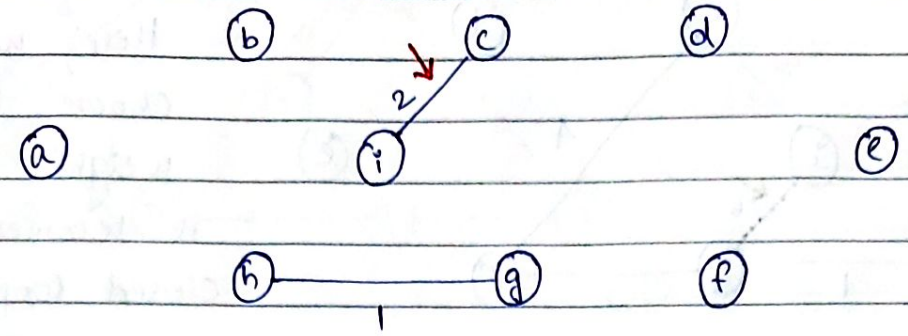

Solution

a)



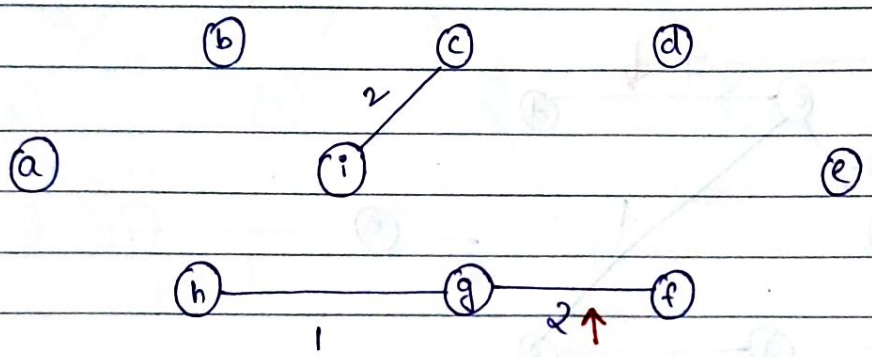

Select the smallest weight.

b)

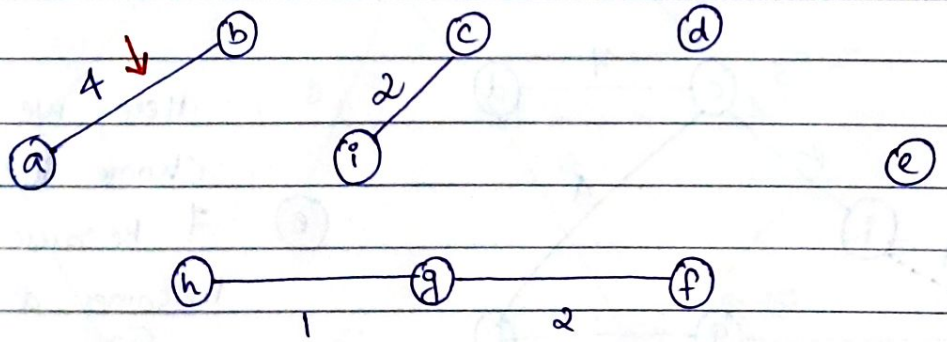



Select the next weight (sorted order) as your choice. i↔c or g↔f.

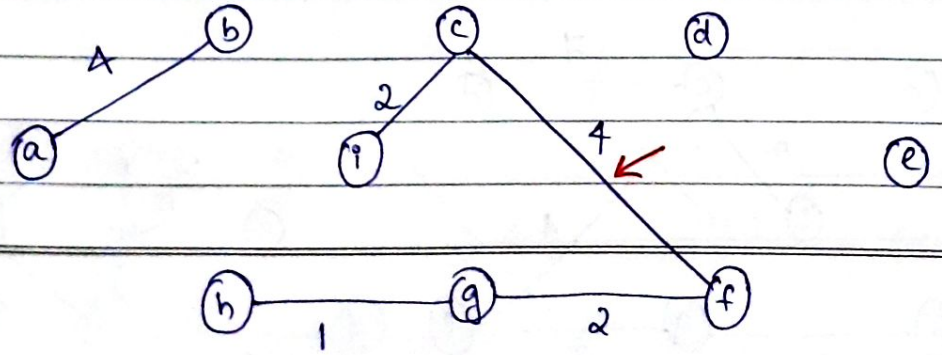
c)



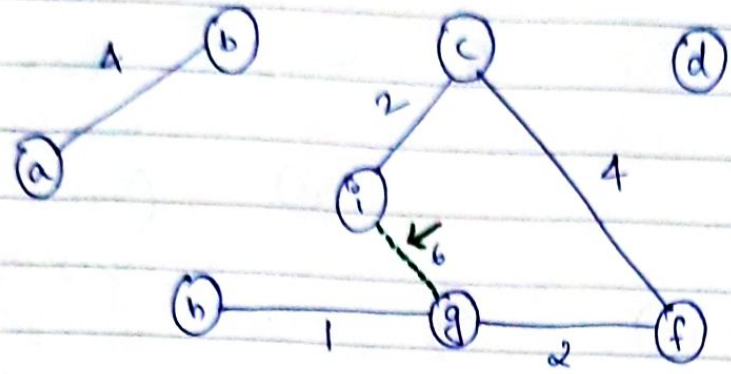

d)




e)

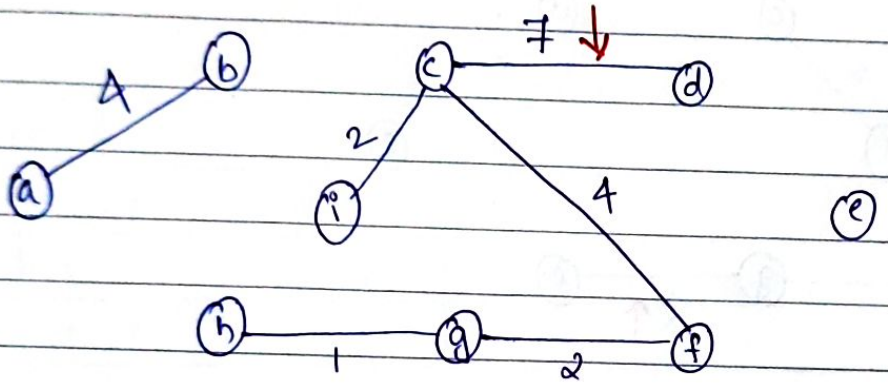


f)

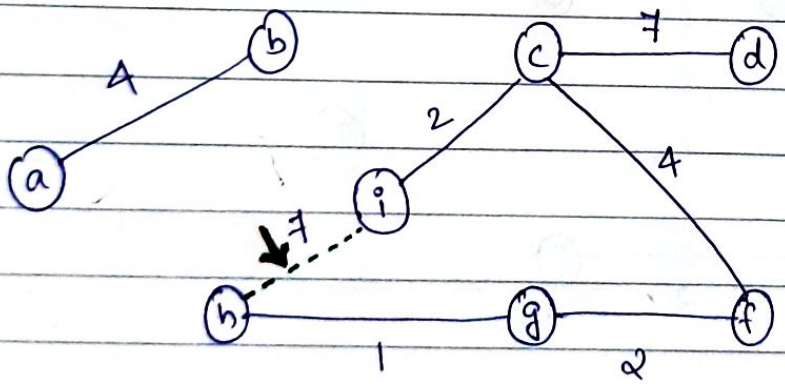



Here, we cannot choose the weight 6. Because it becomes a closed loop when we choose that weight. So, we ignored that.

g)

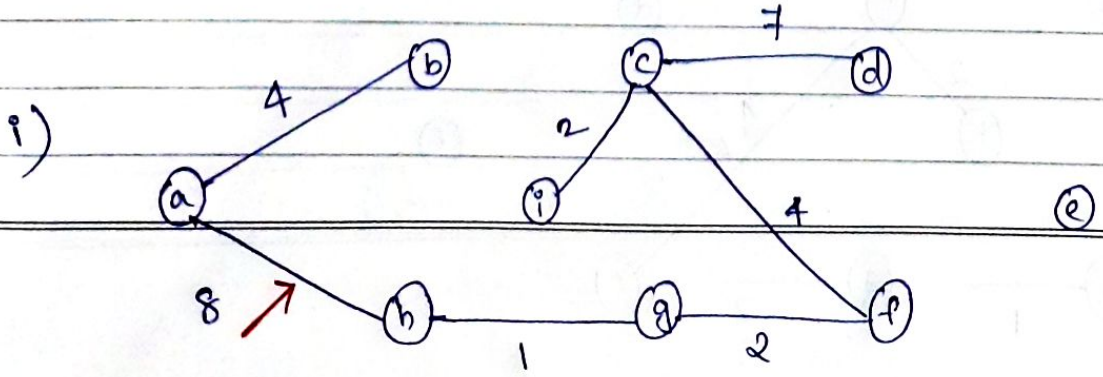



h)

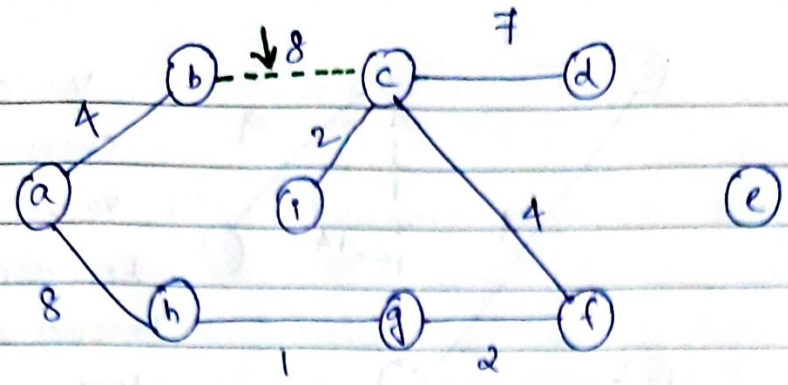



Here, we cannot choose the weight 7. Because it becomes a closed loop.

i)

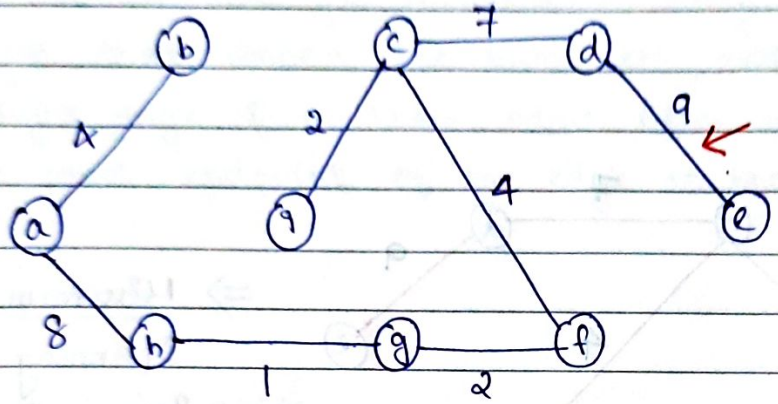


j)

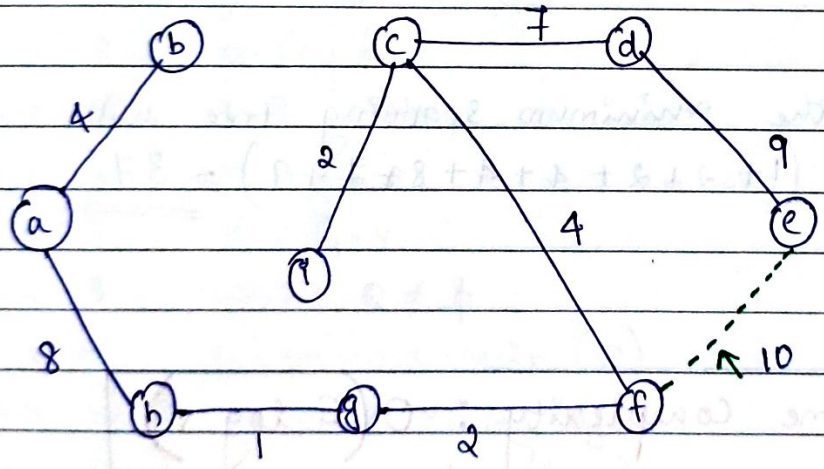



We cannot choose this weight, 8. Because it becomes a closed loop.

k)

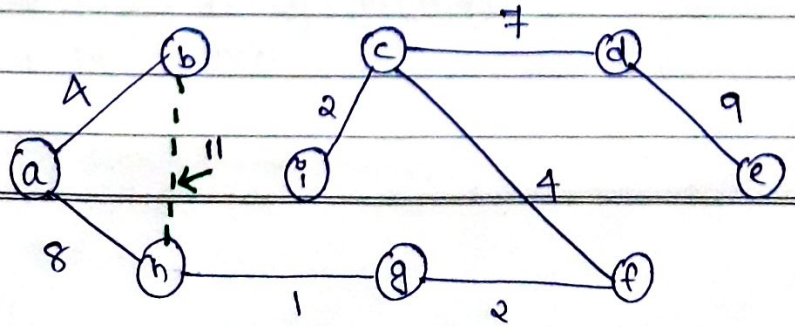



l)



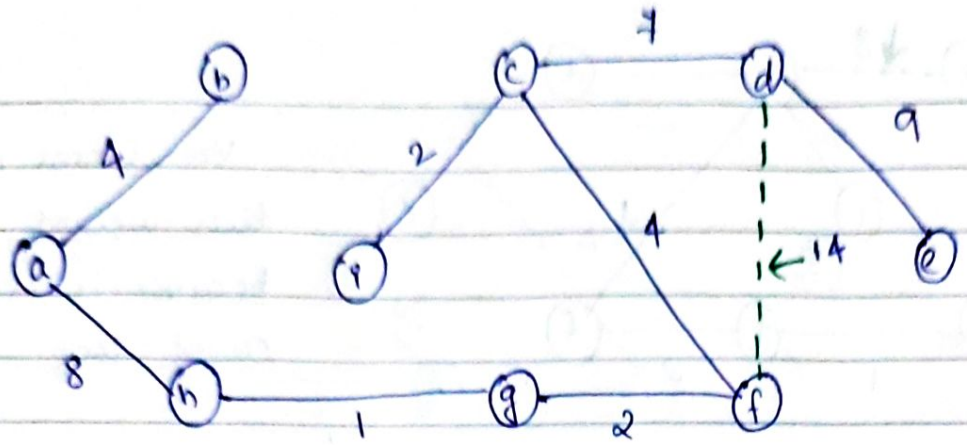

We cannot choose the weight, 10. Because it becomes a closed loop.

m)



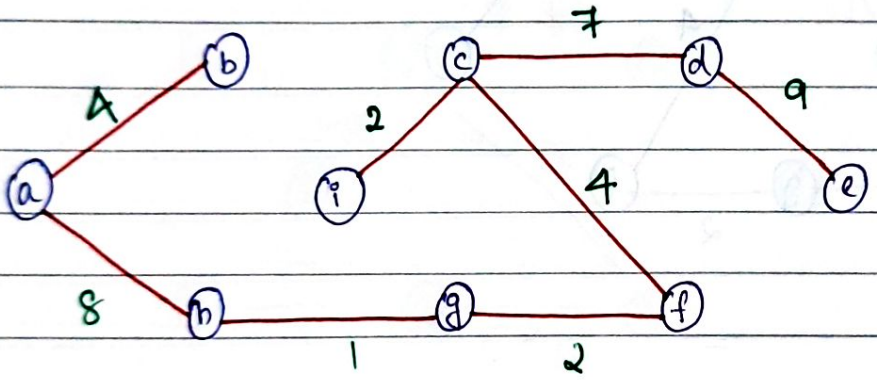

We cannot choose the weight 11. Because it becomes a closed loop.

n)




We cannot choose the value 14. Because it becomes a closed loop.

o)




⇒ Minimum Spanning Tree.

This is the minimum spanning tree with total cost $(1+2+2+4+4+8+7+9) = 37$

**Running Time Complexity : $O(E \log V)$**

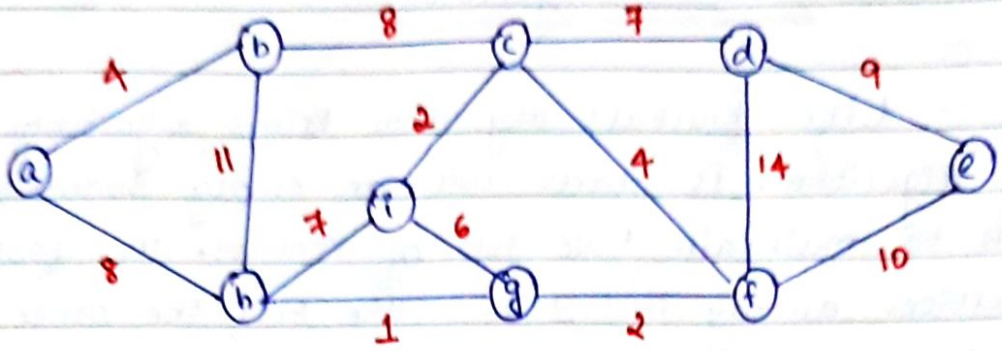
# Prim's Algorithm

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

## Algorithm

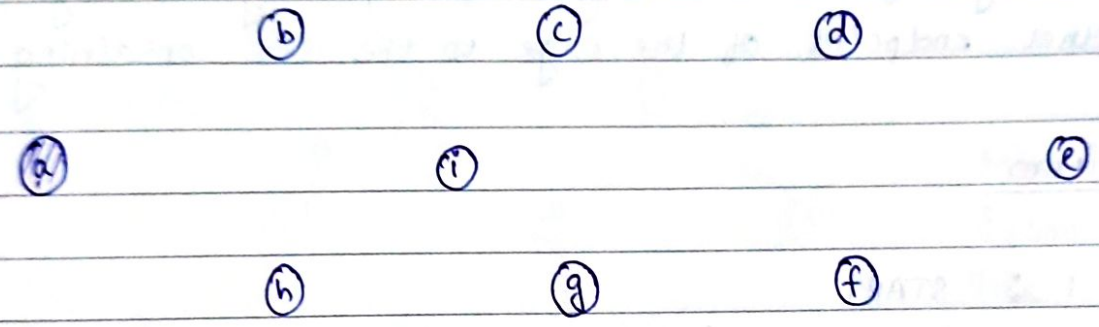
Step 1 : START
Step 2 : MST-PRIM (G, w, r)
Step 3 : for each $u \in G.V$
Step 4 : $u.key = \infty$
Step 5 : $u.\pi = NIL$
Step 6 : $r.key = 0$
Step 7 : $Q = G.V$
Step 8 : while $Q \neq \phi$
Step 9 : $u =$ Extract-min $(Q)$
Step 10 : for each $v \in G.Adj[u]$
Step 11 : if $v \in Q$ and $w(u,v) < v.key$
Step 12 : $v.\pi = u$
Step 13 : $v.key = w(u,v)$
Step 14 : STOP

Example:



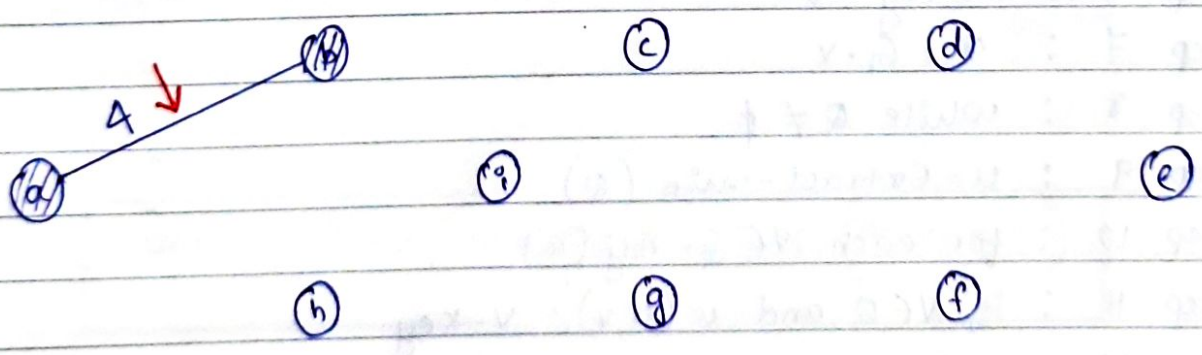

Solution.

a)




Select any node Here we choose the node 'a'.

b)



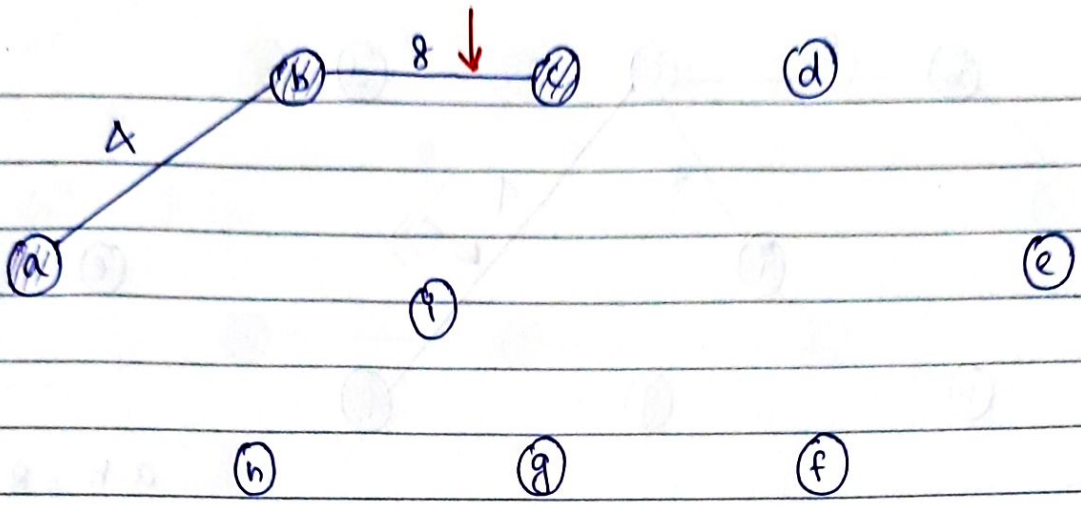

$a \to b = 4$ ✓

$a \to h = 8$

Minimum weight = 4.

So we choose that edge.

c)




a - h = 8

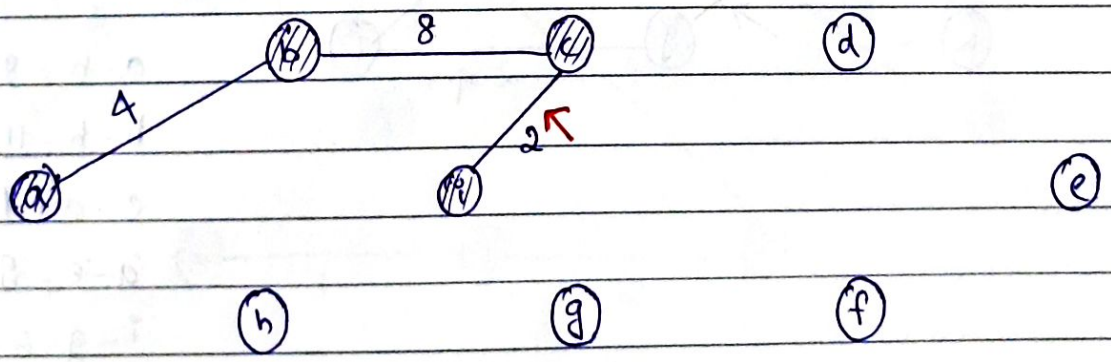
b - c = 8

b - h = 11

Here, we can choose either a-b or b-c (both have same weights).

Here, we select b-c.

d)




a - h = 8

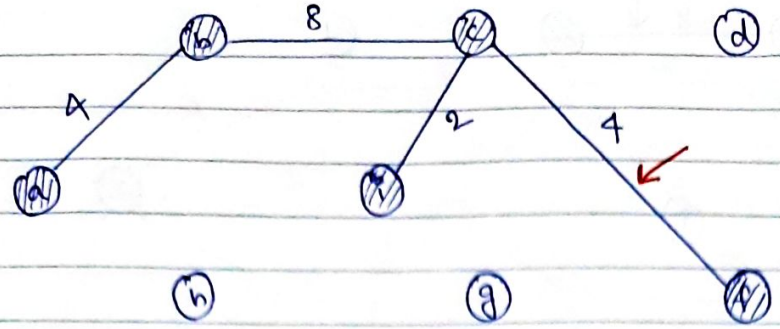
b - h = 11

c - i = 2 ✓

c - d = 7

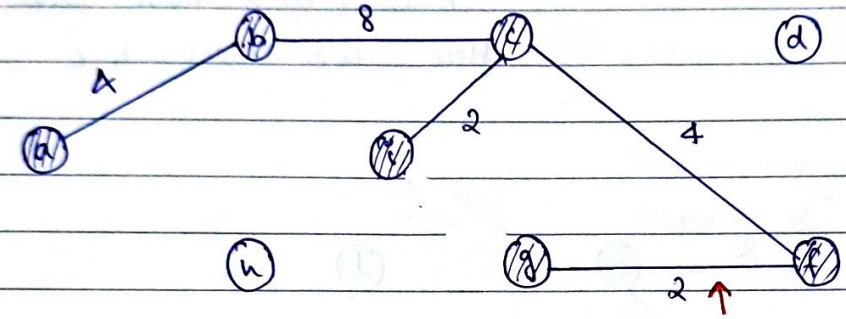
c - f = 4

e)



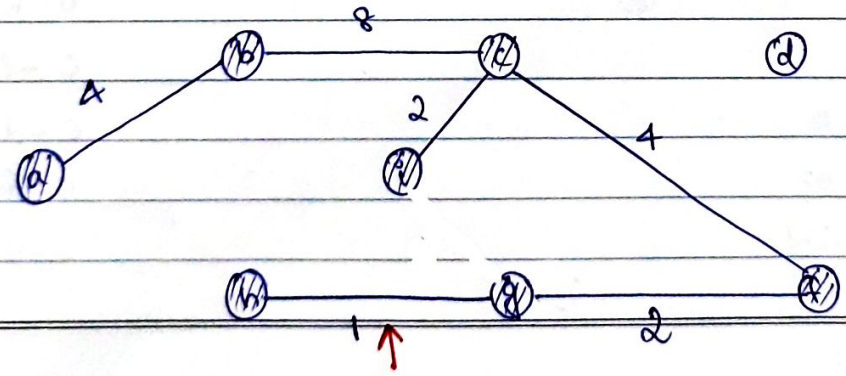

a-h = 8
b-h = 11
c-d = 7
c-f = 4 ✓
i-h = 7
i-g = 6

e)



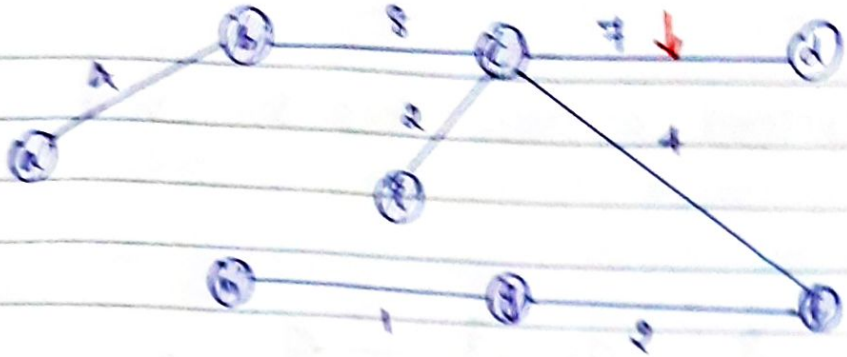

a-h = 8
b-h = 11
c-d = 7
i-h = 7
i-g = 6
f-g = 2 ✓

g)




a-h = 8
b-h = 11
c-d = 7
i-h = 7
f-e = 10
g-h = 1 ✓
f-d = 14
i-g = 6

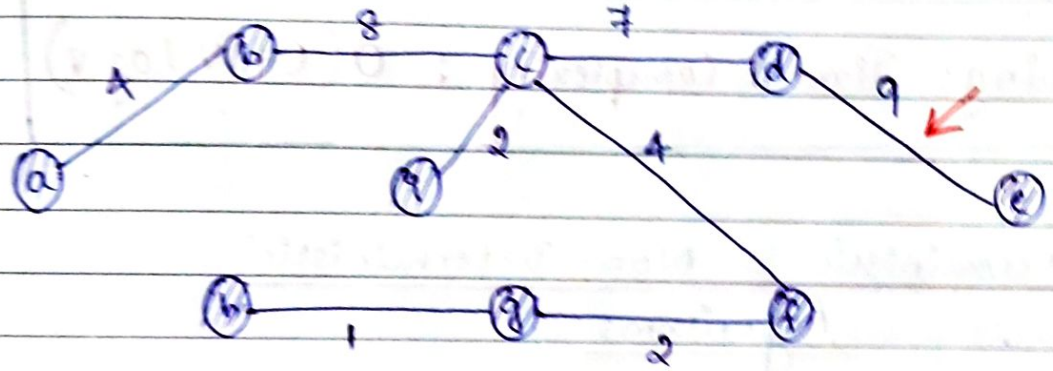
h)




a - h = 8 ✗
b - h = 11
c - d = 7 ✓
i - g = 6 ✗
i - h = 7 ✗
f - e = 10
f - d = 14

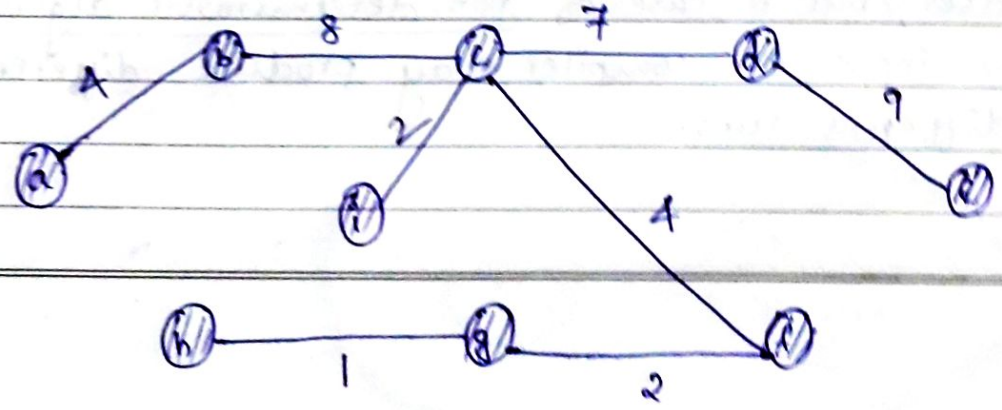
Here minimum weight = i-g → 6. But we cannot choose that weight. Because if we choose that edge, it becomes a closed loop. Next, i-h → 7. We cannot choose that edge also. That also becomes a closed loop. Next, we consider c-d = 7 (No closed loop). So, we select this edge.

i)




a - h = 8 ✗
b - h = 11
i - g = 6 ✗
i - h = 7 ✗
f - e = 10
f - d = 14
d - e = 9 ✓

j)

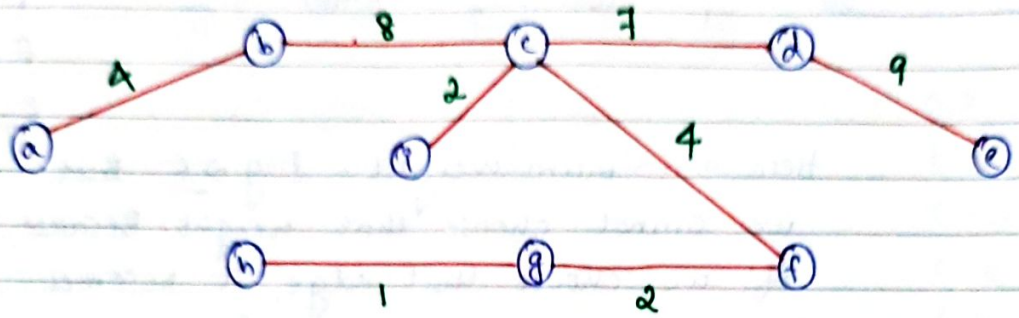



a - h = 8 ✗
b - h = 11 ✗
i - g = 6 ✗
i - h = 7 ✗
f - e = 10 ✗
f - d = 14 ✗

We cannot choose any edges. If we choose

any weight, it becomes a closed loop. We visit all nodes.

final figure ⇒




This is minimum spanning tree with total cost $(4+8+7+9+2+4+1+2) = 37$

**Running Time Complexity : $O(E + V \log V)$**

## Deterministic & Non-Deterministic Algorithms

In Deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states, but in case of non-deterministic algorithm, for the same input, the compiler may produce different output in different runs.

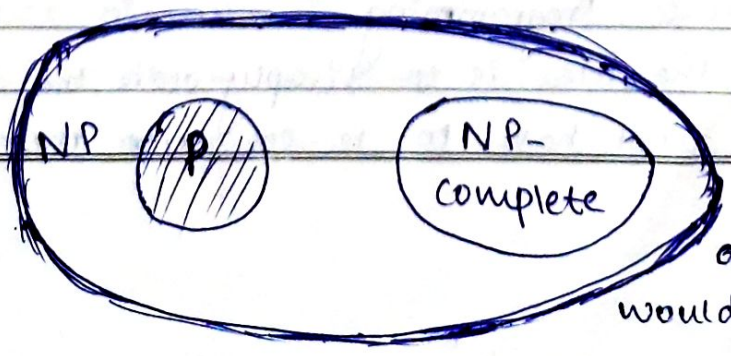
| Deterministic Algorithm | Non-Deterministic Algorithm |
|---|---|
| for a particular input the computer will give always same output. | for a particular input the computer will give different output on different execution |
| Can solve the problem in polynomial time. | can't solve the problem in polynomial time. |
| Can determine the next step of execution. | Cannot determine the next step of execution due to more than one path the algorithm can take. |

## NP-Hard & NP-Complete

The NP problems are set of problems whose solutions are hard to find, but easy to verify & are solved by Non-deterministic machine in polynomial time.

**NP-Hard problem** : Any decision problem, $P_i$ is called NP-Hard if & only if every problem of NP (P ∠ Subject) is reducible to $P_i$ in Polynomial time.

**NP-Complete problem**: Any problem is NP-complete ~~problem~~ if it is a part of both NP & NP-Hard problem.




If a Polynomial time algorithm exists for any of these problems, all problems in NP would be Polynomial

time solvable. These problems are called NP-complete.

| | NP- Hard | NP- complete |
|---|---|---|
| * | NP-Hard problems (say x) can be solved if and only if there is a NP-Complete problem (say y) can be reducible into x in polynomial time. | NP-complete problems can be solved by deterministic algorithm in polynomial time. |
| * | To solve this, it must be a NP problem. | To solve this problem, it must be both NP & NP-Hard problem. |
| * | It is not a decision problem. | It is exclusively decision problem. |