chap-2

Greedy Method

Greedy approaches are applicable to optimization problems.

A optimization problem is associated with an objective function called a value, a predicate P that ~~The greedy~~ represents feasibility criteria, a solution space of all possible solutions U, and an extremum (maximum or minimum) requirement. The aim of solving an optimization problem is to find a set of feasible solutions that satisfies the predicate P and achieve the desired extremum. Such a solution is called an optimal solution. ~~one~~ One example of such optimization problem is, finding the shortest path between vertices of a graph.

Ex: Finding the shorest path:

To find the shorest path between cities, say Chennai and pune, in a road map. It can observed that the objective here is to find the shortest path and reduce the effort.

Making Coin Change:

Let us assume that a friend requests you to lend him 67 paise assuming the denomination of coins {1, 2, 5, 10, 50}. would you give him 67 one-paisa coins? An effective optimal solution for this request would ideally be {50, 10, 5, 2} coins. It can be observed that the goal is to minimize the number of coins.

The problems like "making coin change" and "Finding the shortest path" have objective functions and a set of constraints. The solution to the problem is to create a subset of solutions that satisfies constraints associated with the problem. Any subset solution that satisfies these constraints is called a feasible solution. Any feasible solution that maximizes or ~~minimizes~~ minimizes the given objective function is called the optimal solution.

The greedy approach often works in stages. At every stage, a decision or choice is made. These decisions or choices are locally optimal. Finally, the global solution of the problem is obtained by combining locally optimal decisions. A typical greedy algorithm thus starts with a solution having an empty set. A greedy algorithm then progressively adds a solution at every iteration until the global solution is obtained.

```
Algorithm Greedy(a,n)
//a[1:n] Contains the n inputs
{
  solution:=0; //Initialize the solution
   for i=1 to n do
   {
      x := select(a);
      if Feasible(solution) then
           solution = solution + x;

   }
   return solution;
}
```

Summary (Basic Points)

→ Greedy algorithm can be applied to optimization problems.
→ It constructs solution through a sequence of steps.
→ On each step it makes the choice that looks best at the moment.
→ The choice made must be:

* Feasible - It has to satisfy the problem's constraints
* Locally optimal - It has to be the best local choice among all feasible choices.
* Irrevocable - once choice is made, it cannot be changed.

→ Greedy approach may not always lead to an optimal solution overall for all problems.

T.
Container loading

A large ship is to be loaded with cargo. The cargo is containerized, and all containers are the same size. Different containers may have different weights.
Problem: Load as many containers as possible without sinking the ship!

→ The ship is loaded with cargo. And the cargo is containerized. Cargo capacity be "c".
→ There are "m" containers available for loading
→ The weight of container "i" is "w_i"

→ Each weight is a positive number.

→ The volume of container is fixed.

→ Constraint: Sum of container weights should be less than C.

<u>Solution:-</u>

Load containers in increasing order of weights until we get to a container that doesn't fit.

Let $x_i \in \{0, 1\}$

where

$x_i = 0$ if i^{th} container is loaded

$x_i = 1$ if i^{th} container is not loaded.

Assign values to x_i's such that

$$\sum_{i=1}^{n} x_i w_i \leq C.$$

Ex: Consider capacity of the cargo = 400. And no. of containers are 8. Each container with weights is given as:

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
100	200	50	150	90	50	20	80

Sol:

step1: Select minimum weighted Container from the List. It is w_7.

Basing on the condition $\cancel{check}$, sum of weights of loaded Container $\leq$ C

$$20 < 400$$

$\therefore$ w_7 is added.

Step-2: Select minimum weighted Container from the remaining List. It is w_3 & w_6. Chose one. w_3 is chosen in our case.

$$20+50 < 400$$

$\therefore$ w_3 is added.

Step-3: Repeat step2 until maximum no. of Containers loaded with satisfying the Condition: total weight < C

Step4: w_6 is choosen

$$20+50+50 < 400$$

$\therefore$ w_6 is added

Step5: w_8 is choosen

$$20+50+50+80 < 400$$

$$200 < 400$$

$\therefore$ w_8 is added.

Step 6: w_5 is chosen.

$$20+50+50+80+90 < 400$$

$$290 < 400$$

$\therefore$ w_5 is added

Step 7: w_1 is chosen.

$$20+50+50+80+90+100 < 400$$

$$390 < 400$$

$\therefore$ w_1 is added.

Step-8:- w_4 is chosen.

$$20+50+50+80+90+100+\cancel{200}\,150 \not< 400$$

$$\cancel{590}\,540 \not< 400$$

Since, capacity of the cargo is exceeding w_4 is not added and process will stop.

Therefore, the optimal solution is

w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
100	200	50	150	90	50	20	80
1	0	1	0	1	1	1	1

```
Algorithm Container_loading (c, capacity, noofContainers, x)
// Greedy algorithm for Container loading
// set x[i] = 1 iff Container c[i], i≥1 is loaded.
{
  // Sort into increasing order of weight
  Sort(c, noofContainers);

  n = noofContainers;

  //initialize x

  for i=1 to n do
      x[i] = 0;

  //Select Containers in order of weights

  i=1
  while (i ≤ n && c[i].weight ≤ capacity)
  {
   //enough capacity for Container

      x[c[i]] = 1;
      capacity -= c[i].weight; //remaining capacity
      i++;
  }
}
```

Efficient

$T(n) = \underbrace{O(n\log n)}_{\text{sort (Quick sort)}} + \underbrace{O(n)}_{\text{remaining}}$

$= O(n\log n)$

Knapsack problem:

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in Combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

The knapsack problem is defined as:

"A theif considers taking "W" pounds of loot. The loot is in the form of "n" items, each with weight "w_i" and value "v_i". Any amount of an item can be put in the knapsack as long as the weight limit "W" is not exceeded.

There are two types of knapsack problems:

1. Fractional knapsack problem.
2. 0/1 knapsack problem.

Fractional knapsack problem:

Here the thief can take fractions of items, meaning that the items can be broken into smaller pieces so that thief may decide to carry only a fraction of x_i of item i, where $0 \le x_i \le 1$. Fractional knapsack problem can be solved by "greedy method".

0/1 knapsack problem:

Here the items may not be broken into smaller pieces, so thief may decide either to take an item or to leave it (binary choice), but may not take a fraction of an item. 0/1 knapsack problem do not exhibit greedy algorithm it only follow dynamic programming algorithm.

Problem:

we are given "n" objects and a knapsack or bag. Object "i" has a weight w_i and the knapsack has a capacity "m." If a fraction x_i, $0 \le x_i \le 1$, of object "i" is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

<u>Solution:</u>

Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most "m".

Formally, the problem can be stated as:

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad \text{——— (1)}$$

$$\text{Subject to} \sum_{1 \le i \le n} w_i x_i \le m \quad \text{——— (2)}$$

$$\text{and } 0 \le x_i \le 1, \; 1 \le i \le n \quad \text{——— (3)}$$

A feasible solution is any set satisfying eq (2) and (3) above.

An optimal solution is a feasible solution for which eq (1) is maximized.

<u>Example:</u>

A theif ~~a stabag~~ need to stole, the following items from a house and need to sell them in market with maximize profit. The theif bag Capacity = [redacted] 10

	Gold	Silver	platinum	money	Copper
Profit	20	10	90	30	5
weight	2	[redacted] 3	3	1	2

Sol:- Here capacity of bag is 10.

Items	Gold	silver	platinum	money	copper
p_i profit	20	10	30	30	5
w_i weight	2	3	3	1	2
x_i $\frac{P}{\omega}$	10	3.3	10	30	2.5
	1	1	1	1	$\frac{1}{2}$

10-1=9
9-2=7
7-3=4
4-3=1
1-1=0

$$\Sigma w_i x_i = 2\times 1 + 3\times 1 + 3 * 1 + 1\times 1 + 2\times \tfrac{1}{2}$$
$$= 2+3+3+1+1$$
$$= 10 \leq m \quad \text{— Satisfy's}$$

$$\Sigma P_i x_i = 20\times 1 + 10\times 1 + 30\times 1 + 30\times 1 + 5\times \tfrac{1}{2}$$
$$= 20+10+30+30+2.5$$
$$= 92.5$$

Step-1:- Since x_i is $0 \leq x_i \leq 1$, Calculate Profit/Weight

Step-2:- select the maximum P/w value, ie., Money

∴ add Money of weight "1" into bag.

Remaining bag capacity = 10 - 1 = 9

Step 3:- select the maximum P/w value ie., Gold and platinum. Chose one. Gold is choosen

∴ add Gold of weight "2" into bag.

Remaining bag capacity = 9 - 2 = 7

step-4: choose platinum.

add platinum into bag = 7 - 3 = 4

step-5:-

Next maximum silver

add silver into bag

Remaining 4 - 3 = 1

step-6: only one weight is left in the bag so, consider the next maximum P/w value ie., consider only a copper with weight 2.50, consider only a fractional part

Therefore, as only one weight is left, we select 1 weight of copper from 2 weight ie., ½ of weight.

add copper with 1 weight into bag

Remaining 1 - 1 = 0

There is no more capacity left i.e., $\Sigma w_i x_i \leq m$ is satisfied.

$$\therefore \sum_{1 \leq i \leq n} w_i x_i = 2\times1 + 3\times1 + 3\times1 + 1\times1 + 2\times\tfrac{1}{2}$$

$$= 2 + 3 + 3 + 1 + 1$$

$$= 10 \leq m \quad \text{(Satisfied condition)}$$

Profit

$$\sum_{0 \leq x_i \leq 1} p_i x_i = 20\times1 + 10\times1 + 30\times1 + 30\times1 + 5\times\tfrac{1}{2}$$

$$= 20 + 10 + 30 + 30 + 2.5$$

$$= 92.5$$

optimal Solution is

	Gold	Silver	platinum	money	Copper
profit	20	10	30	30	5
weight	2	3	3	1	2
	1	1	1	1	1/2

Ex.2 Consider the following instance of the Knapsack problem: $n=3$, $m=20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Soln

	1	2	3
Profit	25	24	15
weight	18	15	10
	1.39	1.6	1.5

$m = 20$

$20 - 15 = 5$

$5 - 5 = 0$

$\boxed{0 \quad 1 \quad 1/2}$ ← optimal solution

$\Sigma p_i x_i = 25 \times 0 + 24 \times 1 + 15 \times 1/2$

$= 0 + 24 + 7.5$

$= 31.5$

$\Sigma w_i x_i = 18 \times 0 + 15 \times 1 + 10 \times 1/2$

$= 0 + 15 + 5$

$= 20$

Fractional $= \dfrac{\text{No. of weights left}}{\text{Total weight of that item}}$

Algorithm GreedyKnapsack(m,n)

// p[1:n] and w[1:n] contain the profits and weights respectively.

// of the "n" objects ordered such that p[i]/w[i] ≥ p[i+1]/w[i+1]

// m is the knapsack size and x[1:n] is the solution vector.

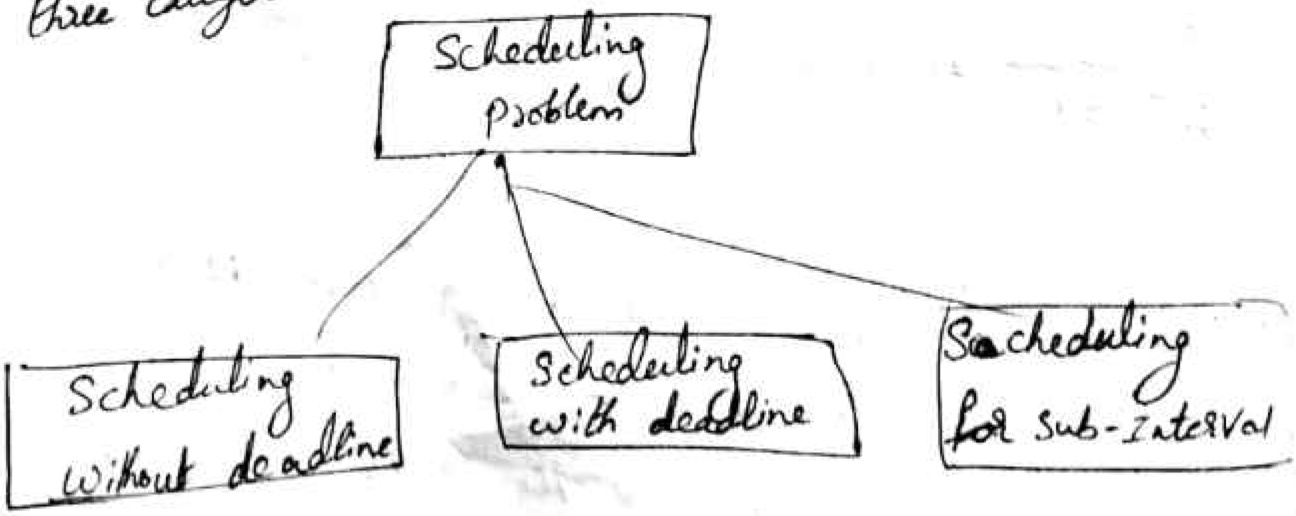
```
{
  for i=1 to n do
    x[i] = 0.0      //Initialize x.

  U = m

  for i=1 to n do
  {
    if (w[i] > U) then break;

    x[i] = 1.0;
    U = U - w[i];
  }
  if (i ≤ n) then x[i] = U/w[i] ;
}
```

<u>Efficiency:-</u> The main time taking step is the sorting of all items in decreasing order of their value/weight ratio. If the items are already arranged in the required order, then loop takes O(n) time. The average time complexity of Quick sort is O(nlogn). Therefore, total time taken including the sort is O(nlogn)

Scheduling problems

Scheduling is another popular problem in the Computer science domain. A scheduling problem is a problem of scheduling resources effectively for a given request. In operating systems, often a single processor of a Computer System may encounter many jobs or user programs. One can visualize the scheduling problem as an optimal ordering of jobs such that the job turn around time is minimized. In short, the aim is to schedule the jobs in an optimal order so as to execute the jobs faster.

This problem has many variations. Hence, scheduling problems can further be classified into three categories.



Job sequencing with deadlines:

problem:
→ Consider that there are 'n' jobs that are to be executed
→ At any time T : 1, 2, 3, ... only exactly one job is to be executed
→ Each job takes 1 unit of time
→ If job starts before (or) as its deadline profit is obtain otherwise no profit.
→ Consider all possible schedules and compute the minimum total time in the system.
→ Goal is to schedule jobs to maximize the total profit.

Ex: Solve the following job sequencing problem using greedy algorithm

Job number	1	2	3	4	5	6
profit	300	250	130	212	100	424
Deadline	4	2	3	3	3	3

Sol: Step-1: First arrange the jobs in order of their profit
Step-2: Now pick the job that gives the highest profit, and place it at the position depicted by its deadline

$0 \longrightarrow 1 \,\_\_\_\_\, 2 \,\underline{\;J_6\;}\, 3 \,\_\_\_\_\, 4$

Step-3:- For the job with second highest profit, place it at the number depicted by its deadline

0 ____ 1 ____ 2 $\underline{J_6}$ 3 $\underline{J_1}$ 4

Step-4:- For the next highest profit job, place it at the number depicted by its deadline. Since, the second position is already filled, therefore, any position before the second that is empty is chosen

0 ____ 1 $\underline{J_2}$ 2 $\underline{J_6}$ 3 $\underline{J_1}$ 4

Step-5: As per the above logic, position number 3 is filled by job number 6. Therefore, proceed backward till an empty slot is detected.

0 $\underline{J_4}$ 1 $\underline{J_2}$ 2 $\underline{J_6}$ 3 $\underline{J_1}$ 4

Step-6:- The rest of the jobs have deadlines whose positions have already been filled. Therefore, the remaining jobs cannot be done.

Hence, the profit = 424 + 300 + 250 + 212 = 1186

Algorithm jobsequencingwithdeadlines( )

{

//@Sort all jobs in decreasing order of profit

//@Sort all jobs in ~~deco~~ decreasing order of profit.

//ⓑIterate on jobs in decreasing order of profit.

For each job, do the following.

/* i, Find a time slot i, such that slot is empty and i < deadline and "i" is greatest. put the job in this slot and mark this slot filled.

ii, If no such i exists, then ignore the job. */

}

Efficiency:-

$$T(n) = \underbrace{O(n\log n)}_{\text{Sort}} + \underbrace{O(n)}_{\text{Remaining Part}}$$

$$= O(n\log n)$$

I.

Minimum Spanning Tree

Introduction: The MST problem is to find a free tree T of a given graph G that contains all the vertices of G and has the minimum total weight of the edges of G over all such trees.

Problem Formulation

Let $G=(V,E,W)$ be a weighted connected undirected graph. Find a tree T that contains all the vertices in G and minimize the sum of the weights of the edges (u,v) of T that is

$$\omega(T) = \sum_{(u,v)\in T} \omega(u,v)$$

Tree that contains every vertex of a connected graph is called spanning tree. The problem of constructing a minimum spanning tree of MST is computing a spanning Tree T with smallest total weight.

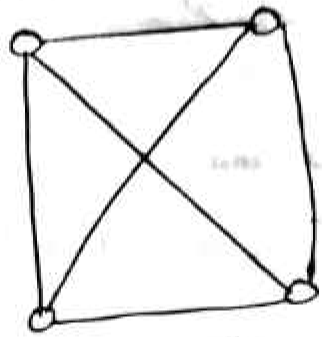
Some important points for MST:

→ A tree is a connected graph with no cycles. A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree.

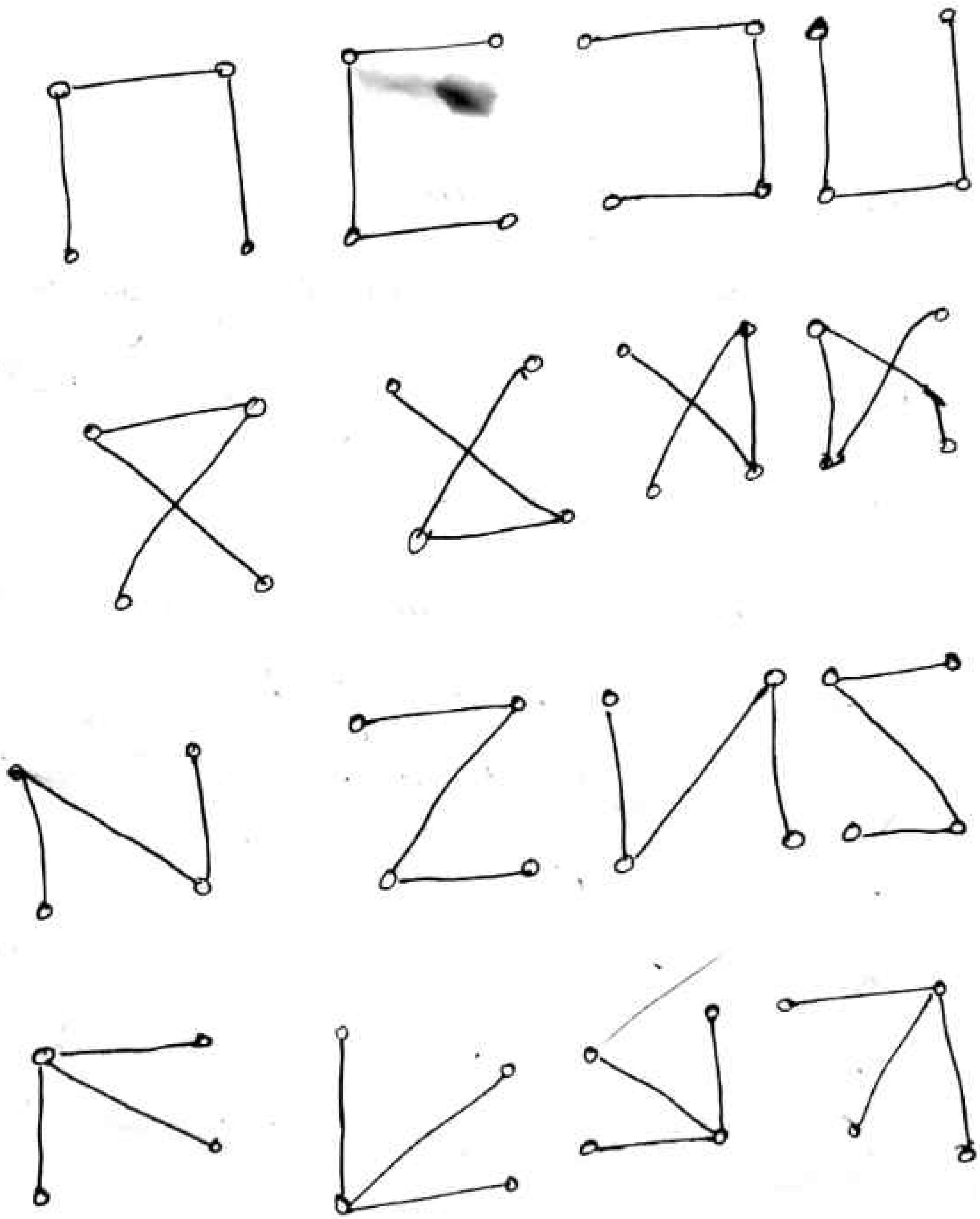
→ A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.

→ A graph may have many spanning trees.

For instane the complete graph on four vertices.



The graph has sixteen spanning trees.



Applications

→ It arises in many applications

→ It is an important example where greedy algorithms always give the optimal answer, for example - Connect all computers in a computer science building using least amount of cable.

→ Connect all logic circuit of a VLSI chip to the power source using minimum amount of telephone wires.

Kruskal's algorithm

Kruskal's algorithm is a greedy approach based algorithm. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.

Kruskal's algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm. It builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm consider each edge in turn, order by increasing weight. If an edge (u,v) connects two different trees, then (u,v) is added to the set of edges of the MST, and two trees connected by an edge(u,v) are merged into a single tree on the other hand, if an edge(u,v) connects two vertices in the some tree, then edge(u,v) is discarded.

<u>Basic structure of kruskal's Algorithm:-</u>

Start with an empty set A, and select at every stage the shortest edge that doesnot been choosen or rejected, regardless of where this edge is situated in the graph.

```
MST_Kruskal(G, w) {
A <- { }   //A will ultimately contains the edges of the MST.
for each vertex v in V(G)
do Make_set(v)
Sort edge of E by non-decreasing weights w.
for each edge(u,v) in E
do if Find_set(u) ≠ Find_set(v)
then A = A U {u,v}
UNION(u,v)
return A
}
```

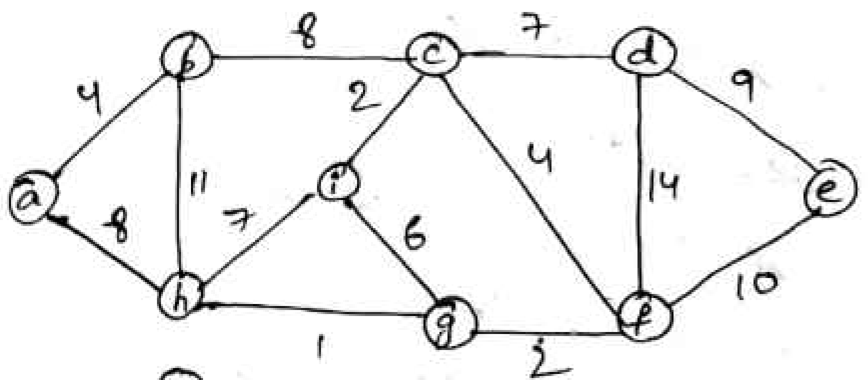
Make-Set(v): create a new set whose only member is pointed to v. Note that for this operation v must already be in a set.

Find-Set(v): Returns a pointer to the set containing v.

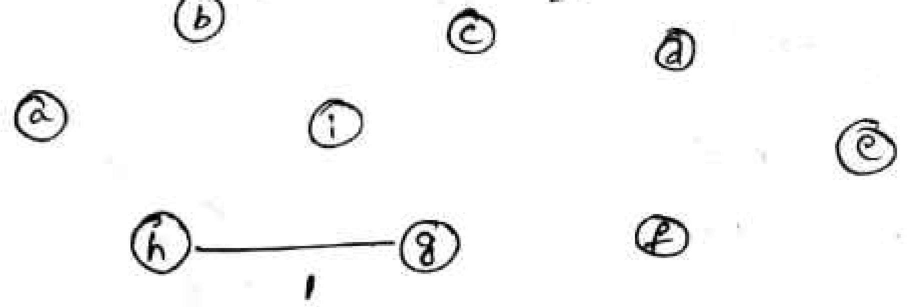
UNION(u,v): unites the dynamic sets that contain u and v, into a new set that is union of these two sets.

Efficiency: $O(n\log n) + O(n) = O(n\log n)$

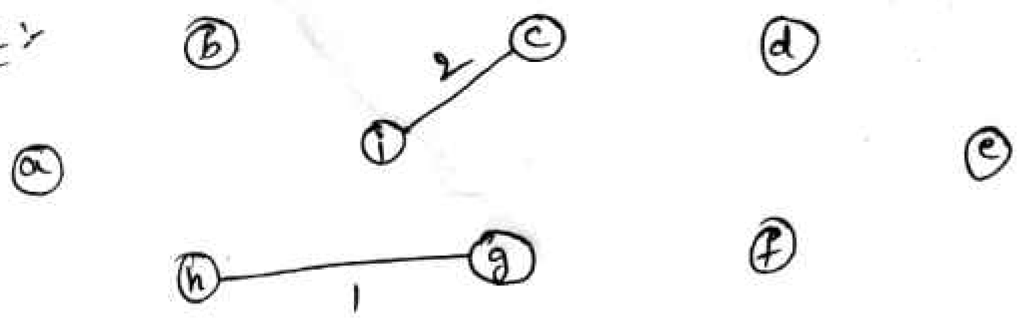
Example: Show step by step operation of Kruskal's algorithm.



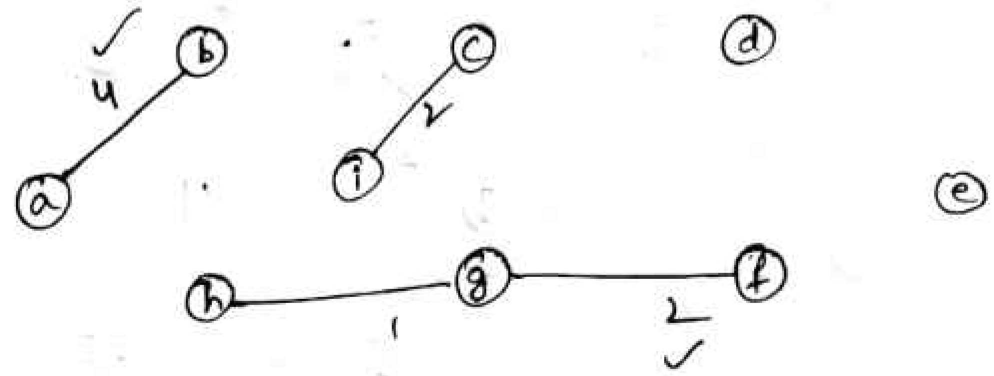
Sol:
Step-1:



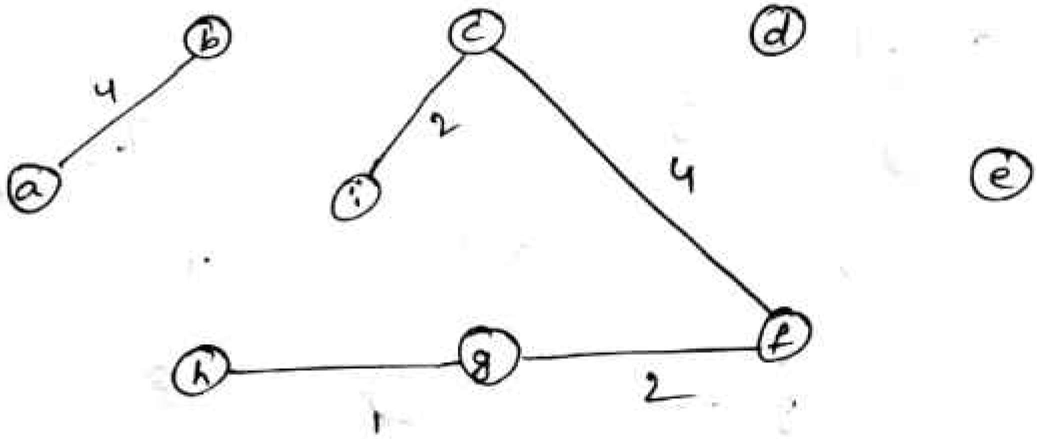
Step-2:



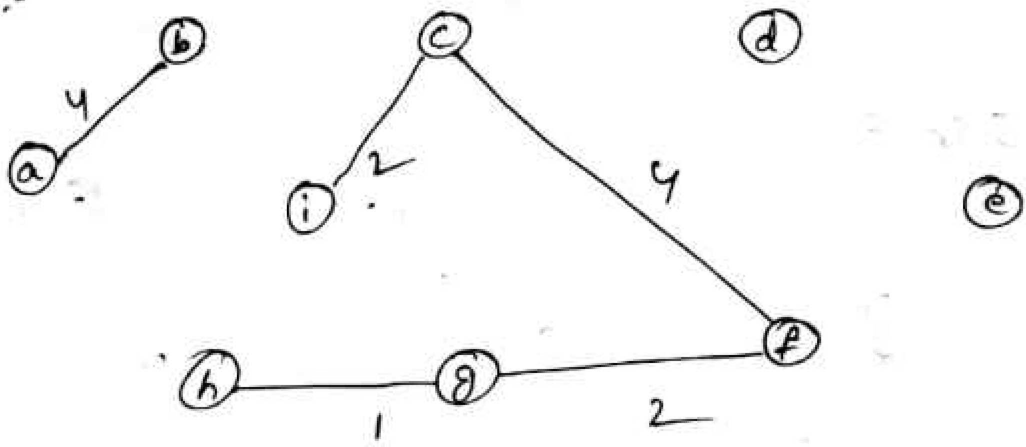
Step-4



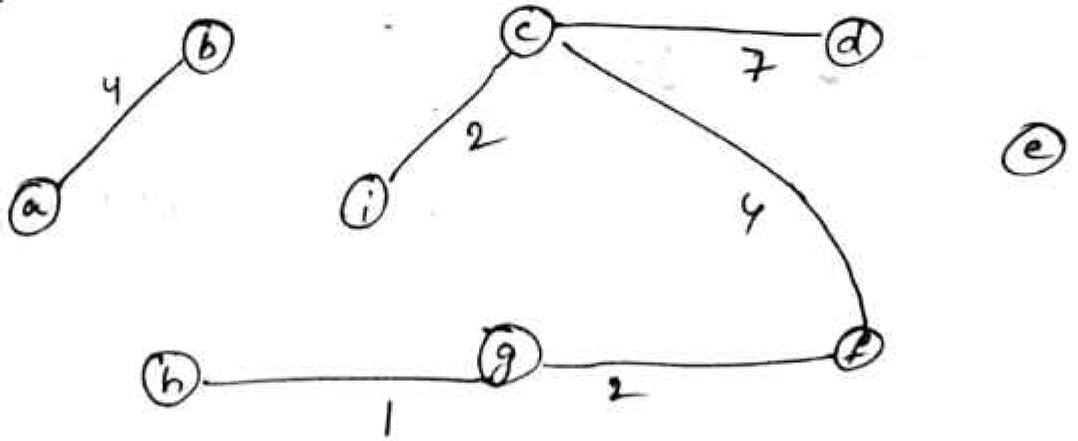
Step-5:-



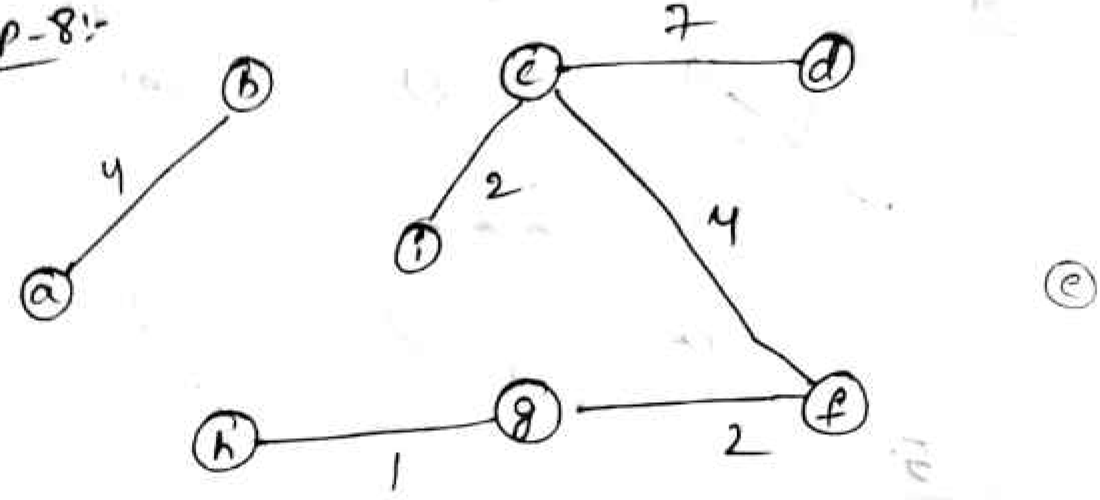
Step-6:-



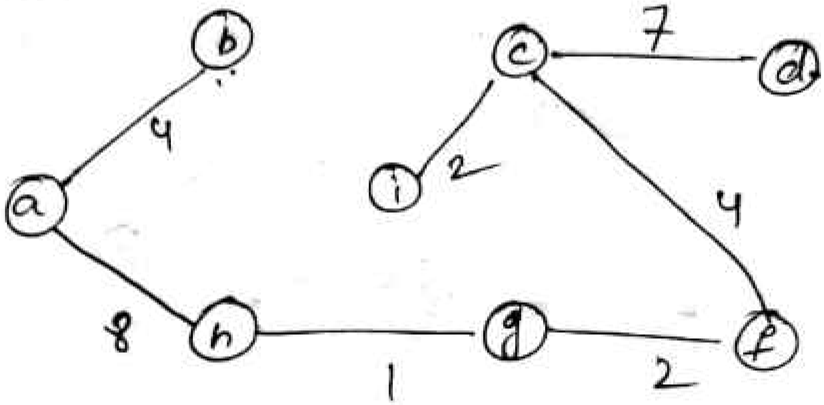
Step-7:-



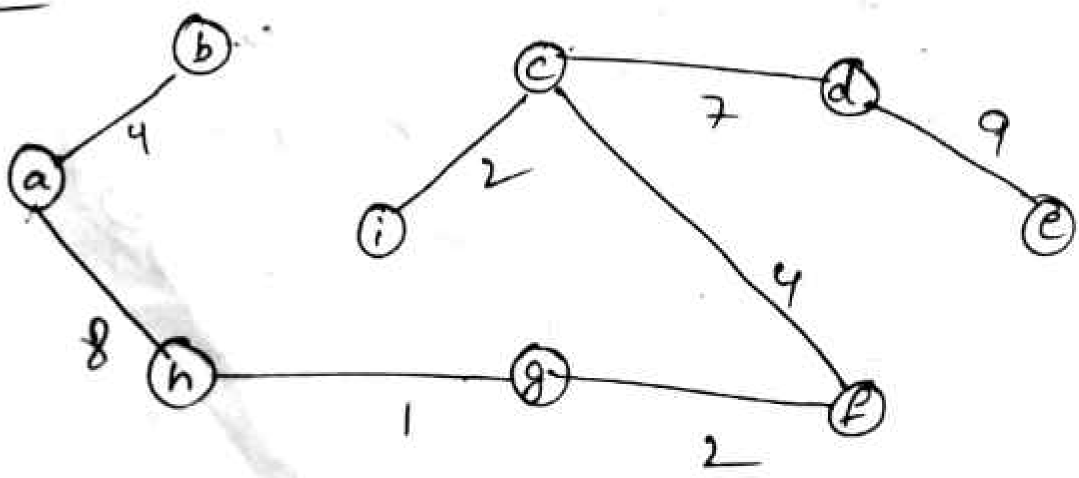
Step-8:-



Step-9:-



Step-10:-



$$\text{Minimum Cost } w(T) = \sum_{u,v \in T} WT$$

$$= 1 + 2 + 2 + 4 + 4 + 7 + 8 + 9$$

$$= 37$$

~~Prim's algorithm~~

~~Exact~~

Algorithm Kruskal(G)

{

//Input : Graph G

// output : MST T

Sort the edges and form edge list $E = \{e_1, e_2, \ldots, e_n\}$

Such that $\omega(e_1) \leq \omega(e_2) \leq \ldots \leq \omega(e_n)$

for all $v \in V$ of G do

make_set(v) //Create a singleton T

T = 0 //put all nodes and no edges in T

while T contains less than n-1 edges and $(E \neq 0)$ do

{

choose an edge(u,v)

Delete the edge(u,v) from the edge list E

//check whether the addition forms a cycle.

// if not in the same tree then add

if (find-set(u) != find-set(v)) then

{

T = T ∪ T(u,v)

UNION(u,v)

else

Discard the edge

}

}

```
if T has n-1 edges then
    return (T)
else
    output "No Spanning Tree is possible"

}
```

Prim's Algorithm

Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time. Like Kruskal's algorithm, Prim's algorithm is based on generic MST algorithm. The main idea of Prim's algorithm is similar to that Dijkstra's algorithm for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex v in a given graph $G=(V,E)$, defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (u,v). Connecting a vertex v in the set A to the vertex u outside of set A. Then vertex u is brought into A. This process is repeated until a spanning tree is formed. The important fact about MST is we always choose the smallest weight edge joining a vertex inside set A to the one out side the set A. The implication of this fact is that it adds only edges that are safe for A, therefore when the algorithm terminates, the edges in set A form a MST.

Basic of prim's algorithm:

Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

Algorithm prims

{

// Select a starting vertex.

// Repeat step 3 and 4 until there are fringe vertices.

// 3 select an edge e connecting the tree vertex and fringe vertex that has minimum weight.

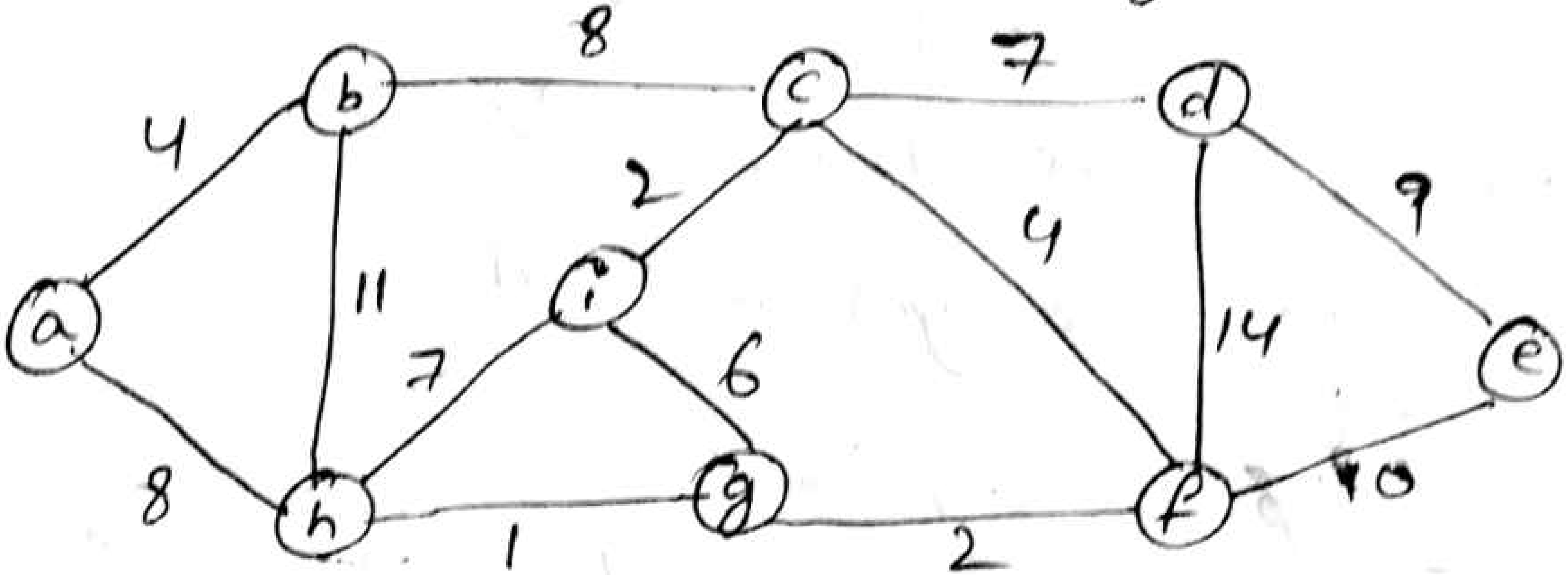
// 4 Add the selected edge and the vertex to the minimum spanning tree T.

}

Efficiency :-

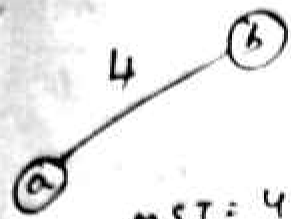
prim's algorithm has a time complexity of $O(n^2)$, 'n' being the no. of vertices and can be improved upto $O(n \log n)$.

Ex: Show the step by step operation of Prim's algorithm.



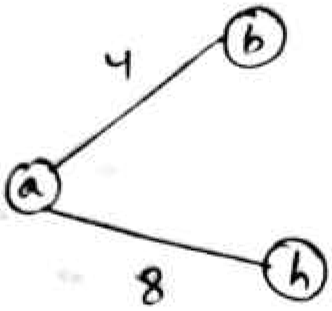
sol^n

i,



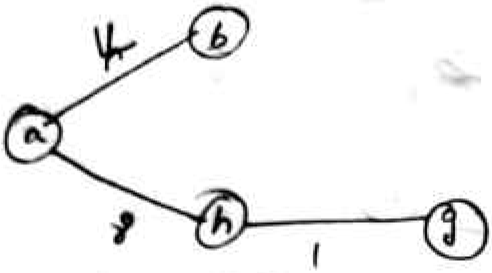
MST = 4

ii,



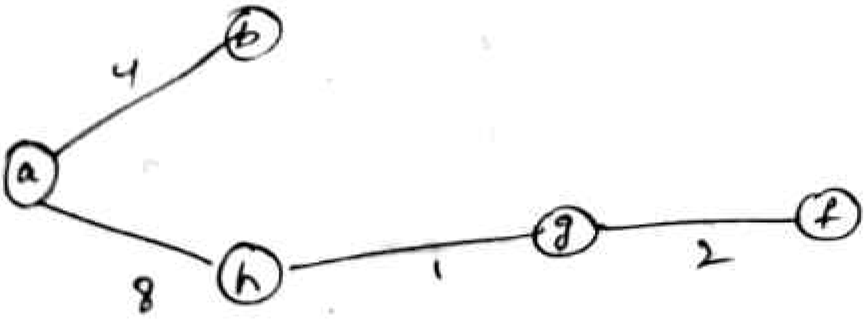
MST = 12

iii,



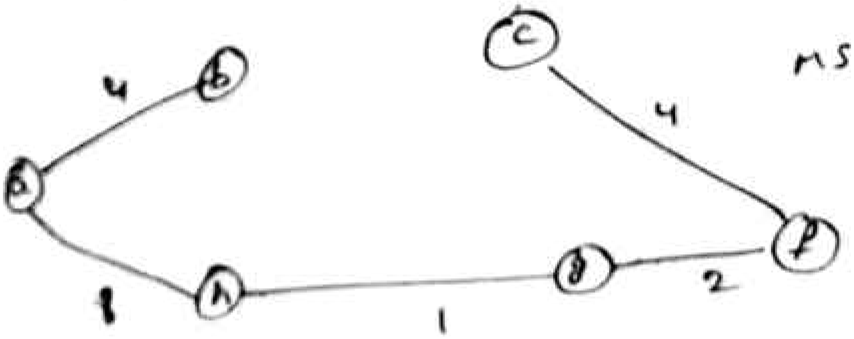
MST = 4+8+1 = 13

iv,



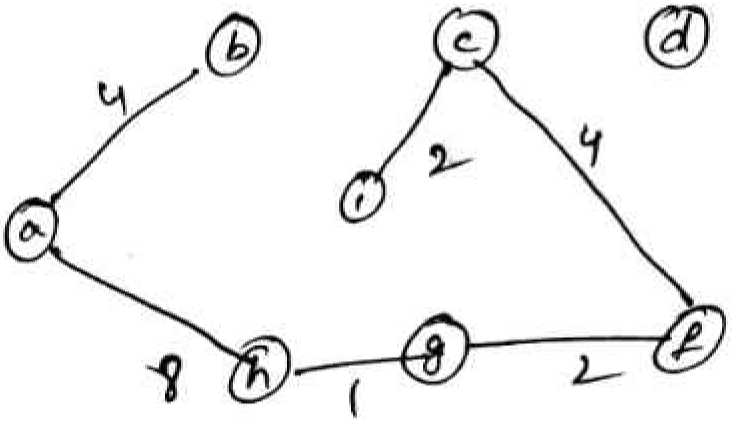
MST = 4+8+1+2 = 15

v,



MST = 4+8+1+2+4
= 19

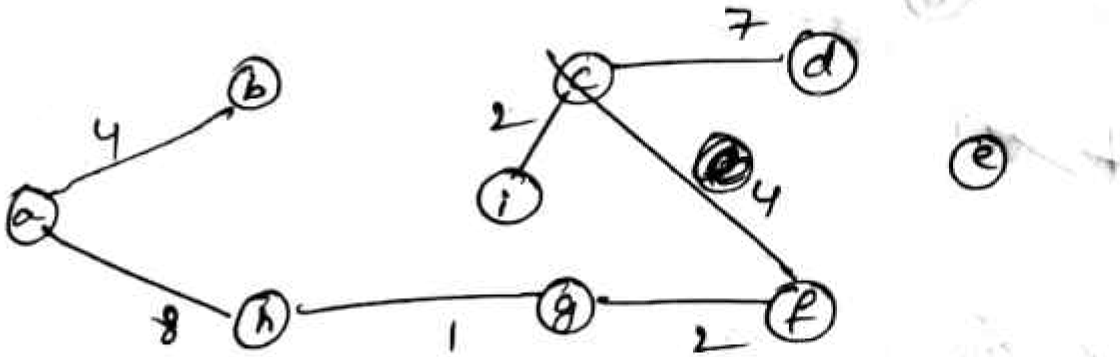
v)



$MST = 4+8+1+2+4+2$

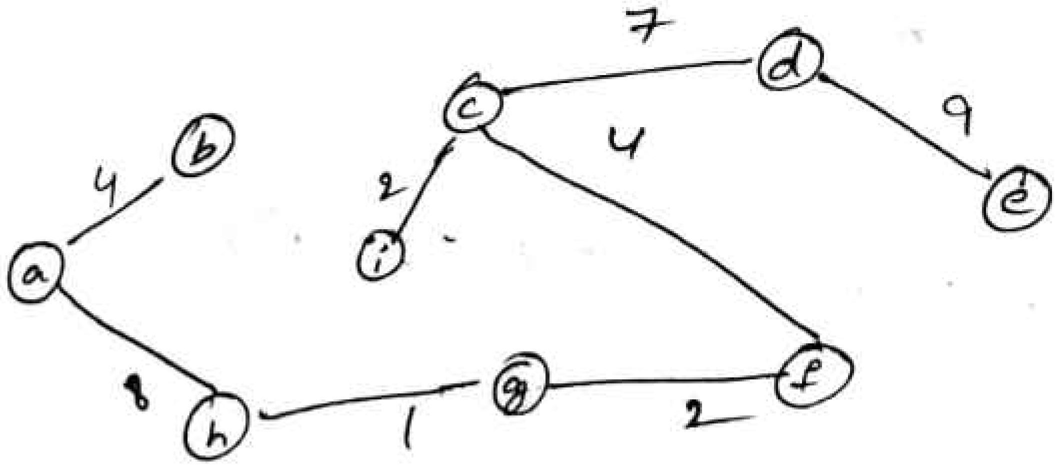
$= 22$

vi)



$MST = 4+8+1+2+4+2+7 = 29$

vii)



$MST = 4+8+1+2+2+4+7+9$

$= \underline{37}$