UNIT-4

Greedy Algorithms

⊛. Optimization Problems and Optimal Solution:

Optimization problem is the problem of finding the best solution from all feasible solutions. Optimization process can be divided into two categories depending on whether the variables are continuous or discrete. An optimization with discrete variables is known as discrete optimization. Problems with continuous variables include constrained problems and multimodal problems.

An optimal solution is a feasible solution where the objective function reaches its maximum value. For example; the most profit or the least cost. A globally optimal solution is one where there are no other feasible solutions with better objective function values. A locally optimal solution is one where there are no feasible solutions in the neighbourhood with better objective function values.

⊛. Introduction to Greedy Algorithms:

Greedy Algorithm solves problems by making best choice that seems best at the particular moment. It does not think about the future and it may or may not give the best output. A greedy algorithm works if it contains the following two properties:

1) Greedy Choice Property: An optimal solution can be obtained by creating "greedy" choices.

2) Optimal substructure: Answers to sub-problems of an optimal solution are optimal.

Greedy approach is used to solve many problems, such as:

→ Finding the shortest path between two vertices using Dijkstra's algorithm.

→ Finding the minimal spanning tree in a graph using Prism's/Kruskal's algorithm.

⊛. Elements of Greedy Strategy:

Greedy algorithms have the following five components:

i) A candidate set: A solution is created from this set.

ii) A selection function: Used to choose the best candidate to be added to the solution.

iii) A feasibility function: Used to determine whether a candidate can be used to contribute to the solution.

iv) An objective function: Used to assign a value to a solution or a partial solution.

v) A solution function: Used to indicate whether a complete solution has been reached.

⊛. Fractional Knapsack Problem: [Imp]

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is w_i and it worth v_i. Any amount of item can be put into the bag. Here the objective is to collect the items that maximize the total profit earned. Here we arrange the items by ratio v_i/w_i.

Algorithm:

```
GreedyFracKnapsack (W, n)
{
  for (i=1; i<=n; i++)
        x[i] = 0.0;
  tempW = W;
  for (i=1; i<=n; i++)
  {
    if (w[i] > tempW) then
          break;
    x[i] = 1.0;
    tempW = tempW - w[i];
  }
  if (i<=n)
      x[i] = tempW/w[i];
}
```

Analysis:

Item must be sorted first which takes $O(n \log n)$.
The algorithm takes $O(n)$ time because of for loop.
So, Total time complexity $T(n) = O(n \log n)$.

Example: Consider five items along with their respective weights and values,

$I = \{I1, I2, I3, I4, I5\}$
$W = \{5, 10, 20, 30, 40\}$
$V = \{30, 20, 100, 90, 160\}$

The knapsack has capacity $W = 60$, then find the optimal profit earned by using fractional knapsack.

Solution:

Initially (Step1):

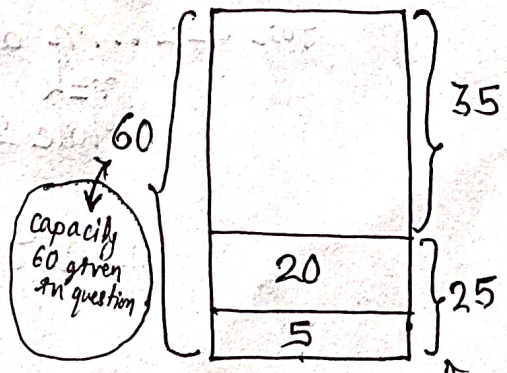
Items	W_i	V_i
I1	5	30
I2	10	20
I3	20	100
I4	30	90
I5	40	160

Step2: calculate V_i/W_i as;

Items	W_i	V_i	$P_i = V_i/W_i$
I1	5	30	6.0
I2	10	20	2.0
I3	20	100	5.0
I4	30	90	3.0
I5	40	160	4.0

Step3: Arranging the items with decreasing order of P_i as;

Items	W_i	V_i	$P_i = V_i/W_i$
I1	5	30	6.0
I2	20	100	5.0
I3	40	160	4.0
I4	30	90	3.0
I5	10	20	2.0



Now filling the knapsack according to decreasing value of P_i

Since $40w = 160v$
$1w = 160/40 = 4v$
$35w = 35 * 4 = 140v$

Thus maximum value = $v1 + v2 + new(v3) = 30 + 100 + 140 = 270$.

⊛. Job Sequencing with Deadlines:

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of jobs, which will be completed within its deadlines and it should yield a maximum profit.

Let there are a number of jobs $J = \{j1, j2, j3, j4, \ldots, jn\}$

Deadline of jobs $= \{d1, d2, d3, d4, \ldots, dn\}$.

The profit can be earned if job is completed within their deadline $= \{p1, p2, \ldots, pn\}$. Here every job can be completed in unit time (i.e, first job begins at time 0 and finished at time 1, the second job begins at time 1 and finished at time 2 and so on). and we have a single machine (processor). The main aim of this problem is to find the feasible sequence of jobs that maximize the profit earned.

Algorithm: 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. Assume, deadline of i^{th} job J_i is D_i and the profit received from this job is P_i.

```
JobSequencingWithDeadline (D,J,n,k)
{
    D(0) = J(0) = 0
    k=1
    J(1) =1 //means first job is selected.
    for i=2 ... n do
            r=k
        while D(J(r)) > D(i) and D(J(r)) ≠ r do
                r=r-1
        if D(J(r)) ≤ D(i) and D(i) > r then.
                for l=k, ..., r+1 by -1 do.
                    J(l+1) = J(l)
                J(r+1) = i
                k= k+1
}
```

Analysis:

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

Example:- Let us assume 5 jobs, with their deadlines and profit as follows:

$J = \{J1, J2, J3, J4, J5\}$

$D = \{d1, d2, d3, d4, d5\} = \{2, 1, 3, 2, 1\}$

$P = \{60, 100, 20, 40, 20\}$

Find the sequence due to which maximize the profit by using job sequencing with deadline algorithm.

Solution:

First we sort the given jobs according to their profit in a descending.

$J = \{J2, J1, J4, J3, J5\}$

$D = \{d2, d1, d4, d3, d5\} = \{1, 2, 2, 3, 1\}$ ← deadlines

$P = \{100, 60, 40, 20, 20\}$

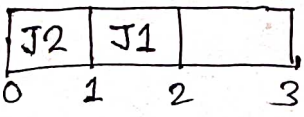
Job	Feasible/Non-feasible	Processing Sequence	Total Profit
J2	Feasible	{J2}	100
J1	Feasible	{J2, J1}	100+60=160
J4	Not feasible	{J2, J1}	160
J3	Feasible	{J2, J1, J3}	160+20=180
J5	Not feasible	{J2, J1, J3}	180

Thus, the solution is the sequence of jobs {J2, J1, J3} with total profit 180.

Explanation ← only to understand table

→ First we select J2, as it can be completed within it's deadline.

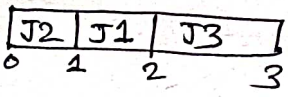
→ Next J1, is selected, and it can also be completed within deadline.



← since we have at most (max) 3 deadline so we have at most 3 feasibles. J2 has dead line 1 so executes between 0 to 1. J1 has deadline 2 so executes between 1 to 2.

→ Next J4 can not be selected (i.e, not-feasible) since it has deadline 2 and is already occupied by J1.

→ J3 executes as it has deadline 3 (can execute on 2 to 3).



→ J3 can not be selected (i.e, not-feasible).

⊛ Kruskal's Algorithm:

Algorithm:

1. Start
2. Sort all the edges from low weight to high.
3. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
4. Keep adding edges until we reach all vertices.
5. Stop.

Pseudo Code:

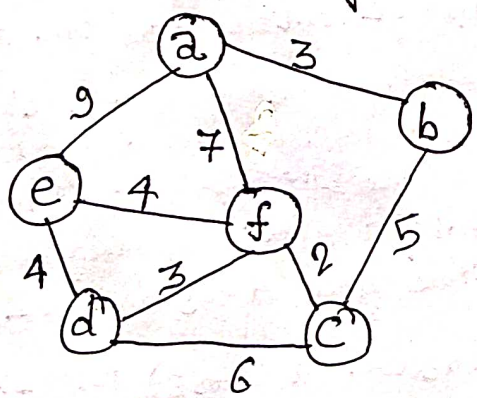
Let G1 be the weighted connected undirected graph with n vertices.

```
Kruskal MST(G1)
{
  T={V}  //collection of n vertices (i.e. tree).
  E= set of edges sorted in non-decreasing order of weight.
  while(|T|<n-1 and E!=Φ)
  {
    • Select (u,v) from E in order.
    • Remove (u,v) from E
    • If ((u,v) does not create a cycle in T))
          T=T∪{(u,v)}
```

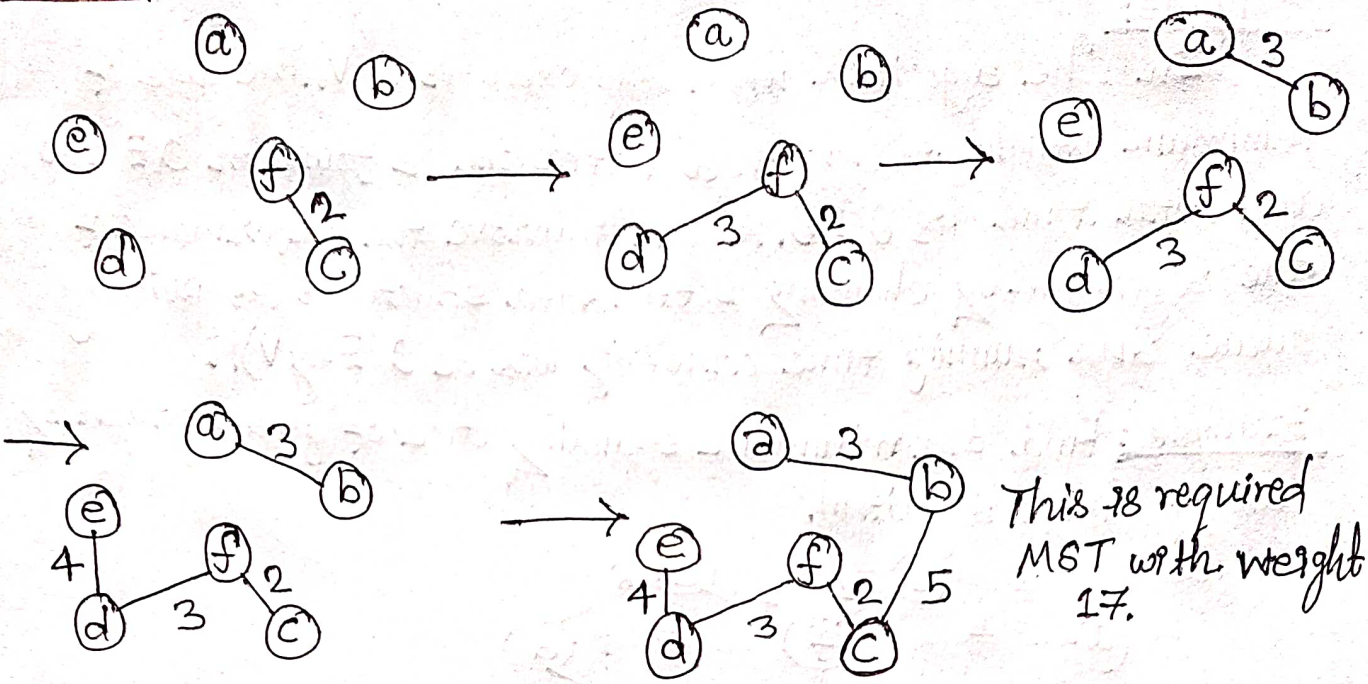
Analysis:

The tree with n vertices/nodes, at the beginning takes (V) time, The creation of set takes O(E log E) time and while loop execute O(n) times and steps inside loop takes almost linear time. So, Total time complexity T(n) = O(E log E).

Example: Find the minimum spanning tree of given graph by using Kruskal's algorithm.



Solution



This is required MST with weight 17.

The edges (d,c), (a,f) and (a,e) forms cycle so discarded these edges from tree during the process.

⊛. Prism's Algorithm: [Imp]

Algorithm:

1. Start
2. Initialize the minimum spanning tree with a vertex chosen at random.
3. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree.
4. Keep repeating step 3 until we get a minimum spanning tree.
5. Stop.

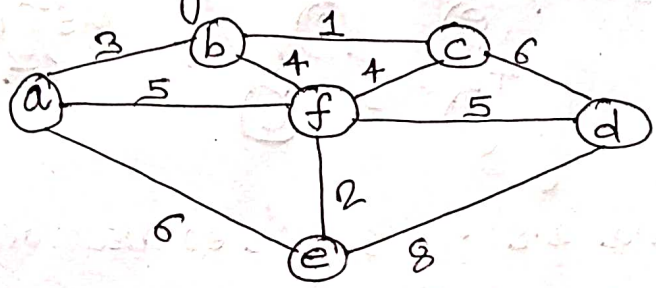
Pseudo Code:

```
PrismMST(G)
{ T=Φ; //T is a set of edges of MST.
  S={s}; //s is randomly choosen vertex and S is set of vertices.
  while(S!=V)
   { e=(u,v)   //an edge of minimum weight incident to vertices
               in T and not forming simple circuit in T if added
               to T i.e, u∈S and v∈V-S.
     T=TU{(u,v)};
     S=SU{v};
   }
}
```

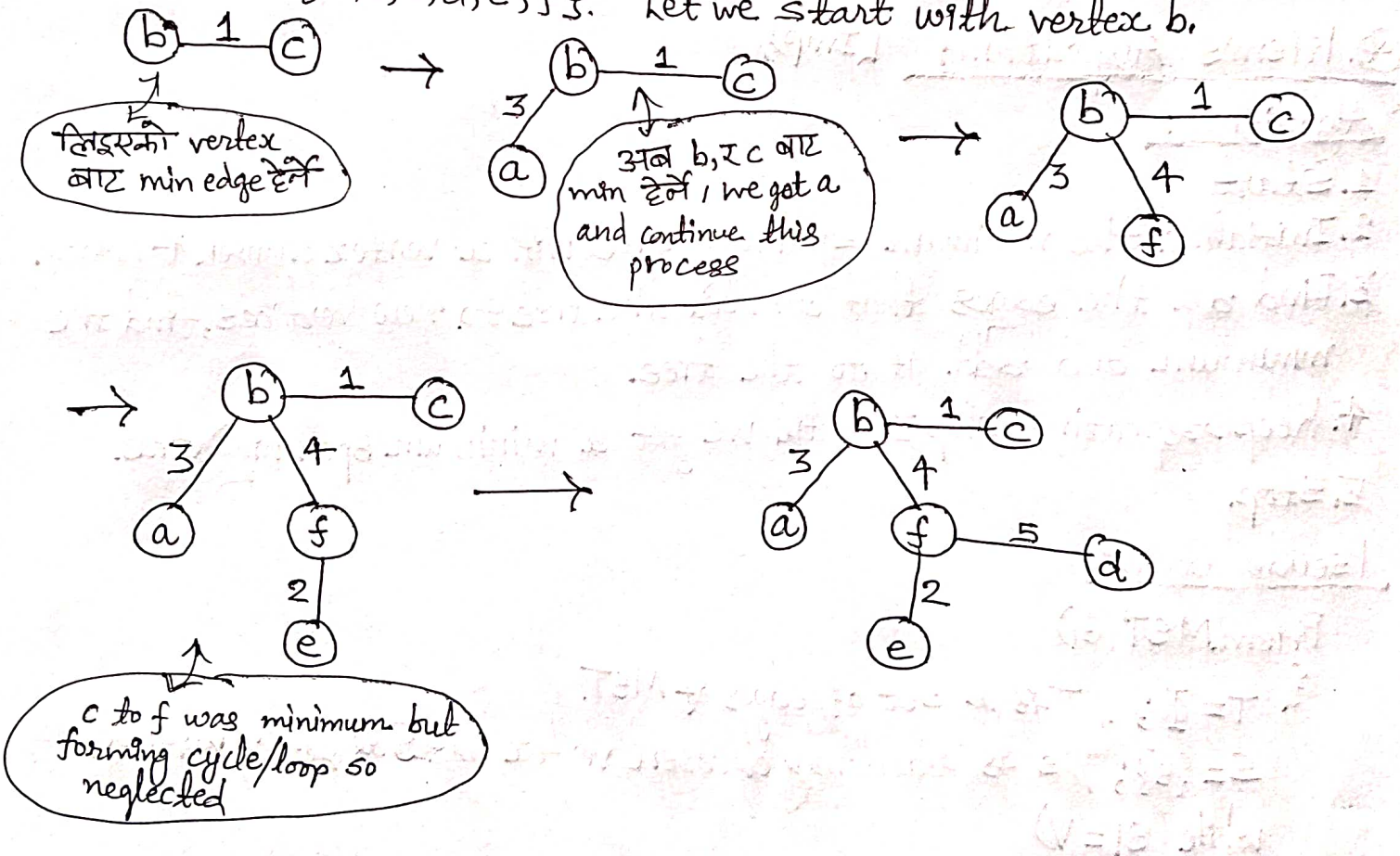
Analysis:

In the algorithm while loop executes O(V). The edge of minimum weight incident on a vertex can be found in O(E), so the total time is O(EV). We can improve the performance of the algorithm by choosing better data structures as priority queue and running time complexity will be O(E log V)!.

Example: Find the minimum spanning tree of given graph by using Prim's algorithm.



Solution: V={a,b,c,d,e,f}. Let we start with vertex b.



Hence, the total MST=15.

⊛. Dijkstra's Algorithm:

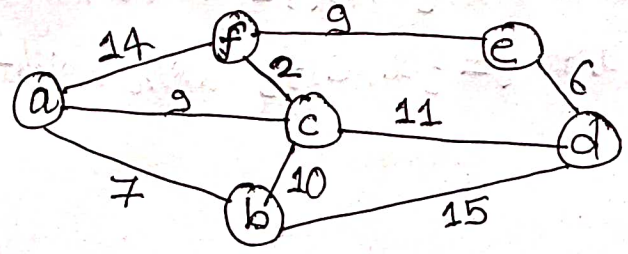
Algorithm:

```
Dijkstra_Algorithm(G,w,s)
{ for each vertex v∈V.
        d[v]=∞
  d[s]=0
  S=∅
  Q=V
  while (Q!=∅){
                u= Take minimum from Q and delete.
                S=SU{u}
                for each vertex v adjcent to u.
                if d[v]>d[u]+w(u,v) then,
                        d[v]=d[u]+w(u,v)
        }
}
```

Analysis:

In the above algorithm the first for loop block takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V) where for each execution the block inside the loop takes O(V) times. Hence, the total running time is $O(V^2)$.

Example: Find the shortest paths from the source node to all other vertices using Dijkstra's algorithm.



Solution:

Let source vertex be a, so we initialize source vertex a with 0 and ∞ to all other remaining vertices. Now we construct the table as follows for faster calculations.

Vertex Selected ↓	a	b	c	d	e	f
a	[0]	∞	∞	∞	∞	∞
b		[7]	9	∞	∞	14
c			[9]	22	∞	14
f				20	∞	[11]
d				[20]	20	
e					[20]	

[ ] → denotes selected vertex.

Rough Calculation

d[b] > d[a] + w[a,b]

∞ > 0 + 7

∞ > 7 [true]

So, d[b] = d[a] + w[a,b]

i.e, d[b] = 7.

#If condition becomes false we neglect new value and copy same old value.

#Similarly for others.

एकचोटि select गरेकोलाई vertex दोस्रो चोटि select हुँदैन।

exam मा सोधे यति rough work देखाउदा राम्रो

We can construct now a solution table to find shortest path for each vertex as follows:

Destination	a	b	c	d	e	f
Minimum Weight	0	7	9	20	20	11

जाहा source a देखि destination b कुन कुन figure मा जोडदा 7 आउछ त्यसको sequence. Similarly for others

Hence,

The shortest path from a to b = {a, b} with weight 7.

The shortest path from a to c = {a, c} with weight 9.

The shortest path from a to d = {a, c, d} with weight 20.

The shortest path from a to e = {a, c, f, e} with weight 20.

The shortest path from a to f = {a, c, f} with weight 11.

⊛ Huffman Coding: [Imp]

Huffman coding also called Huffman Encoding is a famous greedy algorithm that is used for the lossless compression of data. It uses variable length encoding where variable length codes are assigned to all the characters depending on how frequently they occur in the given text. The character which occurs most frequently gets the smallest code and the character which occurs least frequently gets the largest code.

Prefix codes: The variable-length codes assigned to input characters are prefix codes. This means that the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. Prefix codes are desirable because they clarify encoding and decoding.

Steps to build Huffman Tree:

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

Step1: 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).

Step2: Extract two nodes with the minimum frequency from the min heap.

Step3: 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 min heap.

Step4: Repeat steps 2 and 3 until the heap contains only one node. The remaining node is the root node and tree is complete.

Example: Let us take any four characters and their frequencies, and sort this list by increasing frequency. Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

Character	Frequencies
A	03
T	07
E	10
O	05

→ Now sorting these characters according to their frequencies in non-decreasing (i.e, increasing) order.

Character	Frequencies	Code
A	03	00
O	05	01
T	07	10
E	10	11

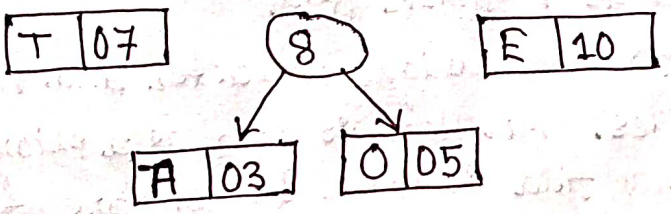
Here before using Huffman algorithm the total number of bits required is $nb = 3*2+5*2+7*2+10*2$

$= 50$ bits.

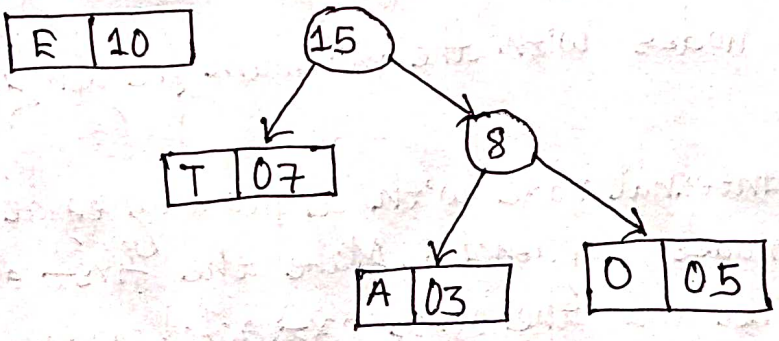
Now using Huffman algorithm as,

Step1: A | 3 ; O | 5 ; T | 7 ; E | 10.

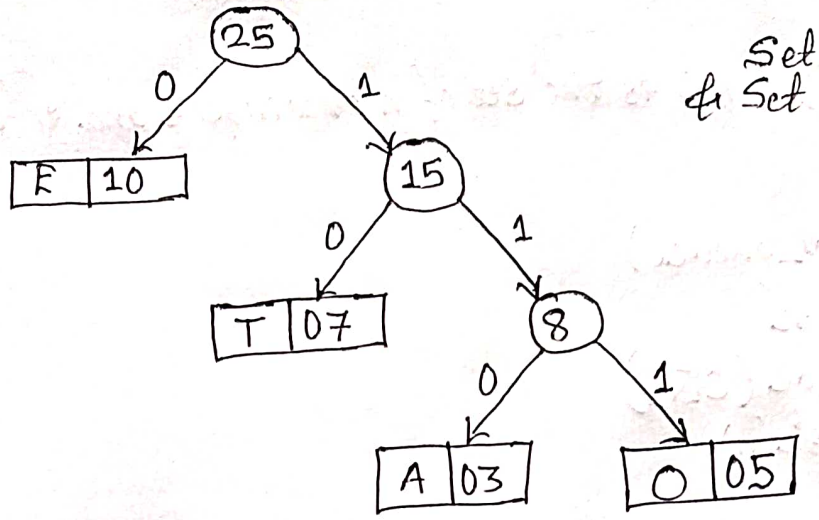
Step2:



Step3:



Step 4:



Set left branch = 0
& Set right branch = 1.

Now from variable length code we get following code sequence.

Character	Frequencies	Code
A	03	110
O	05	111
T	07	10
E	10	0

Thus after using Huffman algorithm the total number of bits required is: $3*3+5*3+7*2+10*1$

$= 48$ bits

i.e, $\frac{50-48}{50} \times 100\%$

$= 4\%$

Hence we save about 4% of space by using Huffman algorithm.

Algorithm:

```
Huffman(C)   //C is the set of n characters and information.
{
  n=C.size
  Q= priority_queue()
  For i=1 to n
      n=node(C[i])
      Q.push(n)
  End for.
  While Q.size() is not equal to 1.
  Z=new node()
  Z.left=x=Q.pop
  Z.right=y=Q.pop
  Z.frequency=x.frequency+y.frequency
  Q.push(z)
 End while
   Return Q
}
```

Analysis:

Since we need to sort the given input symbols in ascending order of their frequencies thus sorting takes $O(n \log n)$ time. The for loop executes at most $O(n)$ time also while loop executes at most $O(n)$ time.

So total time complexity $= O(n \log n) + O(n) + O(n)$

$= O(n \log n)$.