Module -3

Greedy Approach:

3.1 General Method:

- Greedy technique is one of the straight forward design method which can be applied to a wide variety of problems
- The problems that are solved using this technique has 'n' inputs and requires us to obtain a subset satisfying some contraints.
- Any subset that satisfy these constraints is called a 'feasible solution'
- The feasible solution that either minimizes or maximizes a given objective function is called 'optimal solution'

```
Algorithm Greedy (a, n)
// a[1:n] contains 'n' inputs
{
    solution = ϕ
    for i=1 to n do
    {
        x = Select (a);
        if Feasible (solution, x) then
            solution = Union (solution, x);
    }
    return solution;
}
```

Scanned with CamScanner

- function "select" picks an input from a[] & assigns to x.
- "Feasible" is a boolean valued function that determines whether 'x' can be added to solution vector.
- the function "union" combines 'x' with the solution & updates the objective function.

* **Coin - change problem:**

It is the problem of finding the number of ways of making change for a particular amount of cents 'n' using a given set of denominations $d_1, \ldots d_m$

Eg: Let $n=5$

$d_i = \{1, 2, 3\}$

Then following are the different ways of making the change

(i) <1, 1, 1, 1, 1> (ii) <1, 1, 1, 2>

(iii) <1, 1, 3> (iv) <1, 2, 2>

(v) <2, 3>

Scanned with CamScanner

Knapsack problem:

"Given 'n' objects and a knapsack (bag) with capacity 'm', where each object i has a weight "w_i" and profit associated with it "p_i"

- The objective is to obtain a filling of knapsack that maximizes the total profit earned.

i.e

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

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

$$\text{where } 0 \le x_i \le 1 \ \& \ 1 \le i \le n \quad \text{— (3)}$$

A feasible solution is any set $\{x_1 \dots x_n\}$ satisfying (2) and (3)

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

Scanned with CamScanner

① Consider the following instance of Knapsack problem & find all feasible and hence an optimal solution

$n = 3, \quad m = 20 \quad (P_1, P_2, P_3) = \{25, 24, 15\}$

$(w_1, w_2, w_3) = \{18, 15, 10\}$

I feasible solution:

- Consider decreasing order of profit P_i

i,e P_1 has highest value = 25,

so place it in knapsack first

set $x_1 = 1$ & profit $P_i x_i = 25 \times 1 = \underline{25}$ is earned.

So now, Knapsack capacity = $20 - 18 = \underline{02}$

Next largest profit is $P_2 = 24$

but $w_2 = 15$

Knapsack capacity = 02, but $w_2 = 15$

which doesn't fit into knapsack.

Hence $\frac{2}{15}$ can fill knapsack

$$\sum_{i=1 \text{ to } 3} w_i x_i = 18 \times 1 + 15 \times \frac{2}{15} = 20$$

$$\sum_{i=1 \text{ to } 3} P_i X_i = 25 \times 1 + 24 \times \frac{2}{15} = 28.2$$

Solution vector $(x_1, x_2, x_3) = (1, 2/15, 0)$

Feasible Solution = 28.2

Scanned with CamScanner

II feasible solution:

- Consider "increasing order of weight w_i"

i.e object 3 has least weight $w_3 = 10$

Hence place object 3 into knapsack & set $x_3 = 1$ & profit $p_i x_i = P_3 x_3 = 15 \times 1 = 15$

Knapsack capacity reduces to

$$20 - 10 = 10$$

- Next least weight is $w_2 = 15$, but capacity now is 10, so it does not fit

Hence use fraction $10/15 = 2/3$ to fill knapsack.

Now $$\sum_{i=1 \text{ to } 3} w_i x_i = 10 \times 1 + 15 \times \frac{2}{3} = 20$$

$$\sum_{i=1 \text{ to } 3} p_i x_i = 15 \times 1 + 24 \times \frac{2}{3} = 31$$

Solution Vector $(x_1, x_2, x_3) = (0, 2/3, 1)$

and Feasible Solution $= 31$

Scanned with CamScanner

III Feasible solution:

Consider decreasing order of ratio P_i/w_i"

$$P_1/w_1 = \frac{25}{18} = 1.38$$

$$P_2/w_2 = \frac{24}{15} = 1.6$$

$$P_3/w_3 = 15/10 = 1.5$$

- According to decreasing order of P_i/w_i first consider object 2 with weight 15 & place it in knapsack & set $x_2 = 1$

& profit $P_i x_i = 24 \times 1 = 24$ is earned

capacity reduces to $20 - 15 = 5$

- Next object 3 is considered ie $w_3 = 10$. which does not fit into knapsack and hence a fraction $5/10 = 1/2$ is used to fill knapsack $\therefore x_3 = 1/2$

Now $\sum_{i=1 \text{ to } 3} w_i x_i = 15 \times 1 + 10 \times \frac{1}{2} = 20$

$$\sum_{i=1 \text{ to } 3} P_i x_i = 24 \times 1 + 15 \times 1/2 = 31.5$$

Solution vector $(x_1, x_2, x_3) = (0, 1, 1/2)$

Feasible solution = 31.5

Scanned with CamScanner

- Now comparing all three feasible solution, it can be concluded that the 3rd feasible solution i.e 31.5 is maximum

- Hence the optimal solution is 31.5 with solution vector (x_1, x_2, x_3)

$$= (0, 1, 1/2)$$

* Obtain the optimal solution for the following instance of knapsack problem.

① $m = 40$ $n = 3$

$w_i = \{20, 25, 10\}$ $P_i = \{30, 40, 35\}$

$1 \le i \le 3$ 82.5

② $n = 3$, $m = 20$, $w_i = \{18, 15, 10\}$ $P_i = \{30, 21, 18\}$ 32.8

③ $n = 7$, $m = 15$, $w_i = \{2, 3, 5, 7, 1, 4, 1\}$

$P_i = \{10, 5, 15, 7, 6, 18, 3\}$ 55.35

Scanned with CamScanner

→ Disregarding the time to initially sort the objects, each of three strategies outline $O(n)$ time.

Theorem: If $p_1/w_1 \geq \frac{p_2}{w_2} \geq \frac{p_3}{w_3} \cdots \geq \frac{p_n}{w_n}$,

then Greed knapsack generates an optimal solution to the given instance of the knapsack problem.

Scanned with CamScanner

```
Algorithm  Greedy_Knapsack (m, n)
// finds the solution vector
// Input : m, the capacity of knapsack,
           n, number of objects &
           wi the weights of 'n' objects
// output : the solution vector x
{
                                  // initialize x
    for  i=1  to  n  do  x[i] = 0.0 ;
    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] ;
}
```

Scanned with CamScanner

3.2 **Job sequencing with deadlines:**

Given a set of 'n' jobs,

deadline d_i for each job i ($d_i > 0$)

profit $p_i > 0$ for each each job

- For any job i, the profit p_i is earned iff the job is completed by its deadline.
- only one machine is available for processing
- A job is completed, when it gets executed on a machine for one unit of time.
- A feasible solution for this problem is a subset J of jobs, such that each jobs in this subset can be completed by its deadline.
- An optimal solution is a feasible solution with maximum value.

Greedy Approach

1. Sort all jobs in decreasing order of profit.
2. Initialize the result sequence as first in sorted jobs, as late as possible
3. Do the following for remaining n-1 jobs:
   - If the current job can fit in the current result sequence without missing the deadline, add current job to the

Scanned with CamScanner

-result, else ignore the current job.

Ex: Solve the below instance of job sequencing with deadlines problem and obtain the optimal solution:

Let $n = 4$

$P_i = \{ 100, 10, 15, 27 \}$ $\quad 1 \le i \le 4$

$d_i = \{ 2, 1, 2, 1 \}$ $\quad 1 \le i \le 4$

step 1

Sort all jobs in decreasing order of profit

job	A	B	C	D
P_i	100	10	15	27
d_i	2	1	2	1

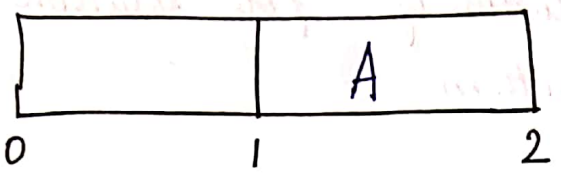
⇓ After sorting.

job	A	D	C	B
P_i	100	27	15	10
d_i	2	1	2	1

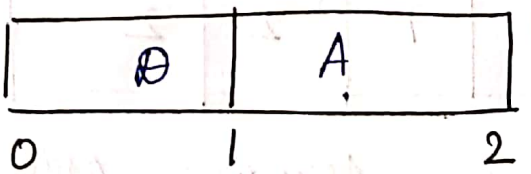
Scanned with CamScanner

Step 2 - Initialize Resultant as 1st Job and schedule as late as possible

Draw Gant chart



Step 3 : Repeat the same for others:



- other jobs are Rejected as they wont fit

∴ the selected job set

$J = \{D, A\}$

profit earned = 27 + 100 = 127

2. Solve the following instance of job sequencing problem. Let n=5, profits (10, 3, 33, 11, 40) and deadlines (3, 1, 1, 2, 2). Find the optimal sequence of execution of jobs solution using greedy algorithm

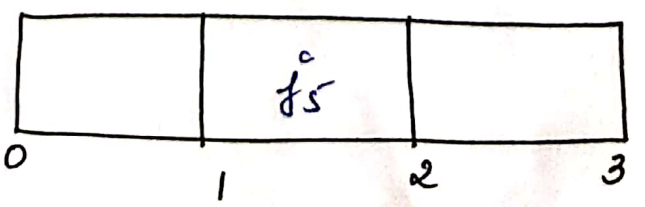
i	j_1	j_2	j_3	j_4	j_5
p_i	10	3	33	11	40
d_i	3	1	1	2	2

Step 1: Sort it in Ascending order

	j_5	j_3	j_4	j_1	j_2
p_i	40	33	11	10	3
d_i	2	1	2	3	1

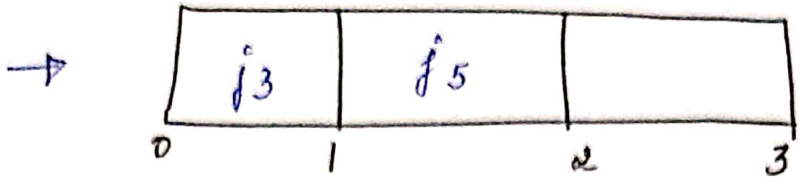
Step 2: Initialize Result as 1st job and schedule as late as possible.

Draw Gantt chart

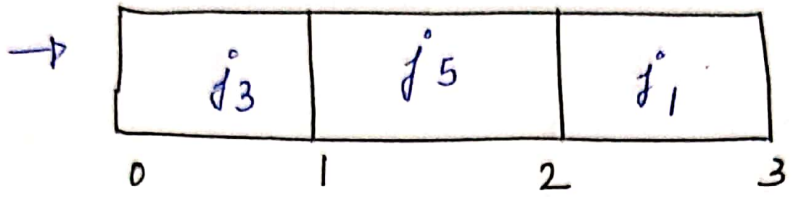


Scanned with CamScanner

Step 3 : Repeat the same for others:



j_4 is rejected as its deadline is 2, consider j_1



- j_2 is rejected as the sequence is full.

∴ the selected job set :

$$J = \{ j_3, j_5, j_1 \}$$

Profit earned = 33 + 40 + 10 = **83**

3. solve the following instance of job sequencing with deadlines problem :

p_i	20	15	10	5	1
d_i	2	2	1	3	3

Sol^n = $\{ J_1, J_2, J_4 \}$

20 + 15 + 5 = 40

Scanned with CamScanner

Prim's Algorithm:

Spanning tree: A spanning tree of a connected graph is its connected acyclic sub-graph that contains all the vertices of the graph.

Minimum spanning tree:

Minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

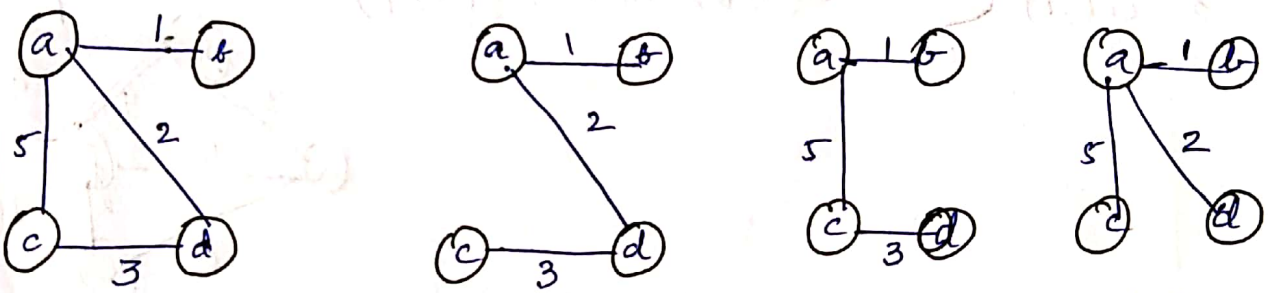


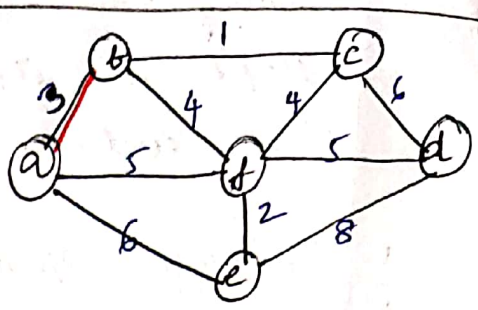
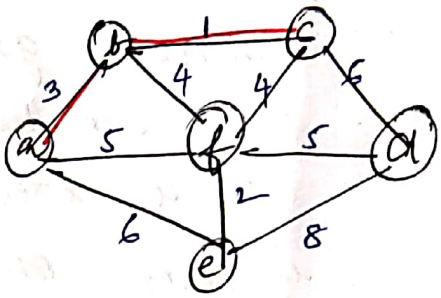
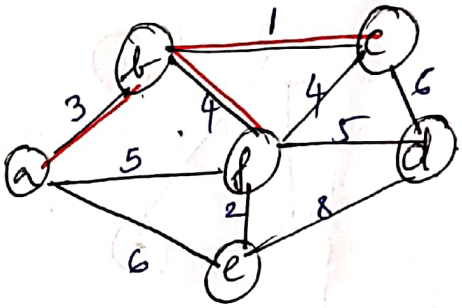
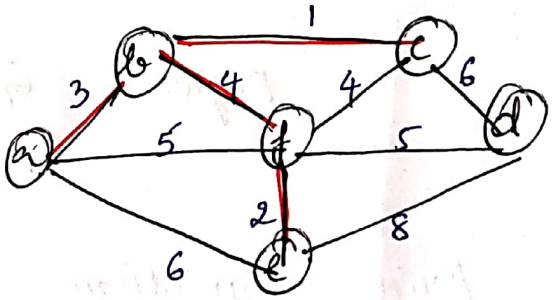
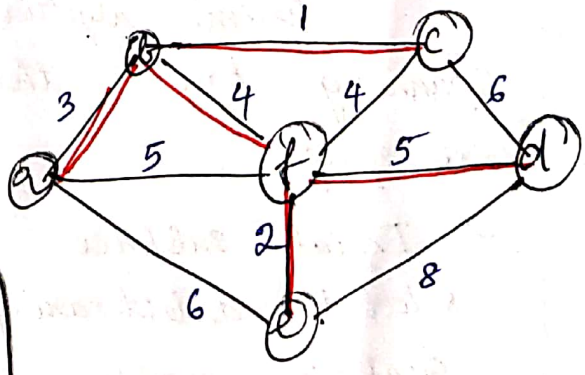
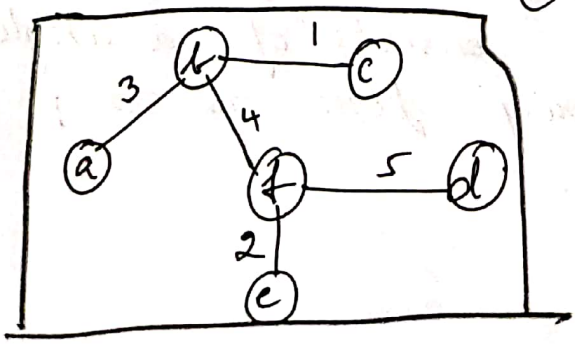
Figure: Graph & its spanning trees
T_1 is the minimum spanning tree.

Prim's Algorithm:

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub trees.

- Initial subtree consists of a single vertex selected arbitrarily from the set V of the graph's vertices.

Scanned with CamScanner

	Tree vertices	Remaining vertices	Illustration
1.	a(-, -)	b(a, 3) c(-, ∞) d(-, ∞) e(a, 6) f(a, 5)	
2.	b(a, 3)	c(b, 1) d(-, ∞) e(a, 6) f(b, 4)	
3.	c(b, 1)	d(c, 6) e(a, 6) f(b, 4)	
4.	f(b, 4)	d(f, 5) e(f, 2)	
5.	e(f, 2)	d(f, 5)	
6.	d(f, 5)		

Scanned with CamScanner

- On each iteration, the tree is expanded in greedy manner by simply attaching to the nearest vertex, that is not already selected.

- Nearest vertex means vertex which is not already selected; selut the edge of smallest weight. Ties can be broken.

- The algorithm stops after all the graphs vertices have been included in the tree being constructed.

- Since the algorithm expands a tree by exactly one vertex on each of its iterations, total number of such iterations is $n-1$, where n is the number of vertices in the graph.

-

Scanned with CamScanner

Algorithm prim(G)

// prim's algorithm for constructing minimum spanning tree

// Input: A weighted connected graph $G = (V, E)$

// Output: E_T, the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{V_0\}$

$E_T \leftarrow \phi$

for $i \leftarrow 1$ to $|V| - 1$ do

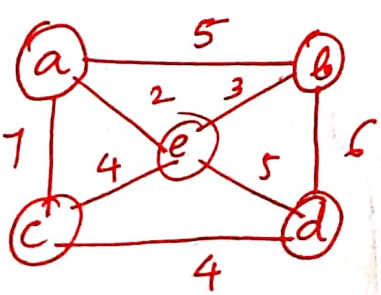
find a minimum weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$

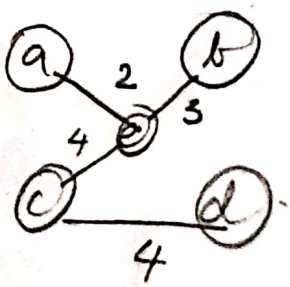
$V_T \leftarrow V_T \cup (u^*)$

$E_T \leftarrow E_T \cup (e^*)$

return E_T

- Find minimum spanning tree using prims Algorithm:





Scanned with CamScanner

How effecient is prim's Algorithm?

- It depends on data structures chosen for the graph:

(i) If graph is represented by its weight matrix and priority queue is implemented as an unordered array;

- Algorithm running time will be

$$O(|V|^2)$$

- On each $|V|-1$ iterations, the array implementing priority queue is traversed to find and delete the minimum & then to update.

(ii) If priority queue is implemented as min-heap:

- This results in $O(\log n)$
- Each $|V|-1$ iteration, so $(\log n) + V^2$

(iii) Graph is implemented with adjacency lists & priority queue - min-heap.

- Running time will be $O(|E| \log |V|)$

$$|V|-1 + |E| O \log(|V|) = \underline{O(|E| \log(|V|)}$$

Assi

(1) How can we use prim's algorithm to find a spanning tree of a connected graph with no weight on its edges?

Scanned with CamScanner

Kruskal's Algorithm:

- It is named after Joseph Kruskal.
- This is a greedy algorithm to obtain a minimum spanning tree, that yields (like prims) an optimal solution.

[Minimum Spanning tree]

- This algorithm constructs a MST as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on intermediate stages of the algorithm.
- Algorithm begins by sorting the edges in increasing order of their weights, then starting with the empty sub-graph. It scans the sorted list and adds the next edge on this list to the current subgraph, if such inclusion does not create a cycle [skip that edge otherwise)

Scanned with CamScanner

Algorithm: Kruskal's (G)

// constructs a minimum spanning tree

// Input: weighted connected graph $G = (V, E)$

// output: E_T, the set of edges composing a minimum spanning tree of G.

Sort E in non-decreasing order of the edge weights $w(e_{i_1}) \leq \ldots\ldots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \phi$;

encounter $\leftarrow 0$;

$k \leftarrow 0$

while ccounter $< |V| - 1$ do

$k \leftarrow k + 1$

if $E_T \cup \{e_{i_k}\}$ is acyclic

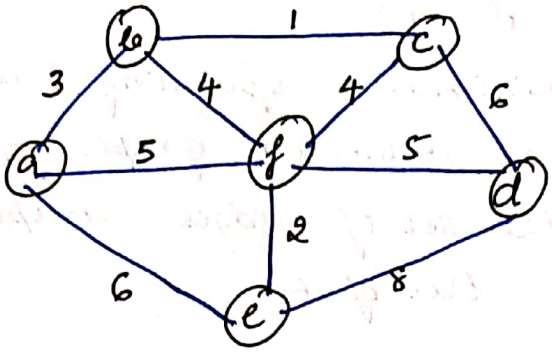
$E_T \leftarrow E_T \cup \{e_{i_k}\}$;

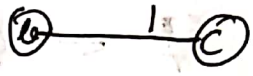
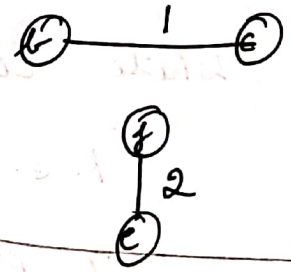
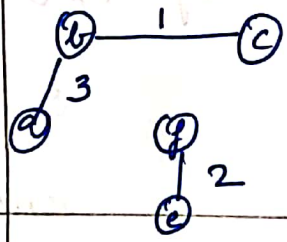
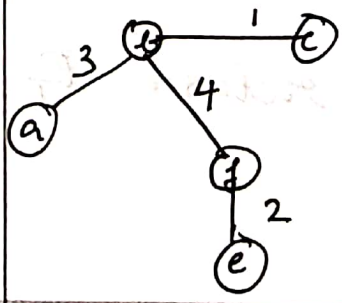
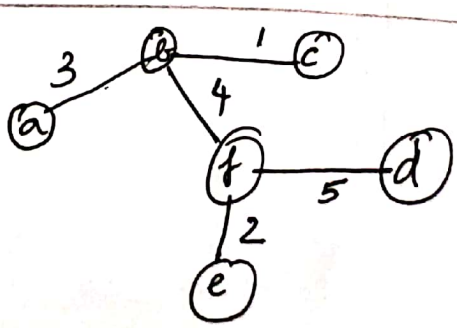
encounter $\leftarrow$ encounter $+ 1$

return E_T

Scanned with CamScanner

Ex:



Tree edges	Sorted list of edges	Illustration.
	bc ef ab bf cf af df 1 2 3 4 4 5 5 ae cd de 6 6 8	
bc 1	bc ef ab bf cf af 1 2 3 4 4 5 df ae cd de 5 6 6 8	
ef 2	bc ef ab bf cf af 1 2 3 4 4 5 df ae cd de 5 6 6 8	
ab 3	bc ef ab bf cf af 1 2 3 4 4 5 df ae cd de 5 6 6 8	
bf 4 df 5	bc ef ab bf cd af 1 2 3 4 4 5 df ae cd de 5 6 6 8	

Scanned with CamScanner

Disjoint subsets and union-Find Algorithms:

- **makeset (x)** – creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the element s of set S only once.

Ex: $S = \{1, 2, 3, 4, 5, 6\}$

makeset (i) creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$

- **find (x)**

returns a subset containing x.

- **union (x, y)**

constructs the union of the disjoint subsets S_x and S_y containing x and y, respectively and adds it to the collection to replace S_x and S_y which are deleted from it.

Ex: performing union (1, 4) and union (5, 2) yields

$\{1, 4\}, \{5, 2\}, \{3\}, \{6\}$

and union (4, 5) and by union (3, 6) is

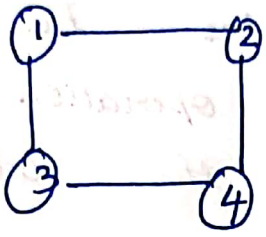
$\{1, 4, 5, 2\}$ $\{3, 6\}$

Scanned with CamScanner

- union is used to check whether there exists a cycle or not.

union (1,2) is not possible as it exists in the same set. {1, 4, 5, 2} and forms a cycle.

Ex: Check the graph:



- makeset (i) - It creates four singleton sets

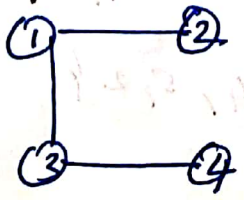
{1}, {2}, {3}, {4}

- find i - locates the elements
- union (1,2) - it is possible as they are 2 different sets:

{1, 2}, {3}, {4}

- union (3,4) - {1, 2}, {3, 4}
- union (1,3) - {1, 2, 3, 4}
- union (2,4) - Not possible as they are from same sets and hence forms cycle.

∴

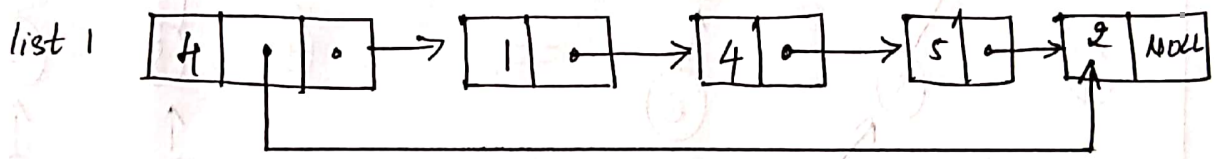


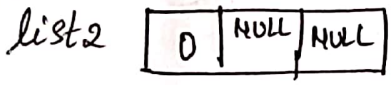
Scanned with CamScanner

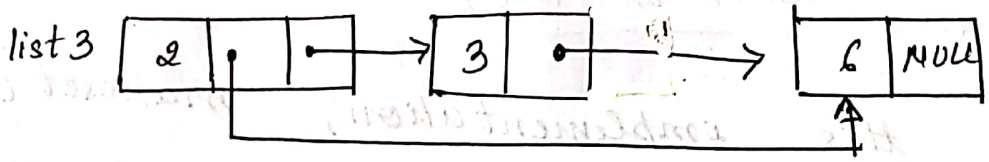
Implementation of find and union operations:

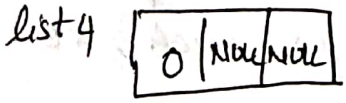
(i) **Quick find :**

- It uses array indexed by elements.
- Array's values indicate the representives of the subsets
- Each subset is implemented as linked list whose header contains the pointers to the first and last element along with number of elements in the list.









- Executing union(x, y) takes longer.
- Time efficiency to find is $O(n^2)$
- This can be overcome by using <u>union by size</u> i/e combining shorter list and perform find operation.

Scanned with CamScanner

- so the time efficiency for union by size is

$$O(n \log n + m)$$

for sequence of $(n-1)$ unions and m finds.

(ii) Quick union:

Second principal alternative for implementing disjoint subsets - are represent each subset by a rooted tree.

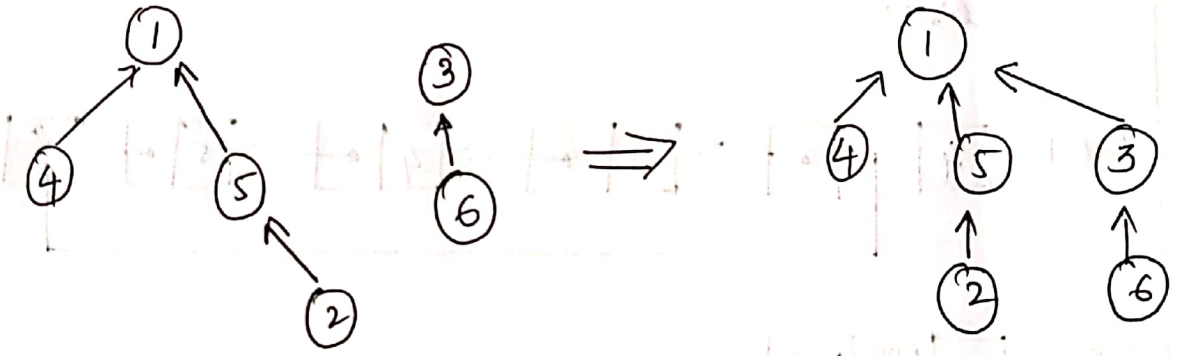


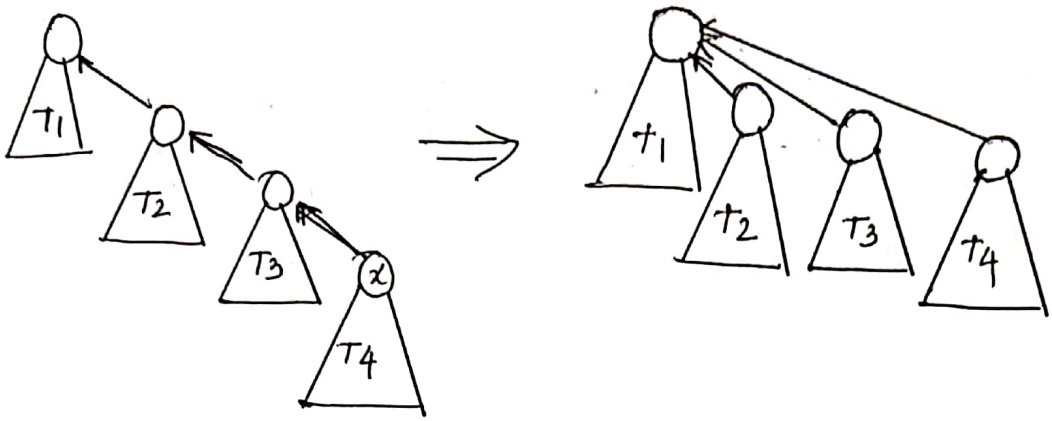
fig → union of (5, 6)

- for this implementation, makeset (x) requires the creation of a single-node tree, which is $\Theta(1)$ operation for each, so n singletone is $\Theta(n)$.

for union operation, root element must be chelked, if they are same, it forms cycle, if not it can be performed.

Scanned with CamScanner

Path compression:



- To improve time effeciency. path compression is used, as in union operation, each time it should refer root element. to check for the cycle.

To reduce the access time for root from node descendants, all the nodes point to the tree's root as shown in above figure.

Scanned with CamScanner

Dijkstra's Algorithm:

The single source shortest path problem states:

"Given a weighted, connected graph and a source vertex, the task is to find the shortest path from source to all other vertices in the given graph $G = (V, E)$".

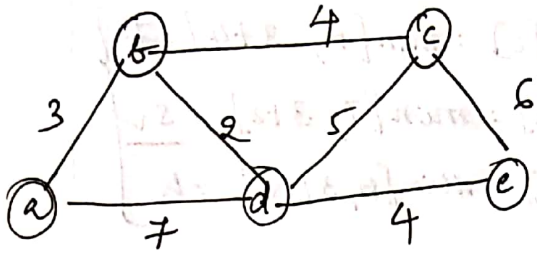
- Dijkstra's Algorithm solves SSSP (single source shortest path problem).
- First the algorithm finds the shortest path from the source to a vertex nearest to it, then to second nearest and so on.
- In general before its i^{th} iteration commences, the algorithm has already identified the shortest paths to $i-1$ other vertices nearest to the source

Scanned with CamScanner

```
Algorithm Dijkstra (n, adj, src, dest)
// computes shortest paths from source vertex
// Input : number of vertices n,
          adjacency matrix adj & source vertex
// output : The shortest paths from source
            vertex to all other vertices.
{
    for i=0 to n-1 do
    {
        dist[i] = adj[src][i];
        visited[i] = 0
    }

    visited[src] = 1

    for i=1 to n-1 do
    {
        find unvist and dist[unvist] such
        dist[unvist] is minimum
        and vertex unvist ∈ V-visted[]

        for every vertex ∈ V-visted[] do
        {

            if dist[unvist] + adj[unvist][vertex]
                                < dist[vertex] then
            {
                dist[vertex] = dist[unvist] +
                               adj[unvist][vertex]
}   }   }
```

Scanned with CamScanner

Problem: Consider the following graph & obtain the shortest path from the vertex "a" to all other vertices using Dijkstra's Algorithm.



adj[ ][ ] =

	a	b	c	d	e
a	0	3	∞	7	∞
b	3	0	4	2	∞
c	∞	4	0	5	6
d	7	2	5	0	4
e	∞	∞	6	4	0

- Initialize distance array ie dist[ ] from source "a".

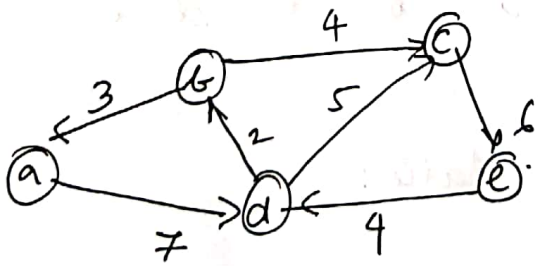
dist[ ] = [0 3 ∞ 7 ∞]
a b c d e

- Build the table:

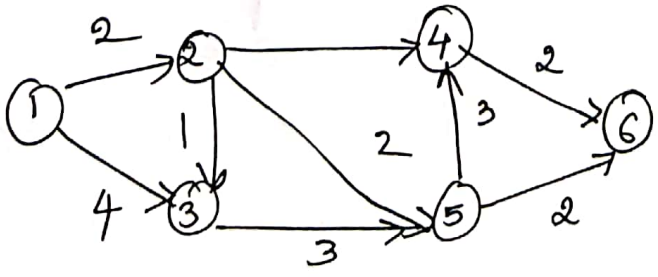
Scanned with CamScanner

visited	vertex = V-visted	dist[vertex] = min [dist[vertex], dist[unvist] + adj[unvist][vertex]]	Find unvist & dist[unvist] such that dist[unvist] is minimum & unvis ∈ V-visited.
a	b, c, de	—	b, 3
a, b	c, d, e	dist[c] = min[∞, 3+4] = 7 dist[d] = min[7, 3+2] = 5 ✓ dist[e] = min[∞, 3+∞] = ∞	d, 5
a, b, d	c, e	dist[c] = min[7, 5+5] = 7 ✓ dist[e] = min[∞, 5+4] = 9	c, 7
a, b, d, c	e	dist[e] = min[9, 7+6] = 9	e, 9
a, b, d, c e	—	—	—

Solve:



a as source



5 as source

Scanned with CamScanner

Huffman Trees:

There are mainly two types of compressing data namely

- fixed length – each character represents same number of bits.
  ex: a=00, b=01 c=11
- variable length
  ↳ characters having varying number of bits
  ex: a=0 b=01 c=111

- Huffman coding [David Huffman] is a popular method for compressing data with variable length codes. The idea behind this approach is:

  (i) It is prefix free

  (ii) least occurances characters have more bit representation compared to more occurances characters.

- Given a set of data symbols (an alphalets) & their frequencies of occurance. The huffman coding algorithm constructs a set of variable length codewords with the shortest average length & assigns them to symbols.

Scanned with CamScanner

- Huffman's Algorithm:-

step 1 - Initialize n one-node trees and label them with characters of the alphabet. Record the frequency of each character in its tree's root to indicate tree's weight.

Step 2 - Repeat the following operation until single tree is obtained.

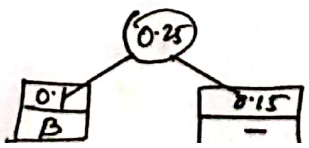
Find two trees with the small weight. Make them left and right subtree of new tree and record the sum of their weight in the root of the new tree as its weight.

P. Construct Huffman tree: for the following data:

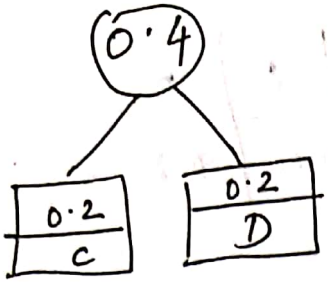
character	A	B	C	D	-
probability	0.35	0.1	0.2	0.2	0.15

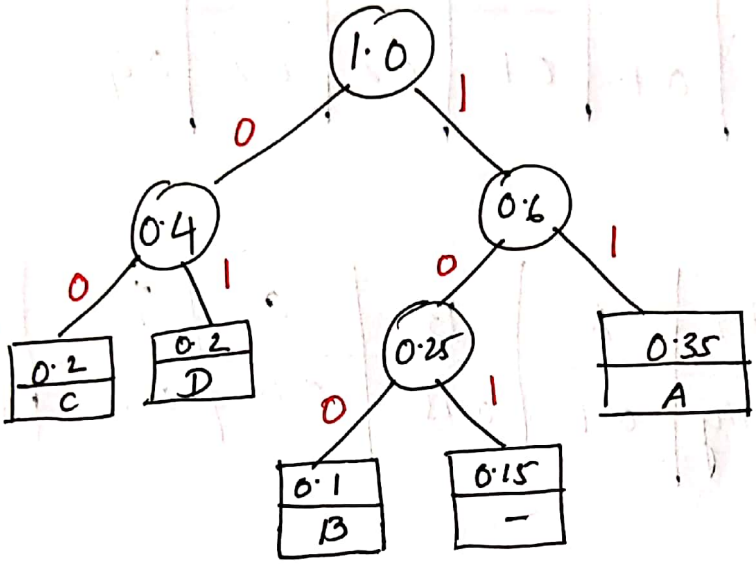
step 1 → Soot based on probaleility.

0.1	0.15	0.2	0.2	0.35
B	-	C	D	A



Scanned with CamScanner





C → 00 – 2 bits

D → 01 – 2 bits

A → 11 – 2 bits

B → 100 – 3

– → 101 – 3

with the occurence probabilities ; the expected number of bits are:

$$(A * 0.35) + (B * 0.1) + (C * 0.2) + (D * 0.2) + (- * 0.15)$$

$$= 2 \times 0.35 + 3 \times 0.1 + 2 \times 0.2 + 2 \times 0.2 + 3 \times 0.15 = 2.25$$

Scanned with CamScanner

Construct Huffman tree:

a)

Character	A	B	C	D	E	–
probability	0.5	0.35	0.5	1	0.4	0.2

b)

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

c)

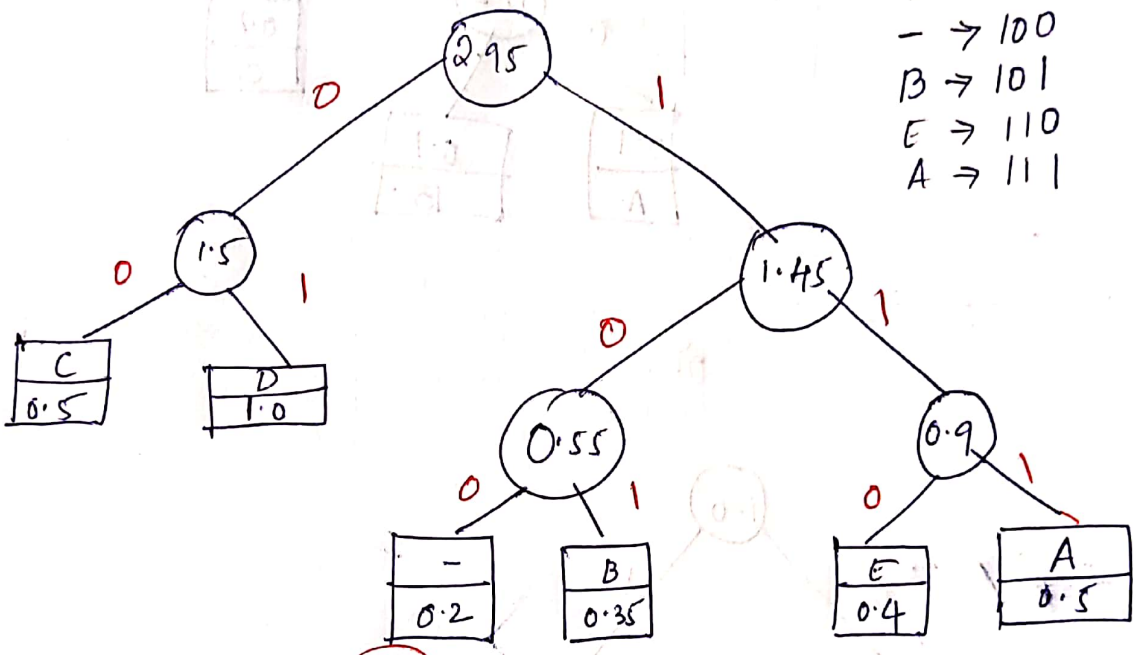
character	A	B	C	D	–
probability	0.1	0.15	0.15	0.2	0.4

Scanned with CamScanner

Sol^n →

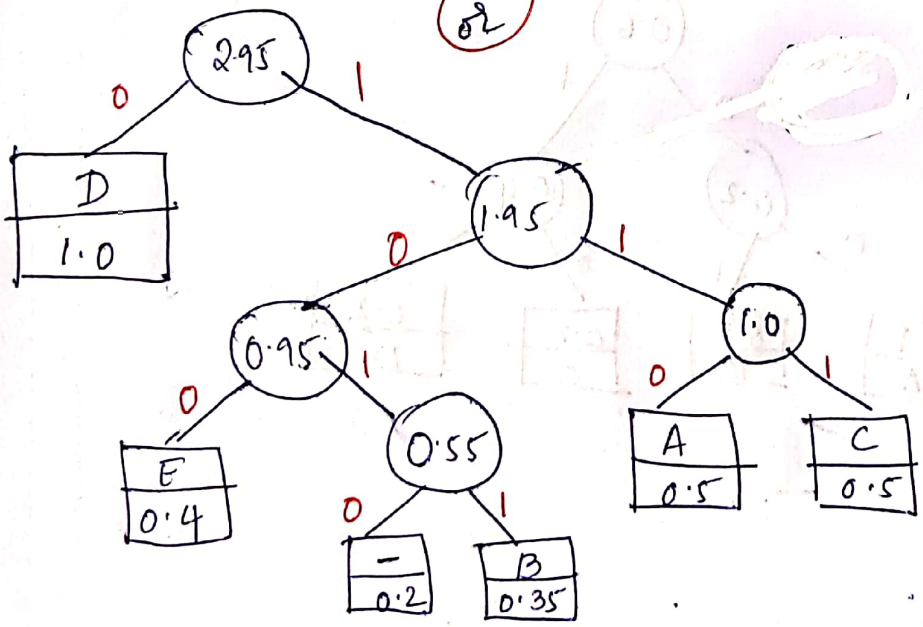
(a) sort

–	B	E	A	C	D
0·2	0·35	0·4	0·5	0·05	1·0



C → 00
D → 01
– → 100
B → 101
E → 110
A → 111

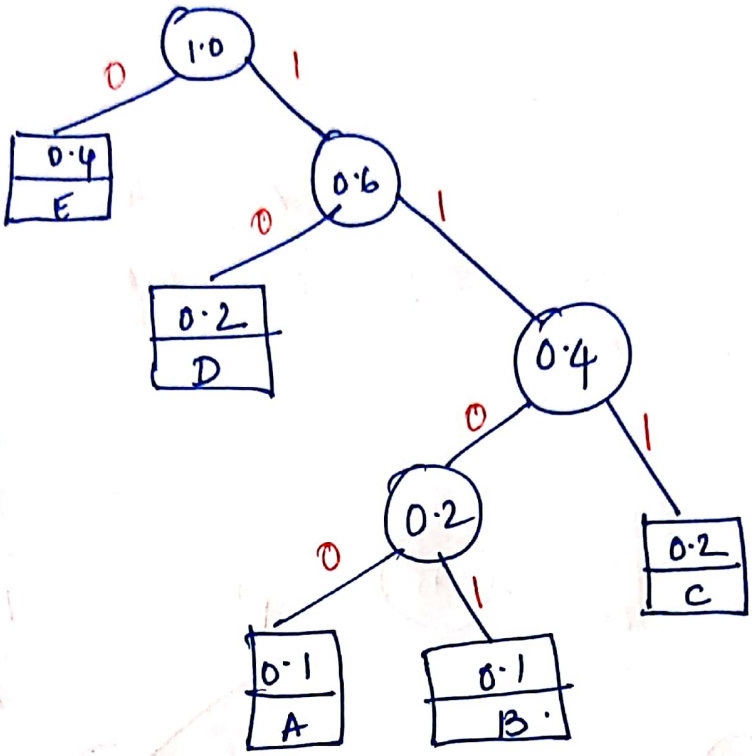
02



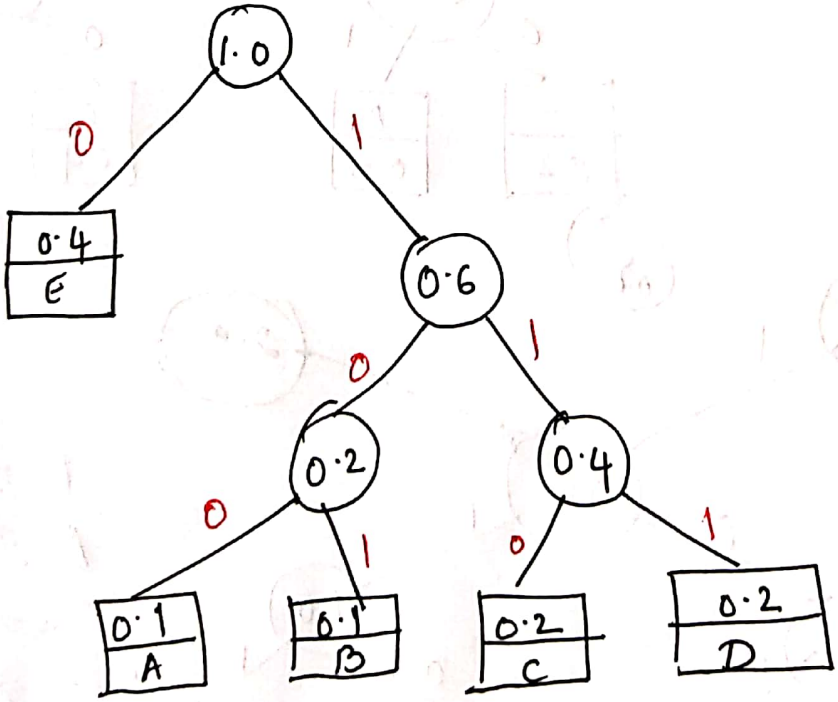
D → 0
A → 110
C → 111
E → 100
– → 1010
B → 1011

Scanned with CamScanner

(b)

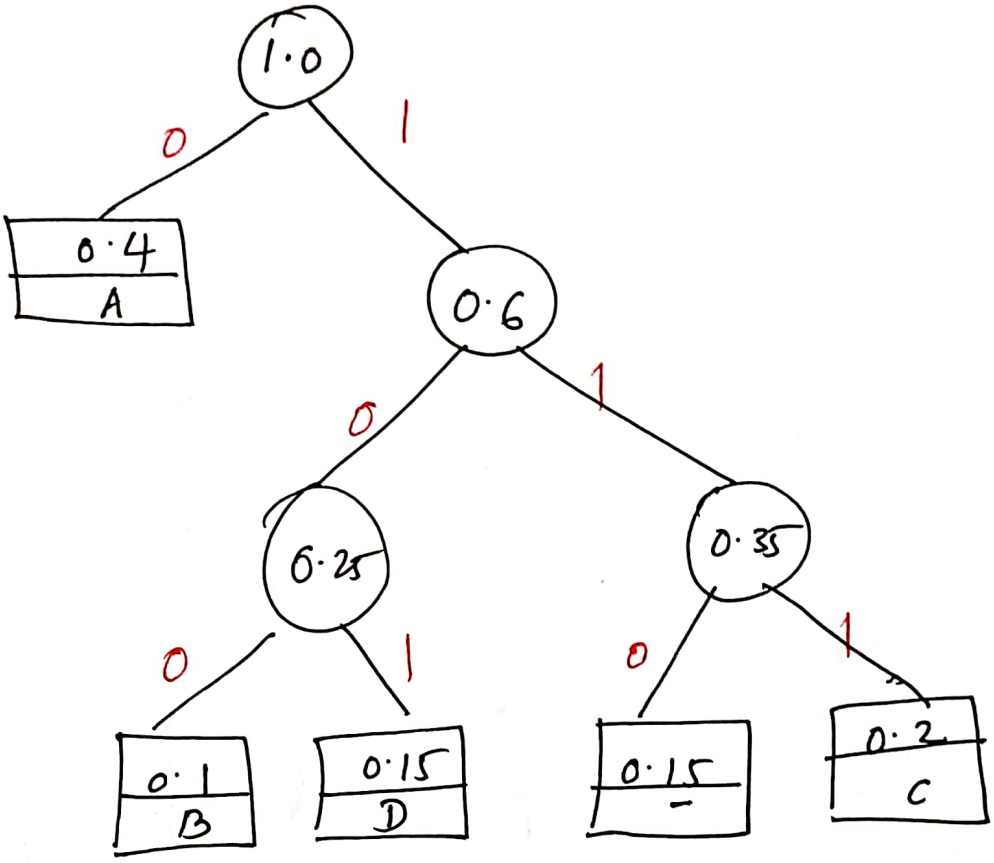


or



Scanned with CamScanner

(c)



Scanned with CamScanner