unit - 6.

# DECREASE-AND-CONQUER

As the name suggests, the decrease-and-conquer methodology suggests to decrease the size of the given problem. Here we try to find a relationship between the solution of original problem and that of smaller instance of the same problem. After finding this relationship, we will solve it either by top-down (recursive) approach or by bottom-up (iterative) approach.

In this technique, decreasing the size of the problem may be any one of these types_

* decrease by a constant
* decrease by a constant factor
* Variable size decrease.

1. Decrease by a constant :- In this technique, the size of an instance is reduced by the same constant on each iteration. Usually, this size we will be 1. Consider an example of finding the values of $a^n$. The relationship between a solution to a problem of size n and of size (n-1) is given by-

$$a^n = a^{n-1} \cdot a$$

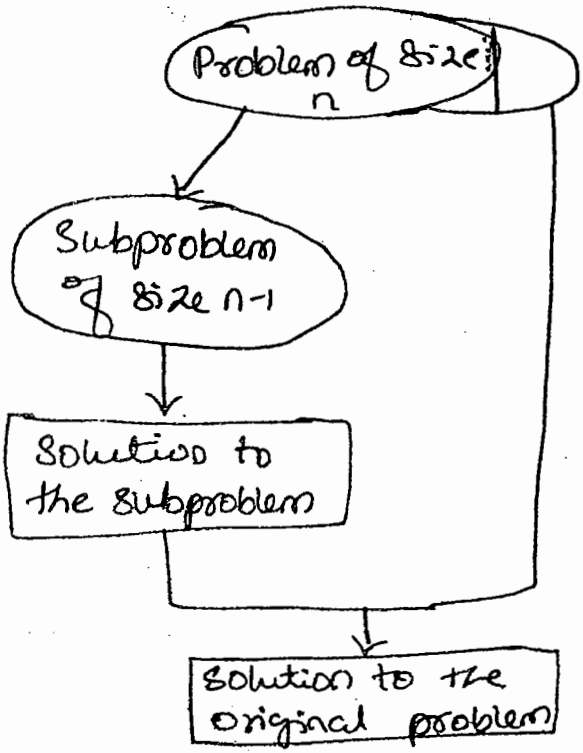
Thus, the function $t(n) = a^n$ can be solved by either top-down approach using

the relation -

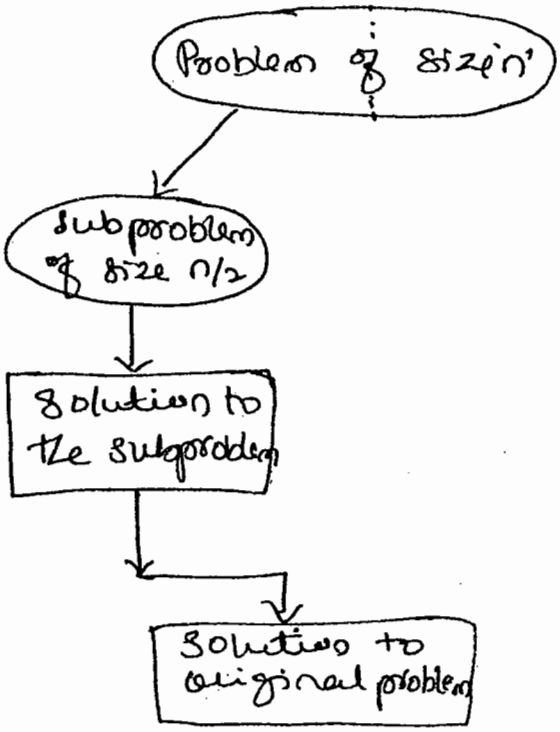
$$t(n) = \begin{cases} t(n-1) \cdot a & , \text{ if } n > 1 \\ a & , \text{ if } n = 1 \end{cases}$$

Or by ~~string~~ multiplying a by itself (n-1) times, which is bottom-up approach.

The pictorial representation of decrease-and-conquer by one is as below -




2. Decrease by a constant factor :- Here, the size of a problem is reduced by same constant factor on each iteration. Usually, this constant factor will be 2. This can be illustrated as —




For example, if we consider the same problem of finding $a^n$, it is given by -

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even \& +ve} \\ [a^{(n-1)/2}]^2 \cdot a & \text{if } n \text{ is odd \& } > 1. \\ a & \text{if } n = 1. \end{cases}$$

3. Decrease by a variable size :- Here, on each iteration, the size of original problem is reduced by different values.
For example, if we consider a problem of finding gcd of two numbers -

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Here, each time $n$ is replaced by $(m \bmod n)$ and obviously it is a varying factor on each step.

## Insertion Sort :-

In this sorting technique, we will compare nth element with all other previous elements and if that element finds its proper position, it is placed there and rest all the elements which are greater, ~~than~~ are pushed one position ahead. In this technique, we follow the idea that, at each step, the elements before nth element are sorted. So, each time, the array size is reduced by one.

The iterative algorithm is given as-

```
ALGORITHM InsertSort(A[0..n-1])
// Sorts given array by insertion sort
// Input: An array A[0..n-1] of n elements
// Output: A sorted array A[0..n-1].
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j >=0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

Analysis:

1. The parameter is input size n.
2. The basic operation is comparison.
3. The time complexity not only depends on the input size, but also on the fact that how the elements are arranged. So, let us discuss various complexities.

Worst case:

If the comparison (A[j] > v) executes for each value of i and j, then there will be maximum number of comparisons. This will happen if the given array is strictly decreasing. Thus, the worst case happens for an array sorted in decreasing order.

Thus,

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1-0+1) = \sum_{i=1}^{n-1} i$$

$$= 1 + 2 + \cdots + (n-1)$$

$$= \frac{(n-1)(n)}{2} \approx n^2$$

$\therefore C_{worst}(n) \in \Theta(n^2)$

Best Case:

Best case occurs if the given array is already sorted. For such case, for each values of i, there will be only one comparison. So,

$$C_{Best}(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1-1+1)$$

$$= n-1$$

$$\therefore C_{Best}(n) \in \Theta(n).$$

Average Case:

Suppose that, we are considering $i^{th}$ element of the array for its checking its proper position. Then $i$ will be at $(i-1)^{th}$ position. Now, there are several possibilities about the position of $i^{th}$ element.

~~If j-th element~~

If $A[j] < A[i]$, then the position of $A[i]$ is fixed, and only one comparison is made.

If $A[j] > A[i]$, but $A[j-1] < A[i]$, then only two comparisons happens.

Continuing in this ways, the possible number of comparisons for the $i^{th}$ element are

1, 2, 3, ..., i. Thus, the average number of comparison for each element is given by,

$$C_{avg}(n) = \sum_{i=1}^{n-1} \frac{1}{i} \cdot \sum_{j=1}^{i} j$$

$$= \sum_{i=1}^{n-1} \left\{ \frac{1}{i} \cdot (1+2+\cdots+i) \right\}$$

$$= \sum_{i=1}^{n-1} \left\{ \frac{1}{i} \cdot \frac{i(i+1)}{2} \right\}$$

$$= \sum_{i=1}^{n-1} \frac{i+1}{2}$$

$$= \frac{1}{2} \left\{ \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \right\}$$

$$= \frac{1}{2} \left\{ \frac{(n-1)n}{2} + (n-1) \right\}$$

$$= \frac{(n-1)}{2} \left\{ n/2 + 1 \right\}$$

$$= \frac{(n-1)(n+2)}{4}$$

$$\approx n^2/4$$

$\therefore C_{avg}(n) \in \Theta(n^2)$

## Depth First Search (DFS) :-

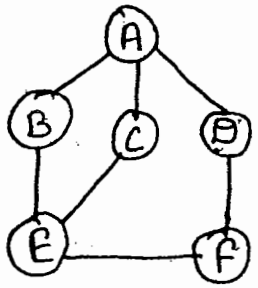
DFS is one of the techniques to process the vertices or edges of a graph in a systematic order. The procedure is as below-

1. Visit a vertex of a graph arbitrarily & mark it as being visited.
2. Visit the any unvisited vertex which is adjacent to the previous one.
3. Repeat the step ② until no adjacent-vertex is left out for a particular vertex.
4. Go back to one vertex before the dead-end vertex and try to find any unvisited vertex and repeat the process if so.
5. The algorithm halts when the starting vertex becomes the dead-end and by this time, all the vertices in that connected component of a graph are visited.
6. If the graph contains still unvisited vertices, reapply the above procedure by choosing any arbitrary vertex.

To implement the operations of DFS, we will use stack. When a vertex is reached for the first time, it is pushed into the stack. When it becomes the dead-end, we pop-up it.

Sometimes, the dfs forest is used for illustrating DFS traversal. In DFS forest, the starting vertex of DFS traversal acts as a root of first tree. Whenever a new unvisited node is encountered, it is put as a child to the vertex from which it is being searched. The edge connecting two such nodes is called as tree edge. In this process, we may get an edge from a particular node to a previously visited node, which is not its parent. Such an edge is called as back-edge.

Consider the following graph.



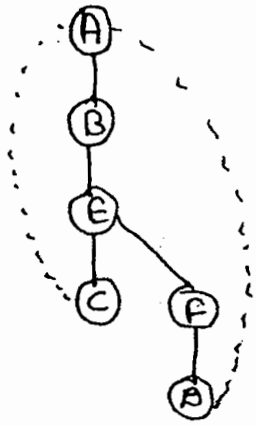

The stack representation of above graph is given below. Here, the first subscript indicates the order in which the node is visited and the second subscript indicates the order in which the node is deleted from the stack, i.e. when it becomes dead-end.

$A_{1,6}$ ~~$D_{5,3}$~~ $F_{5,3}$

$B_{2,5}$ ~~$F_{6,2}$~~ $D_{6,2}$

$E_{3,4}$

$C_{4,1}$




The DFS forest is also given here. Dotted lines shows back-edges.

ALGORITHM DFS(G)
// Implementation of DFS traversal of graph G.
// Input: Graph G = (V, E)
// Output: Graph G with its vertices marked
//  with consecutive integers in the
//  order they visited.

mark each vertex in V with 0 as of
being unvisited.
count ← 0
for each vertex v in V do
  if v is marked with 0
    dfs (v)

dfs (v)
// visits all unvisited vertices connected to v
// and assigns them the ~~ord~~ numbers in the
// order they visited using 'count'.
count ← count +1
mark v with count
for each vertex w in V adjacent to v do
  if w is marked with 0
    dfs (w)

We can create adjacency linked list and adjacency matrix for the given graph based on the DFS algorithm. The efficiency of DFS depends on the fact that which data structure (ie

adjacency matrix or adjacency linked list) we will use for its implementation. If we use adjacency matrix, then the time complexity of DFS is $\Theta(|V|^2)$ and if we use adjacency linked list, the complexity will by $\Theta(|V|+|E|)$. Here, $|V|$ and $|E|$ denotes number of vertices & edges of the graph, respectively.

Applications of DFS are as below-

* To check the connectivity of a graph: when DFS algorithm halts, if all the vertices have been visited, then the graph is connected, otherwise not.
* To check acyclicity of a graph: In a DFS forest, if any tree contains a back-edge, then the graph is cyclic, otherwise, acyclic.

(NOTE: A vertex of a connected graph is said to be articulation point of the graph if its removal with all edges incident to it breaks the graph into disjoint pieces.)

* To find articulation point of a graph.

## Breadth-first Search(BFS):-

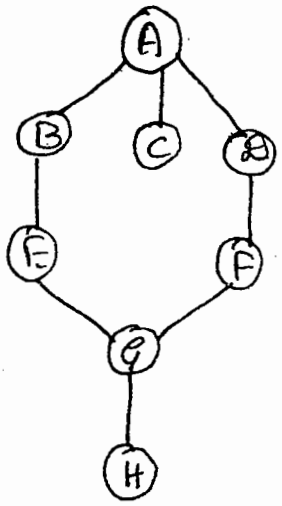
BFS is also a method for processing the vertices or edges of a graph in a systematic order. The procedure is as below-

1. Visit a vertex of a graph arbitrarily & mark it as visited.
2. All the vertices which are adjacent to previously visited vertex are visited sequentially.
3. Repeat the above step for next level and continue till all the vertices are visited in that connected component.
4. If some vertices are still remaining repeat the steps from step ①.

We will use the queue data structure to trace the operations of BFS. The first vertex of the traversal becomes the initial value of queue, i.e. the element at the front. On each iteration, the algorithm finds the vertices which are adjacent to the front vertex. These vertices are inserted into the queue and the front vertex is deleted from the queue.

As in DFS, for BFS also we will construct a BFS forest. The starting vertex of the traversal will be the root of one tree in BFS forest. Whenever a new node is visited, it must be attached to a previous node as a child. Such an edge is known as 'tree-edge'. Whenever a node finds an edge to already visited vertex other than its parent, then it is treated as 'cross-edge'.

Consider a graph-

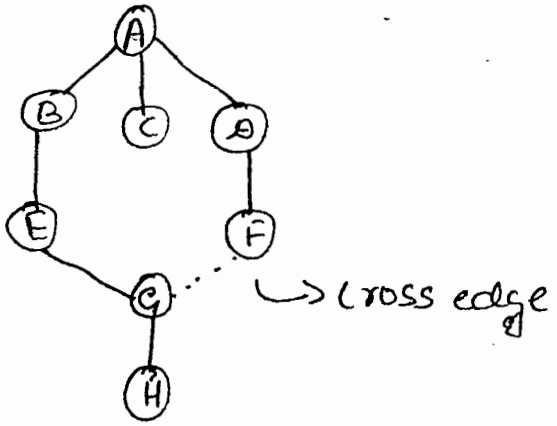



The queue sequence is given by-

$A_1, B_2, C_3, D_4, E_5, F_6, G_7, H_8$

The suffixes denote the order of their insertion into queue.

The BFS forest for this graph will be-



ALGORITHM BFS(G)
// Implementing BFS traversal for a graph G.
// Input: $G = (V, E)$.
// Output: Graph G with its vertices marked
//  with consecutive integers in the
//  order they visited.

mark each vertex in V with 0 as unvisited.
count ← 0
for each vertex v in V do
  if v is marked with 0
    bfs(v)

bfs(v)
// visits all unvisited vertices connected to v.
// and assigns them the numbers in the order
// of their visit.

count ← count + 1
mark v with count and initialize queue
  with v
while the queue is not empty do
  for each vertex w in V adjacent to
      front vertex v do
    if w is marked with 0
      count ← count + 1
      ~~add w to the~~
      mark w with count
      add w to the queue
  remove v from queue.

The analysis of BFS traversal depends on the data structure used for implementation. If we use adjacency matrix, then the time complexity will be $\theta(|V|^2)$ and if we use adjacency linked list, the time complexity will be $\theta(|V|+|E|)$.

Applications:

* Finding connectivity of a graph.
* Finding acyclicity of a graph.
* To find the shortest path between any two given vertices.

Comparison of DFS & BFS

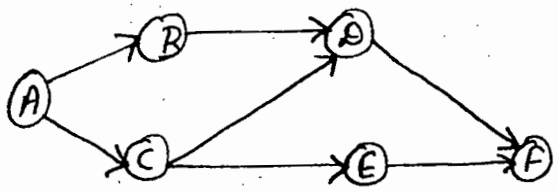
| | DFS | BFS |
|---|---|---|
| Data structure | Stack | Queue |
| No. of vertex orderings | two | one |
| Edge types | tree edge<br>back edge | tree edge<br>cross edge |
| Applications | Connectivity<br>Acyclicity<br>Articulation point | Connectivity<br>Acyclicity<br>Shortest path |
| Efficiency for Adj. Matrix | $\theta(\|V\|^2)$ | $\theta(\|V\|^2)$ |
| Efficiency for Adj. Linked list | $\theta(\|V\|+\|E\|)$ | $\theta(\|V\|+\|E\|)$ |

# Topological Sorting

Consider a problem of completing a project which can be divided into several parts. Usually, we have to estimate the time required for completing it. For doing that, obviously, we estimate the time required for each of the subparts keeping in mind that some of the subparts can be carried out parallelly, some of them can not be started until some others are completed etc. That means, there will be some pre conditions and constraints for carrying out these subparts. If we consider each subtask as a vertex, we can draw a directed graph where the direction of end edge indicates the order in which they must be completed. One thing we can note down here is, if such a digraph contains cycles, then such a problem can not be solved as it will be a deadlock. Thus, if the problem possesses a directed acyclic graph (dag), then we can solve it.

After creating a dag, our intension is to find a sequence of subtasks which says the order of their completion by satisfying all the good constraints & prerequisites. The problem of finding such a sequence is called as topological sorting.

For illustration, consider a project which is divided into 6 subtasks viz. A, B, C, D, E & F. The constraints are as below- To start B and C, A must be completed. D will be carried out only after B and C. E can be carried out after the completion of C. F can be done after D and E. The digraph can be given for this problem as-




It can be observed that the graph is acyclic. i.e. it is a dag. So, we can solve it using topological sorting technique.

There are two methods for solving topological problem.

One method is using DFS traversal. Here, we will perform a DFS traversal for the dag and

note down the order in which the elements becomes dead-ends. ie the order of popping up the vertices from traversal stack. The reverse of this order will give required solution. Thus, for the above problem, DFS traversal would be-

$A_{1,6}$ $B_{2,3}$ $D_{3,2}$ $F_{4,1}$ $C_{5,5}$ $E_{6,4}$

∴ The order of popping up from stack is.

F, D, B, E, C, A.

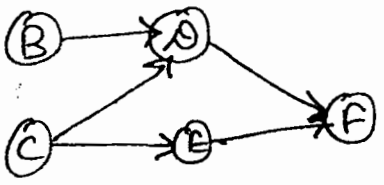
The required order will be-

A, C, E, B, D, F.

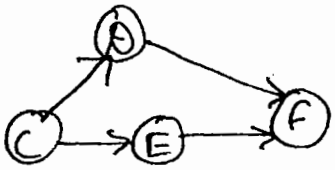
The second method is based on implementation of decrease-by-one-and-conquer strategy. Here, in each step, identify a vertex with no incoming edges. Then delete it with all outgoing edges from it. The order in which the vertices are deleted will be the required solution for a problem.

For the above example, first delete A.

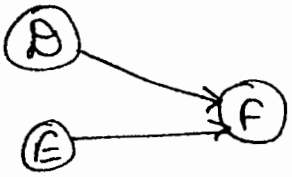
So,

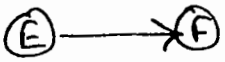


Then delete either B or C, arbitrarily.




Then, deleting C -




Now, deleting D - 


Finally, deleting E - (F).

So, required order is -

A B C D E F.

NOTE: The DFS method and DAC method may & yield different solutions. Because topological sorting problem will be having several solutions.

Following is the algorithm for topological sort using DFS technique -

ALGORITHM DFT (A[0..n-1][0..n-1], n)
// To obtain the sequence of jobs to be executed
// Input: Adjacency matrix A of the graph
// The number of vertices n.
// Output: Sequence in the reverse order that are
// to be executed.

```
for i ← 0 to n-1 do
    S[i] ← 0          // no vertex has been visited.
j ← 0                 // vertex index which is dead-end.
for v ← 0 to n-1 do
    if (S[v] = 0)
        DFS(a, n, v)
```

```
ALGORITHM DFS(a[0..n-1][0..n-1], n, v)
// Input : The vertex v from which traversal starts
S[v] ← 1
for u ← 0 to n-1 do
    if (cost[v][u] = 1 and S[u] = 0) then
        DFS(a, n, u)
j ← j + 1
res[j] ← v
```

Analysis:

In two functions, we have two loops. The time complexity depends on these. As there is one comparison for each loop, we have-

$$\sum_{v=0}^{n-1} \sum_{u=0}^{n-1} 1 = \sum_{v=0}^{n-1} (n-1-0+1)$$

$$= n \sum_{v=0}^{n-1} 1 = n^2$$

∴ Time complexity is $\theta(n^2)$, where $n = |V|$, i.e. the number of vertices in a graph.

Following is the algorithm for topological sort by source removal method.

```
ALGORITHM TS(A[0..n-1][0..n-1], n)
// To get a sequence of vertices to be executed.
// Input: Adjacency matrix A of the graph.
//        Number of vertices, n.
// Output: The sequence of vertices.
for j ← 0 to n-1 do
    sum ← 0
    for i ← 0 to n-1 do
        sum ← sum + A[i][j]
    indegree[j] ← sum
for i ← 0 to n-1 do
    if (indegree[i] = 0)
        top ← top+1
        S[top] ← i
while (top ≠ 1)
    u ← S[top]
    top ← top-1
    Add u to solution vector T
    for each vertex v adjacent to u
        decrement indegree[v] by one
        if (indegree[v] = 0)
            top ← top+1
            S[top] ← v.
```

# Generating Combinatorial Objects

The combinatorial objects like permutations, subset of a set etc. plays a vital role in the field of computer science. Here, we will study few of the algorithms for their generation.

## Generating Permutation :-

As we know, if there are n different items, we can arrange them in n! several ways. That is, we can find n! permutations for n elements.

Now, our problem is to generate these n! permutations. For the sake of simplicity, let us consider the numbers 1, 2, ..., n. We have to find all n! permutations of this set. The decrease-by-one technique says ~~that~~ to generate (n-1)! permutations first, and then to place nth number in n different positions of all of these permutations. ~~The process~~ Now we get $n \times (n-1)! = n!$ permutations.

The process can be applied recursively to get all permutations.

For illustration, consider the generation of permutations of {1, 2, 3, 4}.

start : 1

insert 2 : 12 21

insert 3 : 123 132 312 (right to left) 321 231 213 (left to right)

insert 4 :

| | | | | |
|---|---|---|---|---|
| 123 4 | 12 4 3 | 1 4 23 | 4 123 | right to left |
| 132 4 | 13 4 2 | 1 4 32 | 4 132 | |
| 312 4 | 31 4 2 | 3 4 12 | 4 312 | |
| 4 321 | 3 4 21 | 32 4 1 | 321 4 | left to right |
| 4 231 | 2 4 31 | 23 4 1 | 231 4 | |
| 4 213 | 2 4 13 | 21 4 3 | 213 4 | |

The above method is known as minimal-change technique. Because, each number can be generated by exchanging only two digits of its previous number.

There is one more method called Johnson-Trotter algorithm. In this method, we first take a permutation and associate an arrow mask for each digit 'k' of the permutation, to indicate direction. The digit-

or component k is said to be 'mobile' in arrow-marked permutation if its arrow points to a smaller number adjacent to it. For example—

$\vec{3}\ \overleftarrow{2}\ \vec{4}\ \overleftarrow{1}$.

Here 3 and 4 are mobiles.

Using this notation, the Johnson-Trotter algorithm is implemented as below—

ALGORITHM JT(n)

// Johnson-Trotter algorithm for generating
// permutations.

// Input : A positive integer n.

//Op: A list of all permutations of $\{1, \ldots, n\}$

Initialize the first permutation with
$\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$

While there exists a mobile integer k do
    find the largest mobile integer k.
    Swap k and the adjacent digit towards which k is having arrow.
    reverse the direction of all integers that are larger than k.

Consider an example of generating permutations of {1, 2, 3}.
Initially, we write $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$.
Here, $\overleftarrow{3}$ is largest mobile integer. So, next term will be $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$. Continuously applying the J-T algorithm, we get

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$ $\quad$ $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$ $\quad$ $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$

$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$ $\quad$ $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$ $\quad$ $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

Here, one thing we have to note that, the order of the numbers generated by J-T algorithm is quite unobvious. Because usually, permutations are written by increasing order i.e. for example.
123, 132, 213, 231, 312, 321
This method is known as <u>lexicographic Order</u>.
The method is as explained below— Suppose we want to generate permutations of $a_1 a_2 \dots a_{n-1} a_n$. Then, if $a_{n-1} < a_n$, the next number is obtained by just interchanging $a_{n-1}$ and $a_n$. If $a_{n-1} > a_n$, we will consider $a_{n-2}$. If $a_{n-2} < a_{n-1}$ the last three elements i.e $a_{n-2}$, $a_{n-1}$ and $a_n$ are rearranged in lexicographic order only & the process is continued.

## Generating Subsets :-

In many problems, we need to generate all the subsets of a given set for finding the solution to the problem. Knapsack is one such example.

The decrease-by-one-and-conquer method for finding a power set of a set $\{a_1, \ldots, a_n\}$ suggests to group the sets of power set in to two parts viz. those containing $a_n$ and those not containing $a_n$. Then, the second group will obviously be the subsets of $\{a_1, \ldots, a_{n-1}\}$ and the first group can be obtained by placing $a_n$ in all the subsets of group $\{a_1, \ldots, a_{n-1}\}$ Now, the procedure is reapplied on $\{a_1, \ldots, a_{n-1}\}$.

The following example illustrates this method.

| n | Subsets |
|---|---|
| 0 | $\phi$ |
| 1 | $\phi$ $\{a_1\}$ |
| 2 | $\phi$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ |
| 3 | $\phi$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$, $\{a_3\}$, $\{a_1, a_3\}$, $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$ |

One more method for generating the power set of a set is by establishing a one-to-one relationship between the set $\{a_1, \ldots, a_n\}$ and all $2^n$ bit strings of length $n$. For example, if we take $n = 3$ then.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| bit strings: | 000 | 001 | 010 | 100 | 101 | 110 | 111 |
| Subsets : | $\phi$ | $\{a_3\}$ | $\{a_2\}$ | $\{a_1\}$ | $\{a_1, a_3\}$ | $\{a_1, a_2\}$ | $\{a_1, a_2, a_3\}$ |

unit- 9

# GREEDY TECHNIQUE (9)

This technique is used for designing optimiza-tion problems. Greedy strategy always tries to find the best solution for each sub problem with the hope that this will results in the best solution for a whole problem. That is, greedy approach suggests to construct a solution for a problem through a sequence of steps, each step expanding a partially obtained solution so far, until a complete solution is reached. It is thus important to note that we have to make a choice in each step such that the choice must be-

* feasible - it has to satisfy the constraints of the problem
* locally optimal - it has to be the best local choice among all feasible solutions available at that step.
* irrevocable - Once a choice is made, it can't be changed on subsequent steps of the algorithm.

Chetana Hegde
9448301894

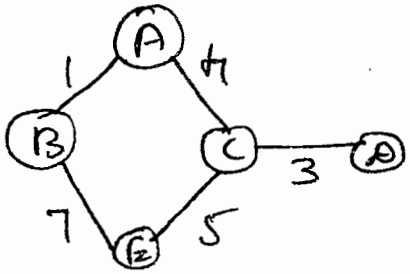
As in every step we look at optimal solution, greedy technique yields best solutions for many of the problems. But, the drawback here is, without aiming at final solution, we will think about the step where we are at. This is just like grabing whatever is available right now without thinking about the future. Thus, for some problems, if we use greedy, then we have satisfy ourselves with only approximate solution.

Consider an example of greedy:
Suppose we have a set of coins as— {1, 1, 1, 1, 1, 5, 5, 5, 10, 10, 25, 25}. We have to give change for 43 paise. Then we will first take the heighest possible value which is less than or equal to 43. i.e 25. For the next choice, we take next-heghest- ie 10. Continuing in this way we get the solution as {25, 10, 5, 1, 1, 1}
Note that, we won't start with 1 here.
The technique used here is nothing but-greedy.

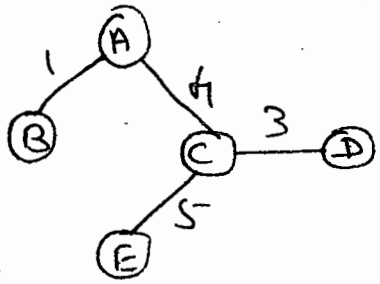
NOTE: Greedy technique is applied in finding minimum spanning tree using the algorithms like Prim's and Kruskal's algorithms.

Def: A connected acyclic subgraph containing all the vertices of a connected graph is known as spanning tree. The minimum spanning tree is defined for weighted connected graph and is the spanning subtree with minimum weight.

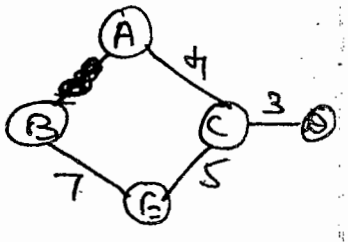
Consider a graph and its spanning trees -




Graph




$W(T_1) = 13$




$W(T_2) = 19$

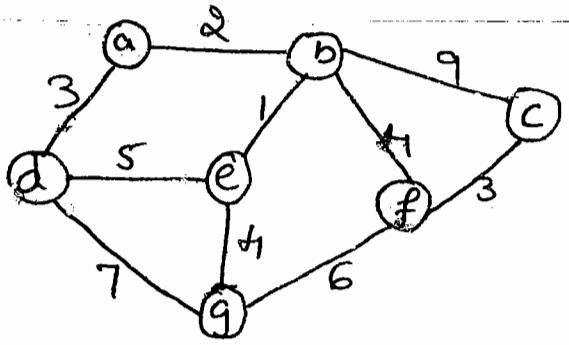
Here, $T_1$ will be the minimum spanning tree as the weight is less compared to that of $T_2$. Here, we have used the exhaustive-search technique. That is, first finding all possible spanning trees and then to select the minimum among all these. This method will be time consuming.

# Prim's Algorithm :-

This algorithm constructs minimum spanning tree through a sequence of expanding subtrees. The initial subtree is taken as any arbitrary vertex. Then among the other vertices of a graph, the nearest vertex to the existing vertex of the tree is found and is attached to the tree. The procedure is continued till all the vertices are included into the tree. Thus, if V is the set of all vertices of a graph and $V_T$ is that of minimum spanning tree then we have do the following operations -

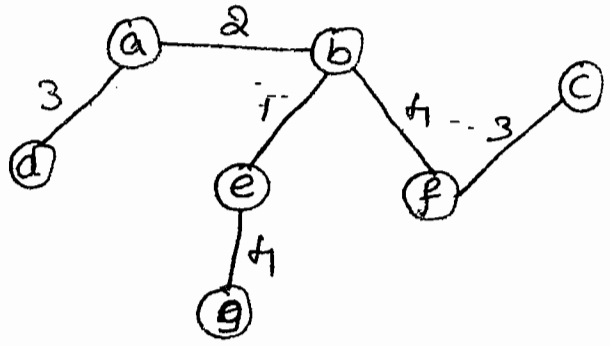
1. Move $u^*$ from the set $V - V_T$ to $V_T$.
2. For each remaining vertex u in $V - V_T$ that is connected to $u^*$ by a shorter edge than the u's current distance label, update its labels by $u^*$ and the weight of the edge between $u^*$ and u respectively.

For illustration, consider a connected weighted graph given below —




| Tree vertices | Remaining vertices |
|---|---|
| a(–, –) | b(a, 2), d(a, 3), c(–, ∞), e(–, ∞), f(–, ∞), g(–, ∞) |
| b(a, 2) | d(a, 3), e(b, 1), c(b, 9), f(b, 4), g(–, ∞) |
| e(b, 1) | d(a, 3), c(b, 9), f(b, 4), g(e, 4) |
| d(a, 3) | c(b, 9), f(b, 4), g(e, 4) |
| f(b, 4) | c(b, 9), g(e, 4), c(f, 3) |
| c(f, 3) | g(e, 4) |
| g(e, 4). | |

Thus the minimum spanning tree is –




chetana hegde
9448301894

ALGORITHM Prim(G)

// Prim's algorithm to construct min. spa. tree.

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

// Output: $E_T$, the set of edges composing

// 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 $v \in V_T$ and $u \in V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return $E_T$.

Analysis:

Note that, in every step of the algorithm, the set of remaining vertices with distances constitute a priority queue. Each time a choice is made, the queue must be updated. Thus, the efficiency of the Prim's algorithm depends on the data structure

selected for the graph and for the priority queue. If we represent a graph by its weight matrix and the priority queue by unordered array. Then the time complexity is of order $\Theta(|V|^2)$.

On the other hand, we can implement priority queue as min-heap, whose efficiency for insertion and deletion is of $O(\log n)$, where $n$ is size of heap. Suppose that the graph is implemented using adjacency linked list. In this case, as the algorithm performs $|V|-1$ deletions and $|E|$ verifications in the time $O(\log n)$, we have

complexity $= (|V|-1+|E|) \cdot O(\log |V|)$

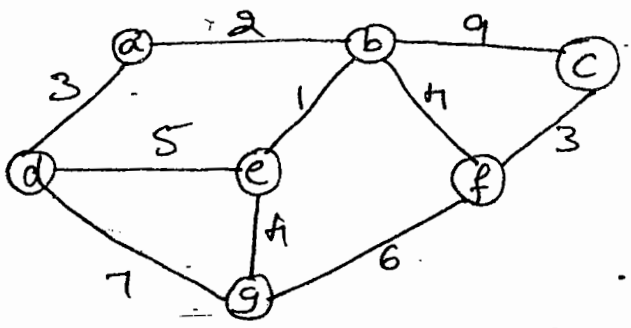
As $|V|-1 \leq |E|$ for connected graph, we have

$C \in O(|E| \cdot \log(|V|))$.

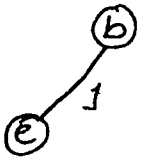
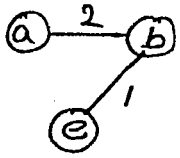
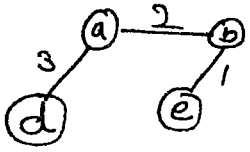
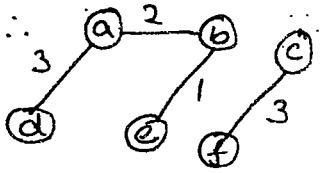
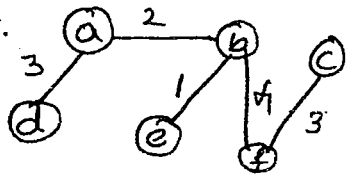
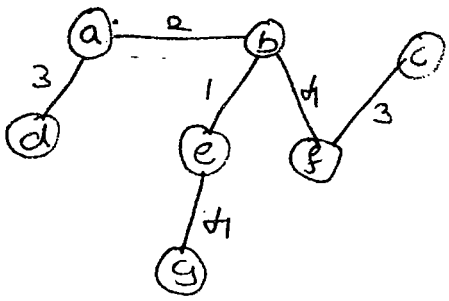
## Kruskal's Algorithm:-

This algorithm is also used for finding minimum spanning tree of a connected weighted graph. Here, we will first sort the edges of the graph based on their weights. The edge with minimum weight is added to the empty tree. Then, among the remaining list, next smallest edge is considered and is added to the existing tree only if the addition will not create a cycle. If the addition of any edge to the existing tree results in a cycle, such an edge is just ignored. The procedure is continued till the tree contains all the vertices of a graph.

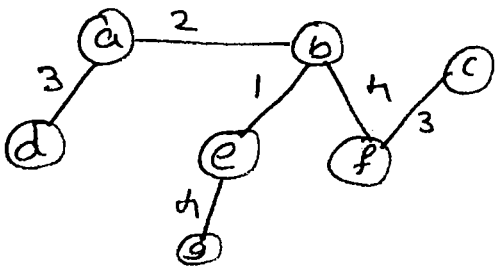
Consider the following graph -




To find the minimum spanning tree, we will proceed as below -

| Tree edges | Sorted list of edges |
|---|---|
| | be ab ad cf bf eg de gf dg bc<br>1 2 3 3 4 4 5 6 7 9 |
| be | be is inserted. So,  |
| ab | Next smallest is ab. So,  |
| ad | Now, ad. ∴  |
| cf | Next smallest is cf. ∴  |
| bf | Now, bf. ∴  |
| eg | Again, eg ⇒  |

Next smallest is de. But this will create a cycle. Similarly, gf, dg, and bc will create cycles. So, we will ignore them.

Thus, required tree is - 

ALGORITHM Kruskal (G)

// For constructing MST by Kruskal's algo.

// Input : Weighted connected graph $G(V, E)$.

// Output : $E_T$, the set of edges which is MST.

Sort E in non-decreasing order of edge weights $w(e_1) \leq \dots \leq w(e_{|E|})$.

$E_T \leftarrow \phi$

edgecount $\leftarrow 0$

$k \leftarrow 0$

While edgecount $< |V| - 1$ do

$k \leftarrow k + 1$

if $E_T \cup \{e_i\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_i\}$

edgecount $\leftarrow$ edgecount $+ 1$

Return $E_T$.

NOTE : 1. Even though Kruskal's algorithm seems to be simpler than Prim's algorithm, it is not the fact. Because, before adding every edge to the MST, we have to check whether it results in a cycle or not in Kruskal's algorithm. That is, we have to compare the newly considered selected edge with already selected edges before inserting it in each step.

2. Unlike in Prim's algorithm, the intermediate steps of Kruskal's algorithm may be consists of disconnected edges. At the end only we will get MST here.

3. As Kruskal's algorithm involves sorting the edges, if we use efficient sorting technique to do so, the efficiency of the algorithm will be in $O(|E| \log |E|)$.

## Dijkstra's Algorithm

We know that Floyd's algorithm is used to find all-pairs shortest path in a given weighted connected graph. That is, we were interested in a path that starts at any arbitrary vertex and visits all the vertices of a graph within minimum cost. On the other hand, some times, we may be interested in finding a shortest paths from a particular vertex to all other vertices. Such a problem is known as single-source shortest-paths problem. To solve this, Dijkstra's algorithm is used. Here, for a given vertex say, source, we will find shortest path

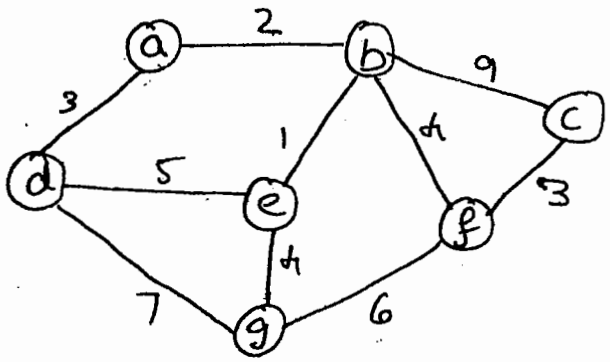
to all other vertices.

In this algorithm, given a source, first find the vertex which is nearer to it. Then find the second nearest & so on. Thus, at the $i^{th}$ iteration, we will find the shortest distance between the source and the $i^{th}$ vertex. At every step, we are going to get a tree of already considered vertices. The set of vertices adjacent to the vertices in $T_i$ is referred to as 'fringe set'. After finding the nearest vertex $u^*$ to the vertex in $T_i$, we have follow the following steps.

* Move $u^*$ from the fringe set to $T_i$.
* For each remaining vertex $u$ in fringe set that is connected to $u^*$ by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of $u$ by $u^*$ and $d_{u^*} + w(u^*, u)$ respectively.

Consider a graph to illustrate Dijkstra's algorithm -

Chetana Hegde
9448301894




Suppose 'a' is given as source vertex. Now we have to find shortest distance of other vertices from a.

| Tree vertices | Fringe set |
|---|---|
| a(-, 0) | b(a, 2), c(-, ∞), d(a, 3), e(-, ∞), f(-, ∞), g(-, ∞) |
| b(a, 2) | d(a, 3), c(b, 2+9), e(b, 2+1), f(b, 2+4), ~~g(b, 2+1+4)~~ g(-, ∞) |
| d(a, 3) | c(b, 2+9), e(b, 2+1), f(b, 2+4), g(d, 3+7) |
| e(b, 3) | c(b, 2+9), f(b, 2+4), g(e, 2+1+4) |
| f(b, 6) | c(f, 2+4+3), g(e, 2+1+4) |
| g(e, 7) | c(f, 2+4+3) |
| c(f, 9) | |

Thus, the shortest paths set is given as –

From a to b : $a \rightarrow b$, Dist = 2

a to c : $a \rightarrow b \rightarrow f \rightarrow c$, Dist = 9

a to d : $a \rightarrow d$, Dist = 3

a to e : $a \rightarrow b \rightarrow e$, dist = 3

a to f : $a \rightarrow b \rightarrow f$, dist = 6

a to g : $a \rightarrow b \rightarrow e \rightarrow g$, dist = 7

```
ALGORITHM Dijkstra(G, s)
// Dijkastra's algorithm for single-source-shortest paths
// Input: A weighted connected graph G=(V, E)
//        and its vertex s, which is source.
// Output: The length d_v of a shortest path from
//         s to v and its penaltimate vertex
//         p_v for every vertex v in V.
Initialize(Q)  // An empty vertex priority queue is
               // initialized.
for every vertex v in V do
    d_v ← ∞
    p_v ← null
    Insert(Q, v, d_v)
d_s ← 0
Decrease(Q, s, d_s)  // Update priority of s with d_s
```

$V_T \leftarrow 0$

for $i \leftarrow 0$ to $|V|-1$ do

$u^* \leftarrow$ DeleteMin($Q$) //delete min. priority element

$V_T \leftarrow V_T \cup \{u^*\}$

for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do

if $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$

$P_u \leftarrow u^*$

Decrease ($Q$, $u$, $d_u$)

Analysis:

The efficiency of Dijkastra's algorithm depends on the data structure used for implementing priority queue and the input graph.

If the priority queue is represented using an array and the graph is represented using weight matrix, then the time complexity of the algorithm will be $\Theta(|V|^2)$.

If the priority queue is represented as a min-heap and the input graph is represented using adjacency linked list, then the time complexity will be. $O(|E| \log |V|)$.

# HUFFMAN TREES

In the field of computer science, encoding and decoding a particular text / information is an important aspect, to save the space. Usually, encoding is done by assigning some sequence of bits called 'codeword' to each character of the text. There are mainly two methods of encoding viz. fixed-length encoding and variable-length encoding.

Fixed-length encoding assigns to each character a bit string of same length. If the text to be encoded has 'n' characters, then the number of bits required to encode that text is greater than or equal to $\log_2 n$. For example, if a text contains 8 characters, $\log_2 8 = 3$ bits are sufficient which we can arrange like 000, 001, 010, 011, 100, 101, 110 and 111.

More specifically, we can say that for a text of n characters, the bits required are $\lceil \log_2 n \rceil$.

Variable-length encoding assigns codewords of different lengths to different characters. ~~If a particular~~ Shorter sequence of bits are assigned to frequently occuring characters and longer sequence of bits are assigned to less frequent characters. But, in this method, we will unable to tell how many bits of an encoded text represent an $i^{th}$ character of a text?

To avoid this problem, a method of prefix-free or prefix codes is used. Here, no codeword is a prefix of another codeword. To create a binary prefix code for some ~~alphabet~~ text, associate each character with leaves of binary tree, in which all left edges ar labeled by 0 and right edges are labeled by 1. Now, the sequence of labels found in a simple path from root to any leaf will give the codeword for a character which is at that leaf. Since, there will not be a simple path from one leaf to other leaf, no code can be prefix of other.

Huffman algorithm is used to construct such a tree which will give shorter code for frequent characters and longer code for less frequent characters.

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

step 2: Find two trees with smallest weights. Make them left and right subtree of a new tree. Put the sum of their weights as a root of this new tree.
Repeat step ② till a single tree is obtained.

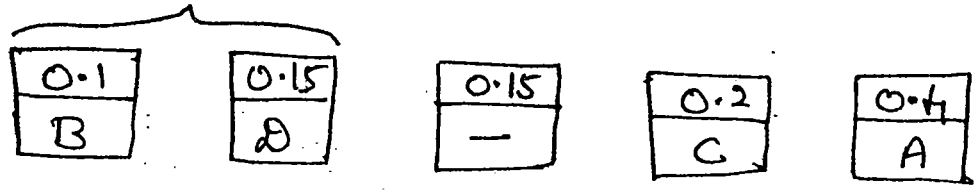
The tree constructed using above algorithm is Huffman tree and the codewords we get is Huffman code.

Ex: Construct a Huffman code for the following data:

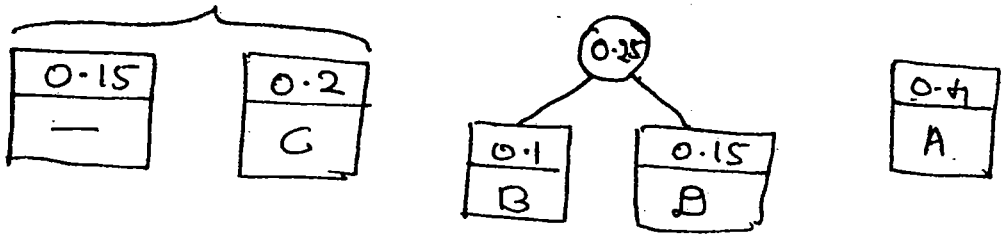
| Character : | A | B | C | D | _ |
|---|---|---|---|---|---|
| Probability : | 0.4 | 0.1 | 0.2 | 0.15 | 0.15 |

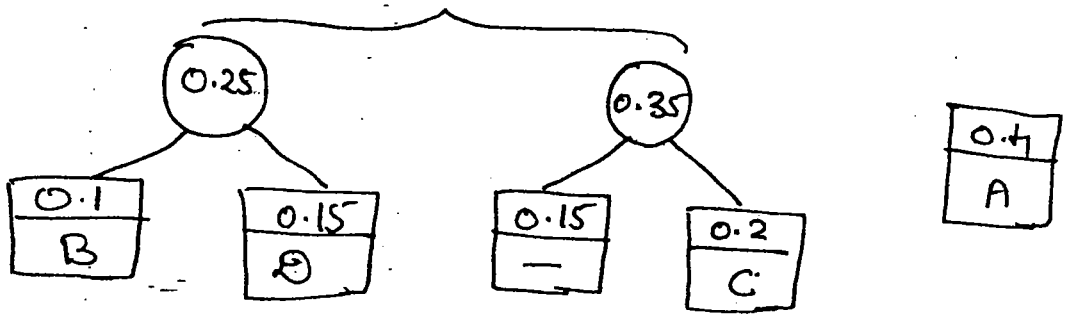
(i) Encode the text ABACABAD using the code.
(ii) Decode the code 100010111001010.

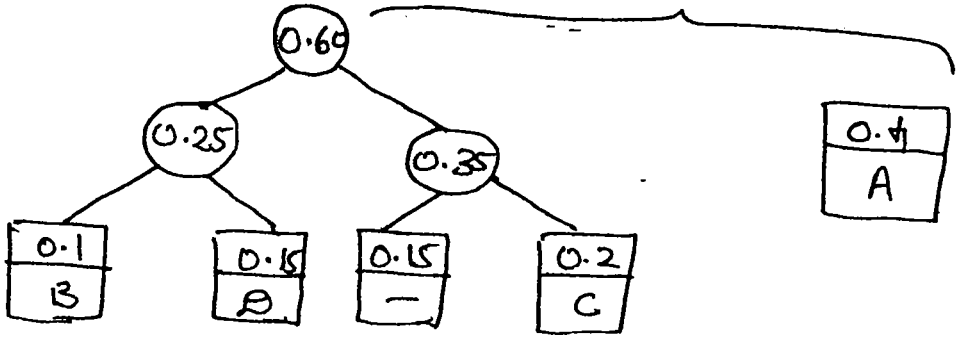
Sol²: In the first step, we create 5 one-node trees as below— (in ascending order of weights)

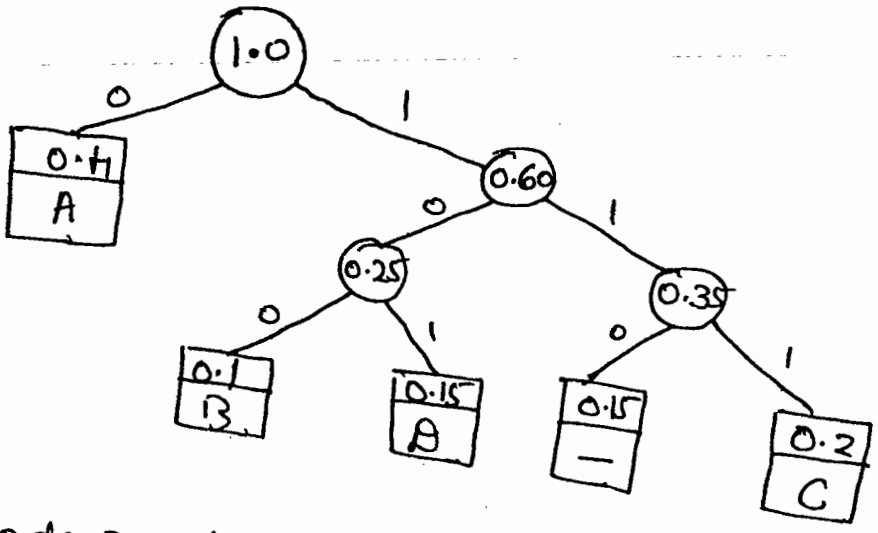



Combining smallest weighted trees—






Now codewords are –

A : 0

B : 100

C : 111

Ø : 101

— : 110

Chetana Hegde
9449830189

So, ABACABAØ can be encoded as –

0 100 0 111 0 100 0 101

And, 100 0 101 110 0 101 0 can be decoded as –

B A Ø — A Ø A