# Algorithms

→ Set of rules or instructions to perform some calculations either by hand or more usually on a machine (computer)

→ Algorithms are a finite set of instructions that accomplishes a particular task.

## Algorithms must satisfy:-

① Input: Quantity given to algorithm.

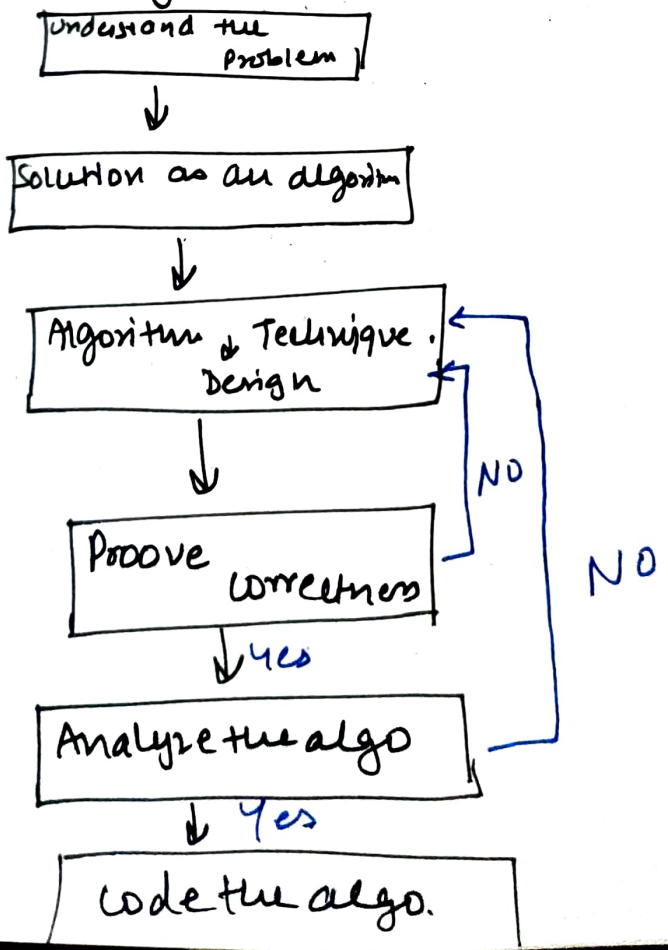
② Output: Quantity produced by an algorithm.

③ Definiteness: Each instruction/step must be easy to understand

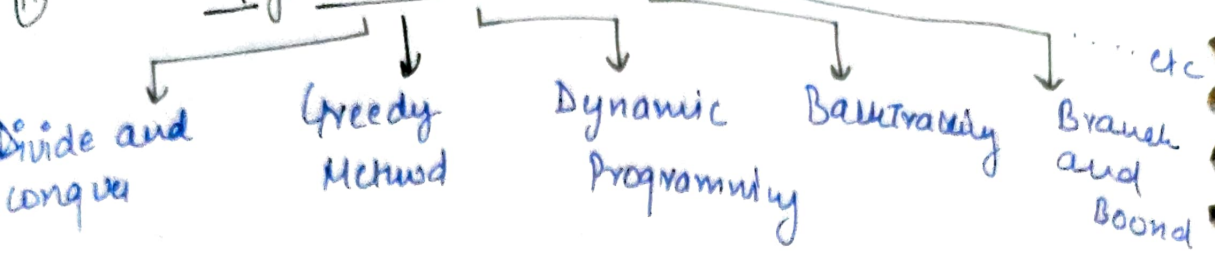
④ Effectiveness: Each instruction must be very basic i.e it can be carried out by a person on pencil and paper

⑤ Finiteness: Algorithms must be of finite number of steps and must be terminated after performing the finite no. of steps.

## 2) Process for Design and Analysis of Algorithms



(ii) Algorithms Technique




(iii) Types of Algorithms :-

(1) Approximate algo: → If algo is infinite and repeating

(2) Probabilistic Algo: If solution is uncertain

(3) Infinite Algo: Which is not finite

(4) Heuristic Algo: Giving fewer inputs and more outputs.

Pseudo Code:- An intemediate step/ procedure between the English description of Algorithm and programming Implementation of the same.

Performance Analysis of Algorithm:-

→ It refers to the task of determining the efficiency of an algorithm. ie how much computing time and storage of an algorithm requires to run or execute This analysis helps us to judge one algorithm over another.

Criteria for the same:

→ Space Complexity

→ Time Complexity

**Space Complexity:-** Is the amount of memory the algorithm needs to run to completion.

→ Space needed by an algorithm has following components:

**Instruction space** : Space needed to store compiled instructions

**Data space**: Space needed to store all constant and variable values.

**Environment Stack Space**: is used during execution of functions.

Space complexity = C + Sp(Instance characteristic)

**Time Complexity:** Amount of Time the algo. needs to run to completion.

↙ **Prior analysis**

→ We will analyze the algo before execution

↘ **Posterior Analysis**

→ while algo executed we measure the execution time

→ It gives accurate value but is very costly.

| Complexity of Algorithms | |
|---|---|
| Best case | Inputs provided in such a way that the minimum time required to process them. |
| Average case | The amt. of time the algo takes on avg. set of inputs |
| Worst case | Amt. of time algo takes on worst possible set of inputs. |

*** Asymptotic Notations:-** It is used to describe the running time of an Algorithm.

- Order of growth of function.

## * Big Oh Notation (O)

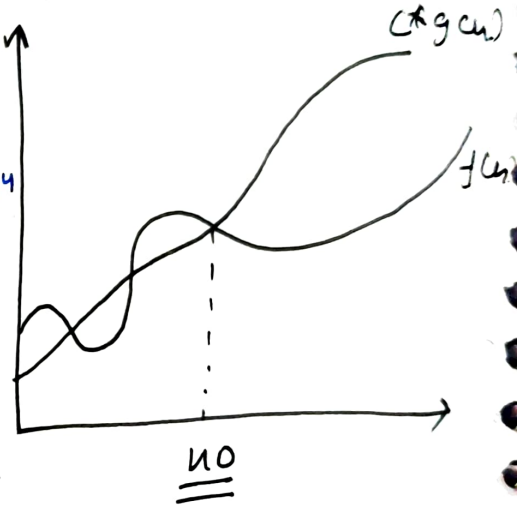
→ It is used to represent the upper bound of an algorithm running time

→ Using Big O we can give the maximum running time of an algorithm

→ Worst case

Definition:- Let $f(n)$ and $g(n)$ be two non negative functions we can say that $f(n)$ is $O(g(n))$ if and only if there exist a positive constant 'C' and 'no' such that $f(n) \leq C \cdot g(n)$ for all non-negative values of n where $n \geq n_0$

Here $g(n)$ is the upper bound of $f(n)$




Ex: $f(n) = 2n^4 + 5n^2 + 2n + 3$

$f(n) \leq 2n^4 + 5n^4 + 2n^4 + 3n^4$

$\leq n^4 (12)$

$\leq 12n^4$

$\therefore$

$g(n) = n^4 \quad , n \geq 1$

$C = 12, \quad n_0 = 1$

$f(n) = O(n^4)$

## * Big Omega (Ω) Notation

→ It is used to represent the lower bound of an algorithms running time

→ Best case

→ Using big Omega can give the minimum running time of an algorithm

**Definition** : The function $f(n) = \Omega(g(n))$ if and only if $f(n) \geq C \cdot g(n)$ for all positive constants C and $n_0$ & $n \geq n_0$.

Ex:-

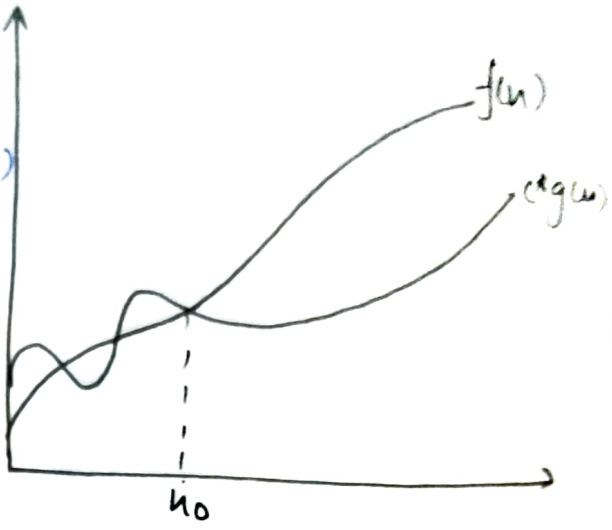
$f(n) = 2n^4 + 5n^2 + 2n + 3$

$f(n) \geq 2n^4$ (as $n \to \infty$, lower terms are insignificant)

$f(n) \geq 2n^4$, and $n \geq 1$

$\therefore g(n) = n^4$, $C = 4$, $n_0 = 1$

$\therefore f(n)$ is $\Omega\, g(n^4)$




## * Big Theta Notation (Θ)

→ Average case

→ It is in between the upper bound and the lower bound of running time of an Algorithm.

**Definition:-** Let $f(n)$ and $g(n)$ be two non negative functions we can say that $f(n)$ is said to be $\Theta(g(n))$ if and only if there exist two positive constants '$c_1$' and '$c_2$' such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all non -ve values of $n$, $n \geq n_0$.

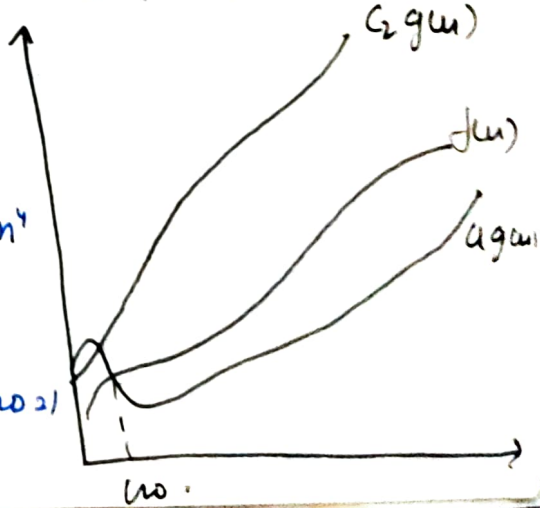
$f(n) = 2n^4 + 5n^2 + 2n + 3$

$= 2n^4 \leq 2n^4 + 5n^2 + 2n + 3 \leq 12n^4$

$= 2n^4 \leq f(n) \leq 12n^4$

$g(n) = n^4$, $\therefore c_1 = 2$, $c_2 = 12$ $n \geq 1$, $n_0 = 1$

$\therefore f(n) = (n^4)$



## Probabilistic Analysis:-

→ an approach to estimate the complexity of an Algorithm.
→ It uses the probability in the analysis of problems
→ It stars from an assumption about a probabilistic distribution of set of all ponible inputs
→ The assumption is then used to design an Algorithm or to compute an expected running time of an unknown algorithm.

for a probabilistic average case analysis it is generally assumed that all ponible permutations are equally likely that is the probability that x will be found at pos 1 is $1/x$ ... $\frac{1}{n}$ so on.

## Amortized Analysis:-

→ It refers to finding the average running time per operation over a worst case of operations. The main goal of amortized analysis is to analyze the time per operation for a series of operation

→ Amortized cost per operation for a sequence of n operations divided by n.

Techniques are

① Aggregate Here upper bound on total cost is selected for a series of n operations : avg. cost = $T(n)/n$

② Accounting In this method the individual cost of each operation is determined by combining the immediate execution Time and its influence on running of future operations

③ Potential Like Accounting but overcharges operations early to compense for undercharges later.

# Heap Sort

**Heap:-** Heap is a data structure which is almost complete binary tree form.

**Maxheap:** It is an almost complete binary tree such that the value of each node is greater than or equal to those of its children.

**Min Heap:-** It is an almost complete Binary Tree such that the value of each node is less than or equal to those in its children.

## Representation of Heap Tree:-

→ Root is stored at index 1.

→ Left child is found at index $2i$

→ Right child is found at $2i+1$

→ Parent of an element stored at $i$ can be found by $\lfloor i/2 \rfloor$.

**Two types of Heaps:-** → Binary heaps
→ Fibonacci heaps

## → Functions of Heap:

① **MAX Heapify:-** In order to maintain the max heap property we have max heapify function whose parameters are Array (input) and index $i$ into the array.

→ This function assumes that left and right subtrees are maximum heaps but the parent is smaller than both thus it lets value at parent to be floated down so that the max heap property doesnot get violated

## Pseudo code for Max heapify:

```
MAX Heapify (A, i):
    L = left (i);
    r = right (i);
    if L ≤ A.heap size and A[L] > A[i];
        largest = L
    else largest = i
    if r ≤ A.heap size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX heapify (A, largest)
```

## Analysis of Max Heapify:

Running time of Max heapify on a node at index $i = \theta(1)$ +

If the recursive call also occurs (if the bottom level is exactly half full) $= \frac{2n}{3}$

$\therefore T(n) \leq T\left(\frac{2n}{3}\right) + \theta(1)$

Solving it using Master Theorum:

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

Or $T(n) = O(h)$ ← height of a node

**Build Max Heap :-** We use the Max heapify in bottom up manner to convert the Array of size n to a max heap.

If begins with $\lfloor \frac{n}{2} \rfloor$ = element's index which is the last internal vertice.

```
Build Max heap (A)
    A.heap size = A.length
    for i = ⌊A.length/2⌋ down to 1
        MAX Heapify (A, i);
```

**Analyze :-** If we need to make n calls and each call makes a recursive call of Max heapify

∴

T(n)/# n-heap has height $\lfloor \log n \rfloor$ and at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

$$= O(n2)$$

$$= O(2n)$$

$$\boxed{= O(n)}$$ Ans

**Heapsort:-** This function exchanges the value at A(1) with A(n) and continue the process by ~~decrementing~~ discarding n that is by decrementing n

```
HeapSort(A)
    BuildMaxHeap(A)
    for i = A.length down to 2
        exchange A[1] with A[i].
        A.heapsize = A.heapsize - 1
        MaxHeapify(A, 1)
```

**Analysis:-**

$T(n)$ = Time complexity of Build Max heap + Time complexity of Max heapify

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

$T(n) = O(n \cdot \log n)$.

| functions | Time complexity |
|---|---|
| Heap Sort | $O(n \lg n)$ |
| Max heapify | $O(\log n)$ |
| Build Max Heap | $O(n)$ |

**Question:** Apply Heap Sort on Array

$A = \{5, 13, 2, 25, 7, 17, 20, 8, 4\}$

Procedure : Heap Sort (A)

Heap Sort (A):

Build MAx Heap(A)

for i = A.length down to 2

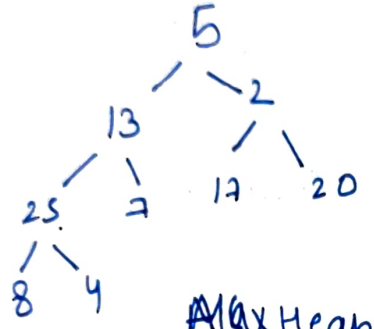
exchange A[1] with A[i]

A.heapsize - - ,

Max Heapify (A, 1)

Step ① Build Max Heap of Array A.

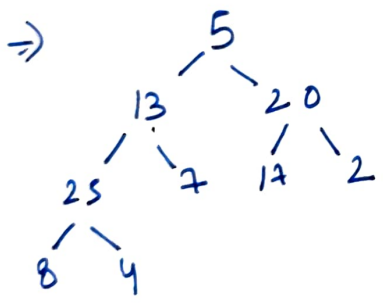
Build MAX Heap (A) =



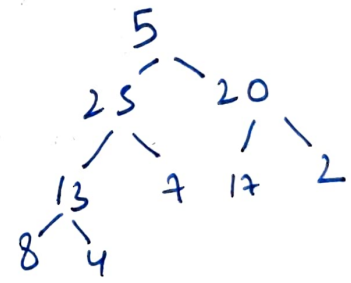

⇒ Start from index $\lfloor \frac{A.length}{2} \rfloor$

$= \lfloor 4.5 \rfloor = 4$ and decreasing upto index ① ie the root

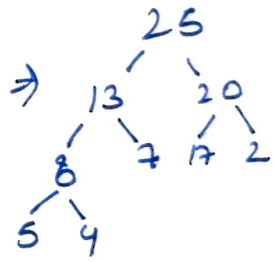
MAx Heapify (A, 4)




(A, 3)



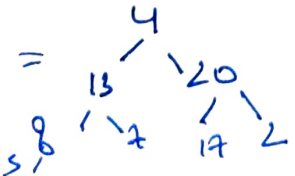

(A, 2)

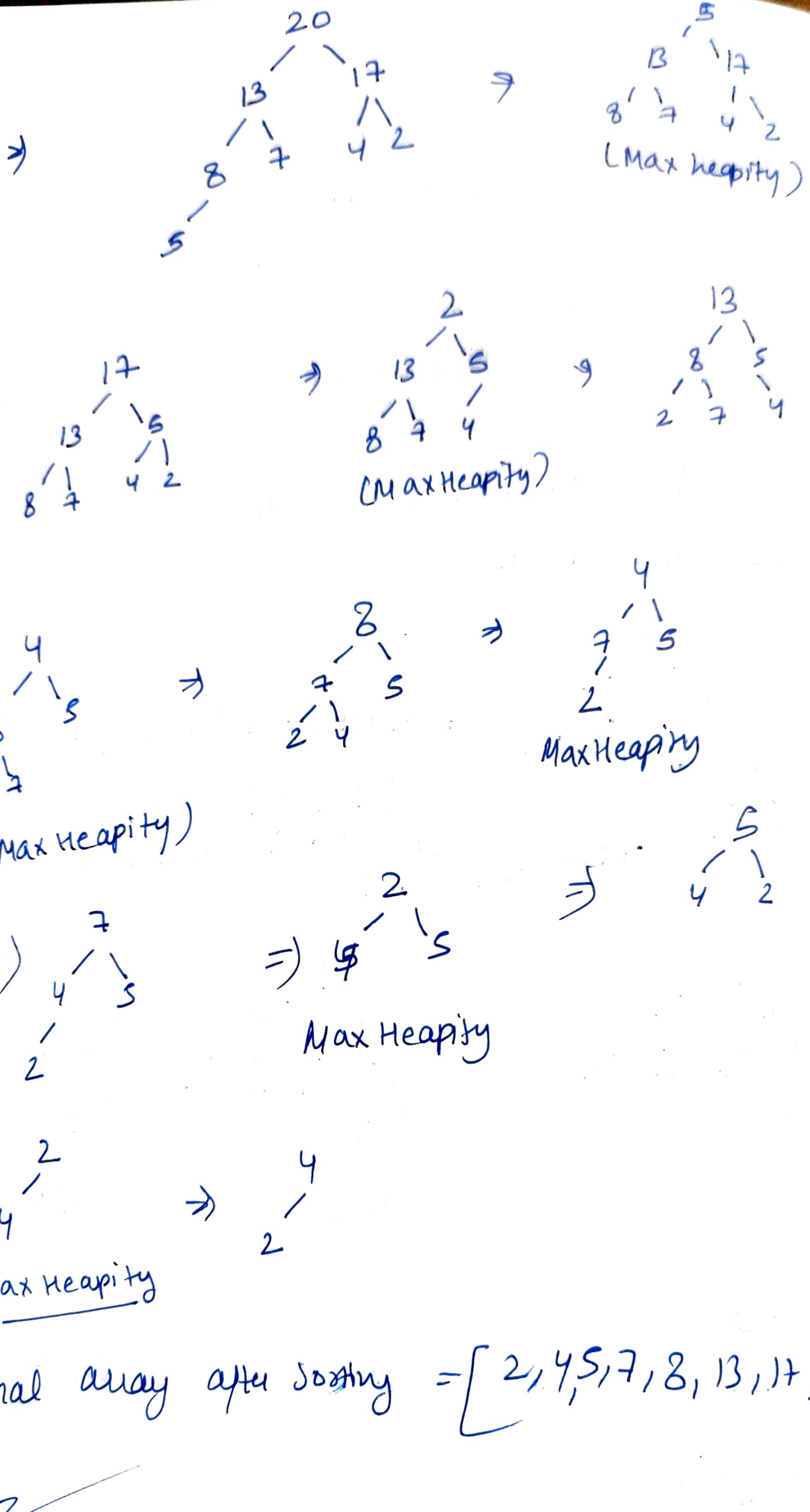



(A, 1)

Step ②

A = [2, 4, 5, 7, 8, 13, 17, 20, 25]






Final array after sorting = [2, 4, 5, 7, 8, 13, 17, 20]

**Priority Queue:-** is implemented using the Heap. Here element can be inserted from last and gets placed at its proper location But always only the root element is deleted/removed.

Insert (S,x) heap increase key (S, ~~k~~, x), Heap extract-max (A);

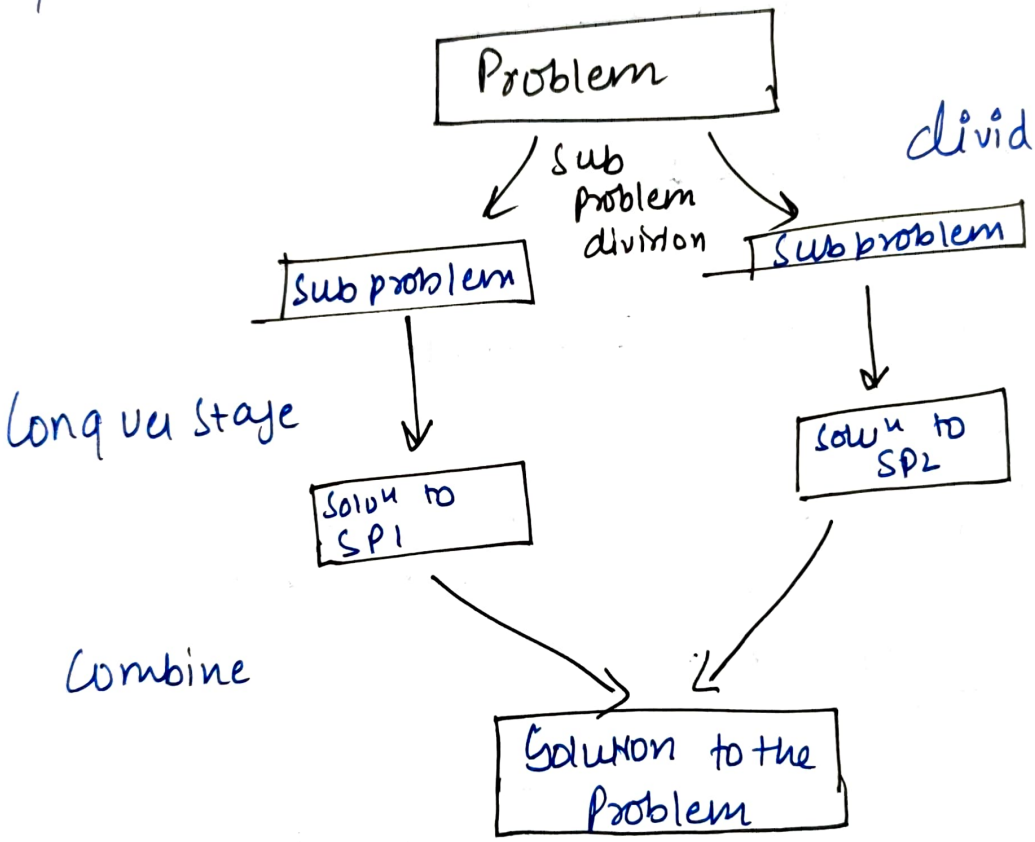
```
MaxHeapInsert (A, key)
  A.heapsize = A.heapsize + 1;
  A[A.heapsize] = -∞)
  Heapincrease key (A. A.heapsize, key)

Heap increase key (A, i, key)
  if key < A[i]
        error new key is smaller.
  A[i] = key
  while i > 1 and A[Parent(i)] < A[i]
        exchange A[i] with A[Parent(i)]
        i = Parent(i)

Heap extract Max (A)
  max = A[1]
  A[1] = A.[A.heapsize]
  A.heapsize--
  Max Heapify (A, 1)
  return max
```

| Operation | Running Time |
|---|---|
| Heapinsert | O(lgn) |
| Heap increase key | O(lgn) |
| Heap extract max | O(lgn) |

# Divide and Conquer

→ Best known general (Algorithm) design technique. Basic principle behind it is that it is easier to solve smaller instances of a problem rather than having 1 big problem

→ Divide and conquer technique involves solving a particular problem by dividing it into smaller or sub problems, recursively solving each problem and then combining the solutions of subproblems to produce a solution of the original problem.




# Applications of Divide and Conquer methods are :-
1) Binary Search
2) Quick Sort
3) Merge Sort

# Recurrence Relations

→ Recurrence Relation for a sequence (S) is a formula that relates all but a finite number of terms of S to previous terms of the sequence namely, $\{a_0, a_1, a_2 \ldots\ldots a_{n-1}\}$, for all integers n with $n \geq n_0$ where $n_0$ is a non negative integer. Recurrence Relations are also called as difference Equations.

→ Sequence are often most easily defined with a recurrence relation.

Ex: factorial, fibonnaci, Quicksort, Binary search

→ Recurrence Relations defines in terms of itself

• Techniques to Solve Recurrence Relations:-

(1) Substitution Method

→ Also called as Plugging method

→ It is a method of solving problems of divide and conquer form.

→ Substitution Method comprises of two steps.

① Guess the form of solution

② Use mathematical inductions to find the constant and show that the solution works

*NOTE:-

We use floor function to determine upper bound.

We use ceil function to determine lower bound.

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Now the solution is of the form $O(n \log n)$
We choose the constant $c > 0$ in such a way that $T(n) \leq c \cdot n \log n$

Now $m < n$
$m = \lfloor \frac{n}{2} \rfloor$

$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \cdot \log(\lfloor n/2 \rfloor)$

Substituting in recurrence yields

$T(n) = 2(c \lfloor n/2 \rfloor \cdot \lg(\lfloor n/2 \rfloor)) + n$

$T(n) = c \cdot n \log(\lfloor n/2 \rfloor) + n$

$T(n) = cn \log n - cn \log 2 + n$

$= cn \log n - cn + n$

$T(n) \leq cn \lg n \quad \forall \; c \geq 1$

# Main disadvantage of this way of solving Recurrence is to know the solution in advance or you must be able to guess the solution really well.

(11) Recursion Tree :-

→ It is a visual approach

→ Also uses repeated substitution to solve a recurrence relation

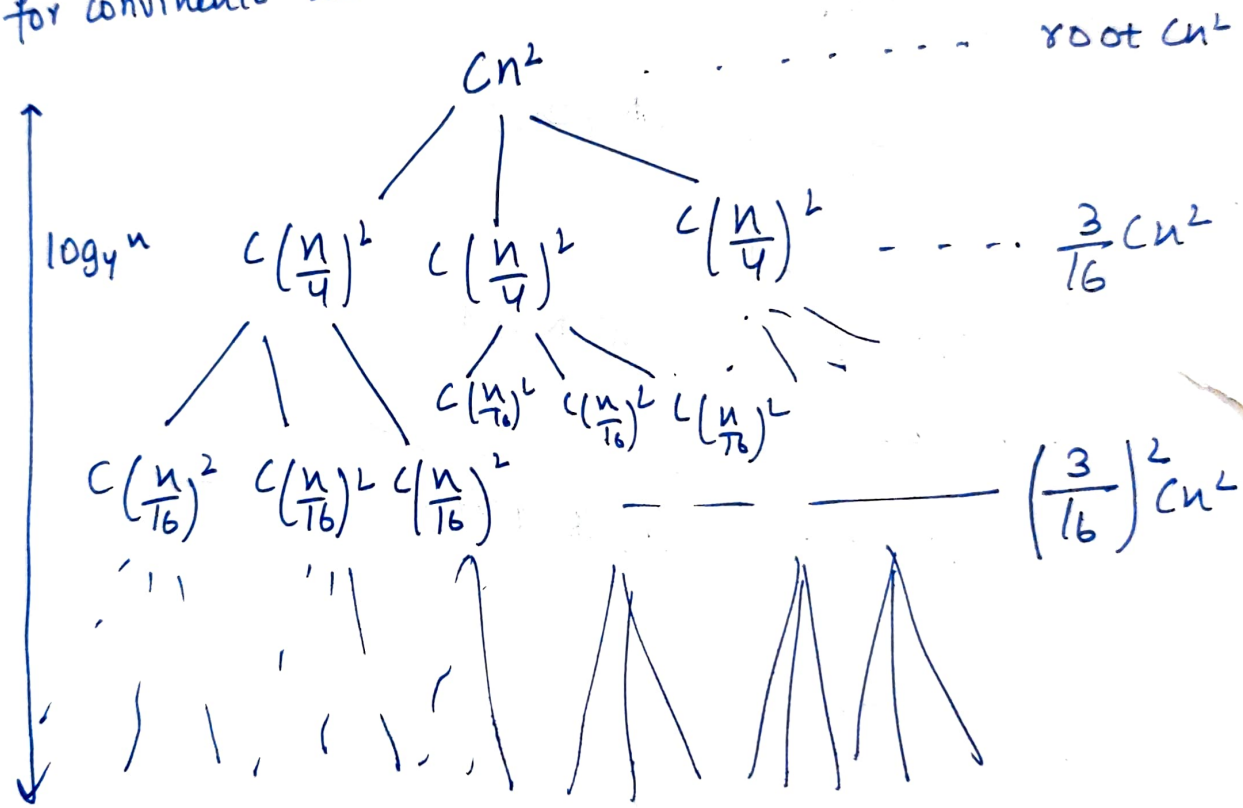
→ In using the recursion tree method, we draw a tree R where each node represents a different substitution of the recurrence.

→ For divide and conquer recurrences the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v.

Ex:- $T(n) = 3T(n/4) + cn^2$

Recursive Tree method :

for convinience we assume n is exact power of 4




Height of tree is $\log_4 n$

No. of nodes $n^{\log_4 3}$

$= \Theta(n^{\log_4 3})$

+

Total $= O(n^2)$

# Master Theorum Method :-

Let us consider the general form for recurrence relation of decreasing function as:-

$$T(n) = aT(n-b) + f(n)$$

where
1) $a > 0$
2) $b > 0$

and $f(n)$ is in the form of $n^k$ where $k \geq 0$

## Rules for Solving

**Rule ①:-** If $a = 1$

$$\text{Soln} = O(n * f(n))$$

**Rule ② :-** If $a > 1$

$$\text{Soln} = O(f(n) * a^n)$$

→ if $b > 1$

$$\text{Soln} = O(f(n) * a^{n/b})$$

**Rule ③ :-** $a < 1$

$$\text{Solution} = O(f(n))$$

→ This is the master theorum for recurrence relation with decreasing function.

# Master Theorum for dividing functions

→ General form of recurrence relation for dividing functions

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where we assume that $f(n)$ is in the form of

(1) $a \geq 1$

(2) $b > 1$

(3) $f(n)$ is in the form of $(n^k \log^p n)$

* for finding the solutions of Master theorum. We need to find ① $\log_b^a$ ~~$\log_b^a$~~

② $k$

Case ① if $\log_b^a > k$ then

Soln is $O(n^{\log_b^a})$

Case ② if $\log_b^a = k$ then

(a) if $p > -1$ solution = $O(n^k \log^{p+1} n)$

(b) if $p = -1$ Soln = $\theta(n^k \log \log n)$

(c) if $p < -1$ Soln = $\theta(n^k)$

Case ③ if $\log_b^a < k$ then

(a) If $P \geq 0$ then Solu$^n$ $= O(n^k \log^P n)$

(b) If $P < 0$ Solu$^n$ $= O(n^k)$

(c)

Example:-

q(a) $\rightarrow T(n) = T(n-2) + n^2$

Here $a = 1$
$b = 2$
$f(n) = n^2$ which is in the form of $n^k$

as $a = 1$

$\therefore$ solution is $= O(n \cdot f(n))$
$= O(n \cdot n^2)$
$= O(n^3)$ Ans

(a) $T(n) = 2T(\frac{n}{2}) + n^4$

Here $a = 2$
$b = 2$
and $f(n)$ is in the form of $(n^k \cdot \log^P n)$

$k = 4$ , $P = 0$

now $\log^a b \Rightarrow \log 1 = 1$

now $1 < 4$ and $P = 0$

$\therefore$ Solution $= O(n^4 \log^0 n) = O(n^4)$ Ans

## * Amortilized Analysis:

→ It is also used to analyze the time complexity of an algorithm.

* It is always used to compute the upper boundary ie the maximum number of steps required to complete the Algorithm

→ It is also used to compute the average case time complexity of Algorithm.

→ Amortization means change

→ In this analysis the actual cost of some of the operations can be transfered to some other operations in the sequence.

→ By doing this reduces the cost of some operations and also some operation's cost increases.

→ Actual time taken by algo is called as its actual cost.

* The sum of amortize cost of all the operation is always greater than or equal to the sum of actual cost of all the operation.

$$\sum_{1 \le i \le n} \text{amortized cost}(i) \geq \sum_{1 \le i \le n} \text{actual cost}(i) \quad \text{---(1)}$$

### Potential:

Potential is the amount of cost by which the first

Operations are overcharged

$$P(i) = \text{amortized}(i) - \text{actual cost}(i) + P(i-1) \quad \text{—(2)}$$

$$\Rightarrow \sum P(i) = \sum (\text{amortized}(i) - \text{actual}(i)) + \sum P(i-1)$$

$$\Rightarrow \sum P(i) - \sum P(i-1) = 0$$

$$\Rightarrow P(i) - P(0) \geq 0 \quad \left[ \begin{array}{l} P(i) = (P(1) - P_0) + (P_2 - P_1) \\ = (P_2 - P_0) \\ \geq P(n) \end{array} \right]$$

$$\Rightarrow P(n) - P(0) \geq 0$$

# Now there are three methods available to compute the ammorized cost

## ① Aggregate Method

It computes the ammortized cost by finding sum of actual costs of all the operations and set ammortized cost equal to the sum of actual cost to n

$$\boxed{\text{amortized} = \frac{\text{sum of actual cost}}{n}}$$

* It satisfies 1st principle

Ex:- A car charges every month in $ 50 and it changes $100 for every quarter and it charges 200 for every year.

| Jan | feb | March | April | May | June | July | Aug | Sep | Oct | Nov | Dec |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 50 | 100 | 50 | 50 | 100 | 50 | 50 | 100 | 50 | 50 | 200 |

Amortised cost is:

$$= 200 \times \frac{n}{12} + 100\left(\frac{n}{3} - \frac{n}{12}\right) + 50 \times \left(n - \frac{n}{3}\right)$$

$$= \frac{100 \times n}{6} + 100 \times \frac{n}{4} + 100\,\frac{n}{3}$$

$$= 100\left(\frac{n}{6} + \frac{n}{4} + \frac{n}{3}\right)$$

$$= 100\left(\frac{9n}{12}\right)$$

$$= 100 \times \frac{3n}{4} = 75 \Rightarrow \frac{3 \times 100}{4} \quad \therefore n = 400$$

Here.

| Jan | feb | March | April | May | June | July | Aug | Sep | Oct | Nov | Dec |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 50 | 100 | 50 | 50 | 100 | 50 | 50 | 100 | 50 | 50 | |
| 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | |
| 25 | 50 | 25 | 50 | 75 | 50 | 75 | 100 | 75 | 100 | 125 | |

Dec

actual : 200

amortised : 75.

Potential : 50

# Accounting Method

In this we analyze time complexity of an algorithm by guessing amortized cost and calculate potential using rule ② and verifies if it satisfies rule ③

* If it doesnot satisfies then we assume another value for ammortized complexity.

# Potential Method

In this method we guess the potential of each operation and calculate ammortized cost of each operation.

Time complexity of the following algorithm:-

```
fun(int n)
{   for (int i=1; i<n; i++) {          ①
      for(int j = 1; j<n ; j+=i){
                                       ②
    { sum = sum + i*j;
      }
   }
   return sum;
}
```

In step count Method adding count++ in ① and ②

# Probabilistic Analysis

→ Use of probability in the analysis of problem ; to evaluate the running time of an algorithm.

→ In order to perform a probabilistic analysis, we must use knowledge of or make assumptions about, the distribution of the inputs Then we analyze the algorithm computing an average case running time, where we take the average over the distribution of the possible inputs.

→ we refer such a running time as average case running time.

Uniform random permutations : each possible $n!$ permutation appears with equal probability.

→ When an algorithm is executed on a valiable is called Sample point

Ex:- Consider a Linear Search Algorithm which searches a target element say x in the given list of size n In the worst case the algo will examine all the elements.

In average case analysis it is generally assumed that all possible terminations are equally likely that is, the probability that x will be found at position 1 is $\frac{1}{x}$ and position 2 is $\frac{1}{n}$ and so on.

The average Seach cost will be sum of all possible Seach cost each multiplied by associative probabilities $n=5$ we have:

Average Search cost $= \frac{1}{n\left(\frac{n(n+1)}{2}\right)} = \left(\frac{n+1}{2}\right)$

## Randomized Algorithms

→ An algorithm is randomized if its behaviour is determined not only by its input but also by values produced by a random number generator

→ We shall assume that we have at our disposal a random number generator RANDOM. A call to RANDOM(a,b) returns any number between a and b inclusive with probability of each occuring equally likely.

→ When analyzing the running time of a randomized algorithm, we take the expectation over distribution of values returned by the random number generator.

→ Expected Running time is the running time of Randomized Algorithm.

### • Indicator Random Variable.

→ provide convienent method for converting between probabilities and expectations.

→ Suppose we are given a sample space S and an event A. The indicator random variable associated with A

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ doesnot occur} \end{cases}$$

$$X_{Head} = I\{H\} = \begin{cases} 1 & \text{if Heads} \\ 0 & \text{if Tails} \end{cases}$$

$$E[H_H] = E[I\{H\}]$$
$$= 1 \cdot Pr\{H\} + 0 \cdot Pr\{T\}$$
$$= 1 \cdot \{1/2\} + 0 \cdot \{1/2\}$$

$$\boxed{E(X_H) = 1/2.}$$ → expectation

⇒ Indicator random variable are useful for analyzing situations in which we perform repeated Random Trials.

Let X be the random variable denoting the total number of heads in the n coin flips.

$$X = \sum_{i=1}^{n} X_i$$

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$$

$$= \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} 1/2$$

$$= n/2.$$

**Closed form of expression**

Sum of 1 n natural no

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

Sum of squares

$$= \frac{n(n+1)(2n+1)}{6}$$

Sum of cubes

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

GP $= 2^{n+1} - 1$

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

# Graphs

→ is a simple way of encoding pairwise relationship among a set of objects, it consist of a collection V of nodes and E collection of Edges, each of which join two nodes

→ **Definition** Graphs are the discrete structures that consist of a non empty set of vertices V and a set of edges connected to those vertices

→ set of Edges.

$G = \{V, E\}$

↓

Set of vertices

**Directed:-** Graphs having directed edges or a graph whose edges have a starting and ending point.

Simple — Multiple

**Undirected graph:-** Graphs having non directed edges or the graph whose edges starting and ending point is not known.

## #Graph Models (Representation of Graphs in real world)

→ Social Networking
- → nodes represent the individual / organisation
- → edges represent communication / relation.

→ Influence
- → directed graphs to represent how a person influence something / somebody.

→ Transportation

→ Call Network

→ Niche Overlap in Ecology.

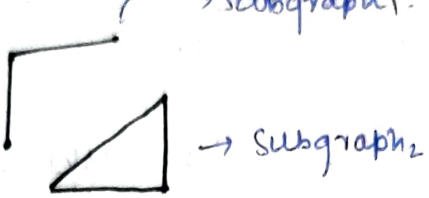
**Path:-** path is a length b/w between two point $v_1$ to $v_n$ consisting of sequence of edges.

→ path consists donot consist an edge morethan once

→ Singe vertex is a path of length zero

**Connectivity:-** An undirected graph is said to be connected if there is a path between distinct vertices of the graph.

Connected.




not connected

★ let G be an undirected graph. Any two of the following statement implies the third.

(1) G is connected

(2) G doesnot contain a cycle

(3) G has n-1 edges.

**Problem:-** Determining s-t connectivity?

→ small problems it can be visually solved but for big/ complex problems then we must drive an algo to search for the path.

2 natural Algos to determine s-t connectivity in a graph are:- → Depth first search

↙

Breadth First Search

## (1) Breadth First Search (BFS)

→ Simplest Algorithm to determine s-t connectivity

→ Layer by layer traversal

→ we start from 's' (source) include all the nodes that are joined by an edge to s [first layer]. we than include all the nodes that have an edge to the nodes of first layer and we proceed like wise.

→ Layer $L_i$ consist of all the nodes that have an edge to the source node.

→ for layers $L_1 \dots L_j$, the layer $L_{j+1}$ consist of all the nodes that donot belong to earlier layer and have node in layer $L_j$.

BFS produces BFS tree as the output in a natural way, a Tree 'T' rooted at 's' on set of nodes reachable from 's'.

### Set of all connected components :-

Pseudocode :-

R will consist of nodes to which s has a path

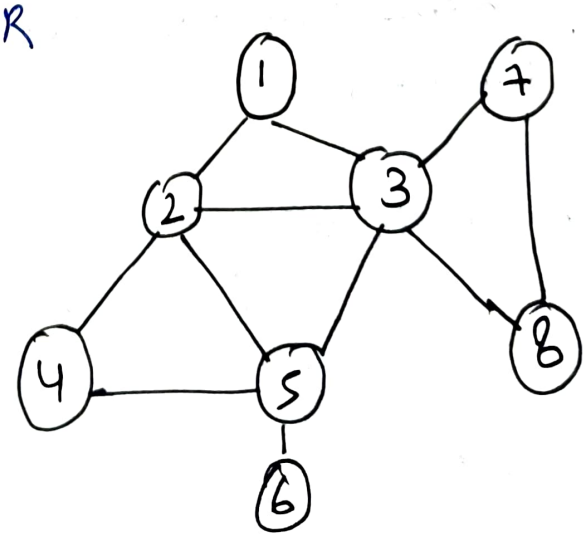
Initially $R = \{s\}$

While there is an edge $(u,v)$ where $u \in R$ and $v \notin R$

add v to R

End while

Graph :-



## Implementation of a Graph → Adjacency Matrix

↓

## Adjacency List

→ List of all the vertex that are adjacent to each vertex of the graph

→ Tradeoff can be used with adjacency list when graph is sparse

Here we represent the graph as a Matrix and if edge exist b/w vertices we add 1 to the Matrix or 0 otherwise.

$$a_{ij} = \begin{cases} 1 & \text{if edge is b/w } v_j \text{ and } v_i \\ 0 & \text{if no edge} \end{cases}$$

→ adjacency matrix for undirected graph is symmetrical.

★ $G = (V, E)$

$n$ = Set of nodes, $m$ = Set of Edges.

Running time of Algo depends on relation b/w $m$ and $n$

∴

Adjacency List = $O(m+n)$ Space

Adjacency matrix = $O(n^2)$ Space

List = length of all List $2m = O(m)$

length of List Adj[v] is $n_v$ and total length over all nodes is $O(\sum_{v \in V} n_v)$

## • Implementation of BFS algo

→ Adjacency List is used to Implement Breadth-first search using Queue Data Structure

↓

(First in First Out)

Pseudocode:-

BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist single element s
  Set layer counter i=0
  Set BFS Tree = ∅
  while L[i] is not empty
    Initialize an Empty list L[i+1]
    For each node u ∈ L[i]
      Consider each edge (v,u) incident to u
      If discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u,v) to tree T
        Add v to list L[i+1]
      Endit
    End for
    Increment layer counter by 1
  End while

The list L[i] = can be stack or queue.

> Analysis of Algo

Atmost we take n L[i] list → O(n) time
now each node appears on atmost one list : for loop runs n times → O(n) times

For loop can take less than O(n) time if it has fewer neighbors :
Let $d_u$ = degree of node u
:
For loop considering edges → O(m) and for bo

$= O(\sum_{v} n_u)$

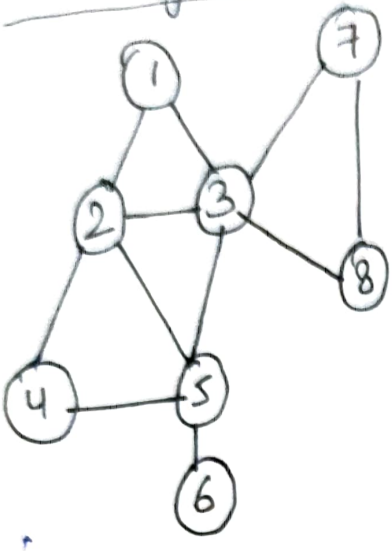
Now $\sum_{v} n_u = 2m$

∴ Total time spent in looking all edges is $O(m)$

So total time spent is $O(m+n)$

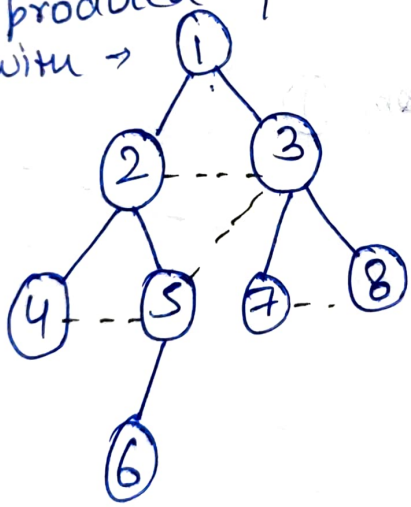
**Running Time = $O(m+n)$**

from graph (3) we implement BFS on the graph




∴

BFS Tree produced after implementing the Traversal starting with →




- - - = part of graph and not of tree

# Depth First Search

→ depth or vertical search for the traversal
→ Search till deadend is found.
→ Involves Backtracking.

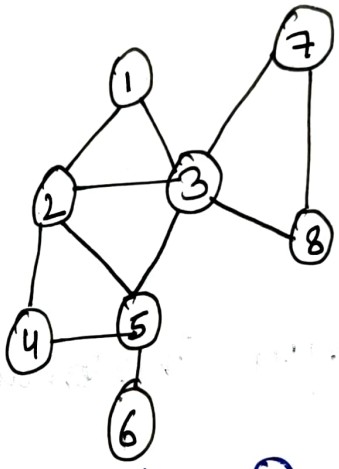
→ We start from node '1' (Source node) then we move to node v ~~until~~ and continue our search in same way/ direction until we reach the dead end

→ It goes deep as possible and retreat when necessary

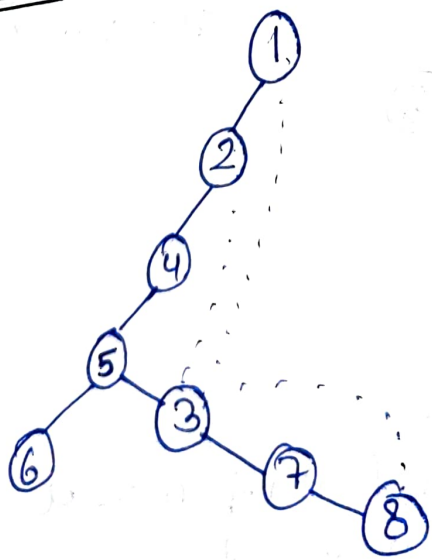
**Pseudocode:-**

DFS(u):
Mark u as Explored and Add u to R
For each edge (u,v) incident to u
If v is not marked "Explored" then
Recursively invoke DFS(v)
Endif
Endfor.

**Graph :-**




∴ After DFS starting from (1)



# Implementation of DFS :-

→ with the help of Stack → last in first out

DFS(s):

Initialise Stack 'S' with Source node.

While S is not empty

Take a node u from S

If explored[u] = false then

Set Explored[u] = ~~false~~ true

For each edge (u,v) incident to u

Add v to Stack.

Endfor.

Endif

Endwhile.

Analysis:

The main step of Algo is to add or delete nodes which takes ⟶ O(1) time

$O(n_v)$ ⟶ elements added to Stack.

Total no. of elements added ⟶ $\sum_{u \in V} n_v = 2m$

∴

This gives $O(m+n)$ bound on DFS.

if graph is implemented by adjacency List representation.

NOTE:- The above algo takes same time as recursive one except that adjacency List is processed in reverse order.

# Applications of BFS and DFS

## ① Bipartitness → application of BFS

**Bipartite graph:** A graph which has a set of vertex V divided into two disjoint sets $V_1$ and $V_2$ such that each edge connects connects vertex from $V_1$ to vertex from $V_2$ only.

→ If graph is bipartite then it cannot contain an odd cycle.

→ If G is a graph and let $L_1, L_2 \cdots$ be the layers produced by BFS starting at node s

→ There is no edge of G joining nodes of same layer then G is a bipartite graph if even numbered layers can be coloured red and odd numbered layers can be coloured blue.

→ If edge exist in same layer then odd cycle exist and graph is not bipartite.

We can easily check bipartitness by taking colour over the nodes

Colour[v] = red if i+1 = even

Colour[v] = blue if i+1 = odd

∴ Total time for colouring = O(m+n)

**Strongly connected** A directed graph is strongly connected if there is path from u to v and also a path exist from v to u

→ Directed Acyclic Graph (DAG)
→ Directed graphs with no cycle
→ used to depict precedence relation or dependencies in a natural way.

**Topological ordering** :- edges point forward in ordering
∴ for nodes $v_1, v_2, v_3 \ldots v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$.

# If G has topological ordering than it is DAG.

**Designing & Algo:-** (Start with the node that have no incoming edges)

Topological ordering of G :

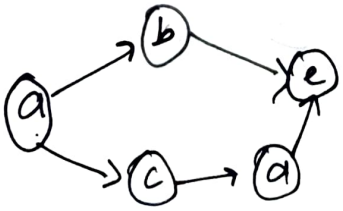
Find a node v with no incoming edges and order it first

Delete v from G

Recursively compute a topological ordering of $G - \{v\}$

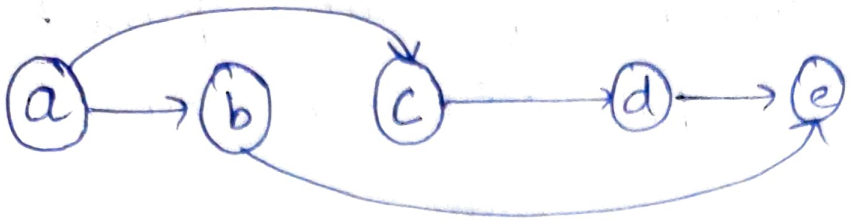
append this after v.

**Question:-** Give the topological ordering of the given graph and also state the possible topological ordering it can have




- Topological ordering / Sort

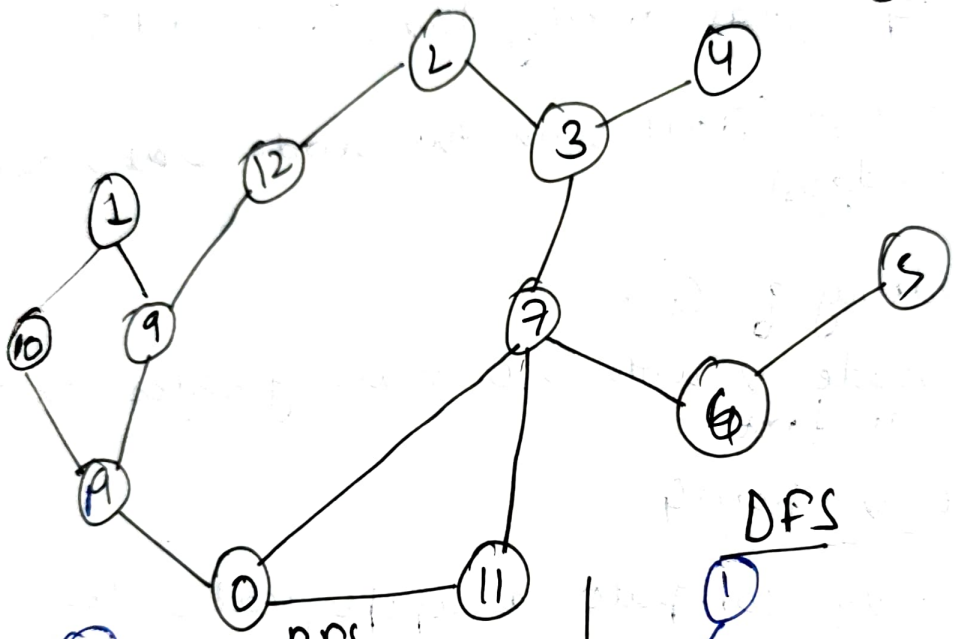
Start with node that has no incoming edge



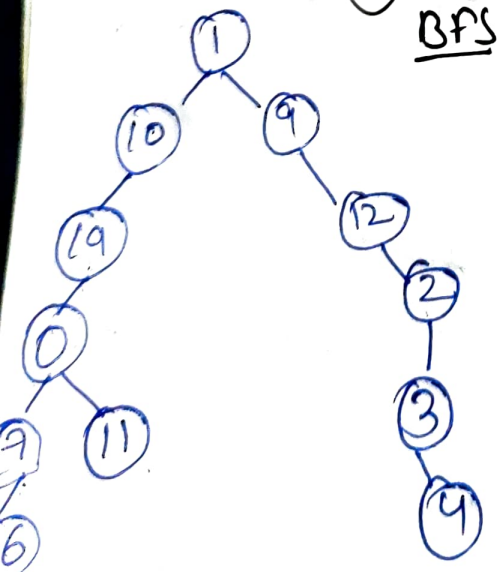

Possible sorting { a b c d e
a c d b e
a b d e }

Question: Implement BFS, DFS on the following graph.

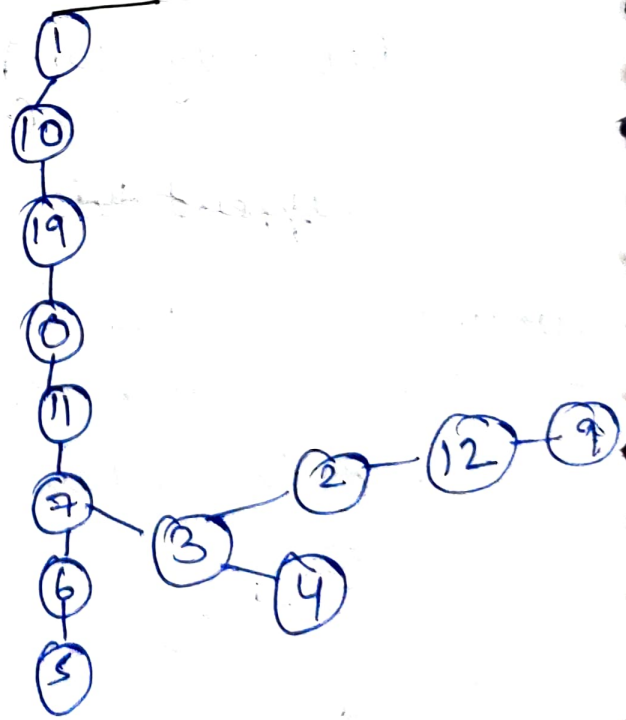
Starting with (1)




BFS




DFS



# MINIMUM COST SPANNING TREE

## ① Spanning Tree:-

Subset of connected graph is called Spanning tree

Let $G(V, E)$ is undirected connected graph $G'(V, E')$ where $E'$ is a subset of $E$ ($E' \subset E$) without any cycle than $G'$ is called Spanning tree

→ A Spanning tree must cover all vertices

→ There is a single connected path b/w any two pair of vertices

→ Spanning Trees have n-1 edges where n is the no. of vertices

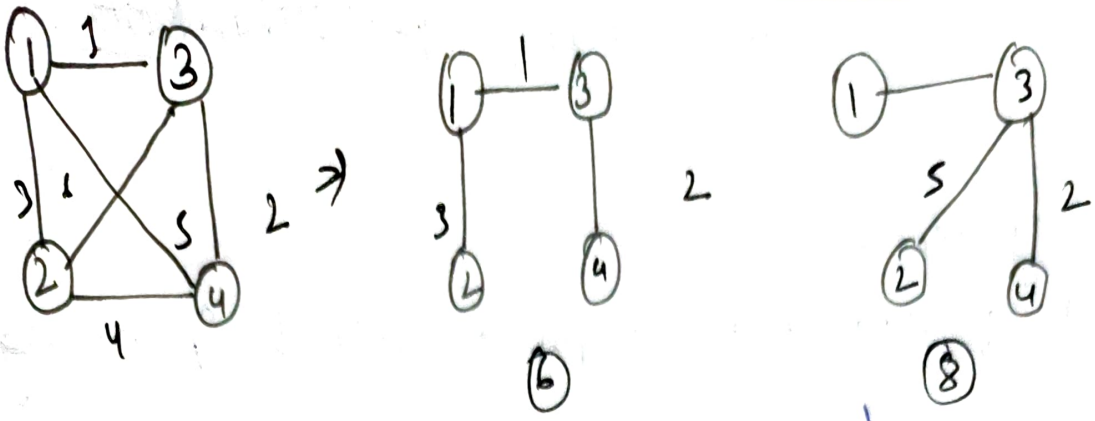
→ If an edge is removed from the tree then it is disconnected into two components.

→ When an additional edge is added to tree then it acts as graph

→ In Spanning tree each edge is labelled with an integer value called as weight of edge. It describes the distance from source to destination

→ Sum of weights of all edges in the Spanning tree of Spanning or cost of Spanning tree.

→ Minimum Spanning Tree is a tree of Spaning tree with minimum cost.




* Possibilities of Spanning Tree

## Minimum Cost Spanning Tree:

For a graph there are so many spanning trees but less number of minimum cost Spanning Trees

→ If two or more have same weight than MST is more than one but if they have different weight MST is one with least.

Applications → communication network
→ airline routes

## To construct Minimum Spanning Tree:

① Prim's Approach

② Kruskal's Approach

Both approaches build the minimum Spanning Tree by selecting ~~edges~~ subset of edges in the given graph so that it follows subset paradigm in greedy method to construct min Spanning Tree.

* In greedy subset paradigm min Spanning tree is constructed by selecting one edge in each stage

→ The metric used to optimization measurement used to select and edge is minimum cost. It always select an edge which satisfy→

① not included in any tree constructed so far.

② minimies cost of tree and there is no cycle formed by included it.

## Prims Algorithm.

- It constructs the min Spanning Tree based on Vertex selection.
- By selecting (i,j) 'i' must be already existed in the graph and 'j' must be added to the graph.

## Steps for constructing Minimum Spanning Tree:-

① Select an edge (i,j) where vertex 'i' is included in the tree and j is not included in the tree

② The cost of (i,j) is minimum for all possible values of j

③ To determine a minimum cost edge (i,j) it needs to assign for each vertex 'j' an integer value near of j describes a vertex already in the tree i=near[j]

* Now we choose minimum cost edge (j,j) where cost (j, near(j)) is minimum.

## # Pseudocode for prim's Algorithm

Algorithm Greedyprims (E, cost, n, t)

```
MST_prim (G, w, r)                    // r = root
                                      // u = node
1  for each u ∈ G.V                   // V = set of vertices
2      u.key = ∞                      // G = Graph
3      u.π = NIL                      // w = weight
4  r.key = 0
5  Q = G.V                            // Q = min priority Queue
6  while Q ! empty
7      u = Extract_MIN (Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u,w) < v.key
10             v.π = u
11             v.key = w(u,v)
```

Alternatively :-

Algo Greedy prims (E, cost, n, t)
(E → set of edges; n → no. of edges; t → to store no. of edges; cost → cost of adjacency matrix)

Let an edge (k, l) with cost of MCT is selected

minimum = cost (k, l)

t[1,1] = k; t[1,2] = l.

for i = 1 to n do

if (cost[i,k] < cost(i,l)) then

near(i) = k;

else near(i) = l;

near[k] := near[l] := 0;

for j = 2 to n-1 do

{

```
Let us select a vertex 'j' when near[j]≠0 and
cost (j, near(j)) is min
  t[j,1] = j;  t[j,2] = near(j);
  mincost = mincost + cost (j, near(j))
  for k = 1 ton do
   if (near(k)≠0 && (cost (k, near(k)) >
                     cost (k,j)) then

      near(k) = j

  }
  return mincost;
}
```

## Analysis of MST PRIMS :-

BUILD MIN Heap in Line 1-5 ⟶ O(V)

While loop execute V time
and extract max execute logV time ⟶ O(V logV)

The for loop in Line 8-11 execute for the number of edges ⟶ O(E)
+ Line 11 involves decreasekey = O(2m)
= O(logV)

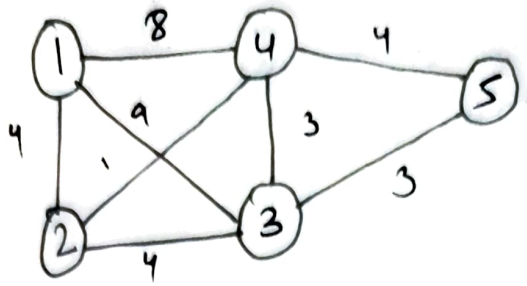
Thus Total Time

= O(VlogV + ElogV)

Running Time = O(ElogV)

Complexity = O(2m logn) Ans

**Question** Construct the Min Spanning Tree for the given graph using Prim's Algorithm




Adjacency Matrix Cost(weight):

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 4 | 9 | 8 | ∞ |
| 2 | 4 | 0 | 4 | 1 | ∞ |
| 3 | 9 | 4 | 0 | 3 | 3 |
| 4 | 8 | 1 | 3 | 0 | 4 |
| 5 | ∞ | ∞ | 3 | 4 | 0 |

Minimum cost edge = 2-4

Cost = 1

for i = 1 to 5

i=1 cost(1,2) < cost(1,4)

4 < 8

near[1] = 2

i=2 cost(2,2) < cost(2,4)

0 < 1

near[2] = 2;

i=3 cost(3,2) < cost[3,4]

4 < 3

near[3] = 2

$i=4$ cost [4,2] < cost [4,4]

1 < 0

cost [4] = 4

$i=5$ cost [5,2] < cost [5,4]

$\infty$ < 4

near [5] = 4

| 2 | 2 | 2 | 4 | 4 |
|---|---|---|---|---|

near [2] = near [4] = 0

$j=1$ near [1] ≠ 0 → True

cost [1, near [1]] = cost [1, 2] = 4

$j=2$ near [2] ≠ 0 → false

$j=3$ near [3] ≠ 0 → ~~False~~.

cost [3, near [3]] = cost [3, 2] = 4

$j=4$ cost (4) ≠ 0 → false

$j=5$ cost (5) ≠ 0 → true

cost [5, (near (5)] = cost (5, 4) = 4

Select j = 1

min cost = 1 + cost (1, 2)

= 1 + 4 = 5.

| 0 | 0 | 2 | 0 | 4 |
|---|---|---|---|---|

K = 1 to 5 if cost (k, near (k)) > cost (k, j)

Check the cost for which not included in the

k=> Cost (3, near(3)) > cost(2,1)

4 > 4 → false

k=5 cost (5, near(5)) > cost(5,1)

4 > ∞ → false.

| 0 | 0 | 2 | 0 | 4 |
|---|---|---|---|---|

Step ②

j=3 near[3] ≠ 0 → True

cost (3, near[3]) = cost (3,2) = 4

j=5 near(5) ≠ 0 → True

cost (5, near(5)) = cost (5, 4) = 4

Select j=3

minmost = 5 + cost(3,2)

= 5+4 = 9

t[3,1] = 3, t[3,2] = 4

near[j] = 0

| 0 | 0 | 0 | 0 | 4 |
|---|---|---|---|---|

if cost (k, near[k]) > cost (k, j)

k=5 cost (5, near(5) > cost (5,3))

cost (5, 4) > cost (5,3)

4 > 3

near (5) = 3

| 0 | 0 | 0 | 0 | 3 |
|---|---|---|---|---|

Step 3 ⓷ $j=5$ $near[5] \neq 0 \rightarrow$ True.

$cost(5, near(5)) = cost(5,3) = 3$

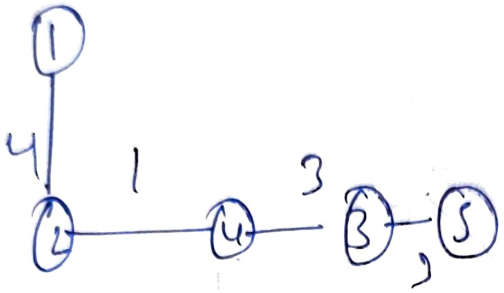
Select $j=5$

$mincost = 9+3 = 11$

$t[4,1] = 5$ $t[4,2] = 3$

| 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|

MST

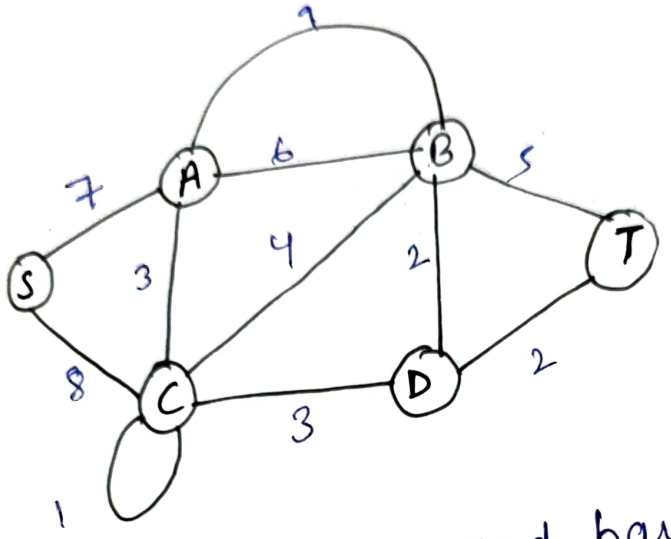


## * KRUSKAL'S ALGORITHM:-

→ helps to determine the minimum cost spanning tree using the greedy approach.

→ This algo treats graph as a forest and every node as its individual tree.

→ A tree connects to other tree only if it has MST properties and doesnot violate it.



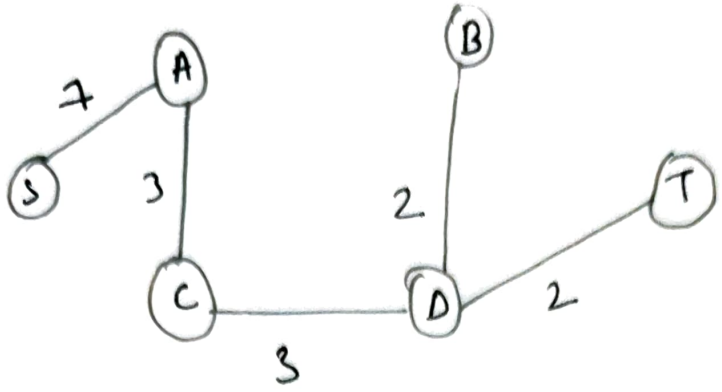

Step ①:- Remove all loops and parallel edges
In case of parallel edges remove all others except the one with the least cost.

Step ②: Arrange all edges in the increasing order of weight

| B,D | D,T | A,C | C,D | C,B | B,T | A,D | A,S | S,C |
|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 9 | 8 |

Step: ③ Add edge which has least weightage
→ We start adding edges to the graph from the one which has least weight. Throughout we shall keep checking that spanning properties rem

intact




This is the required MST

MST kruskal (G, w):

$A = \phi$

for each vertex $v \in V$

Make Set(v)

Sort the edges of $G.E$ in increasing order of weight

for each edge taken in ~~non~~ increasing order of weight

if Find-Set(u) ≠ Find-Set(v)

$A = A \cup \{(u, v)\}$

UNION(u, v)

return A;

Analyse

$A = \phi$ ⟶ $O(1)$

Sorting takes ⟶ $O(E \log E)$

Findsetandunion ⟶ $O(E)$

$= O(E \log E)$ $E \leq |V|^2$

Kruskal's Running Time is $O(E \log V)$