# ALGORITHMS:

- METHODS OF ANALYSIS:

① APOSTERIORI ANALYSIS:

☺ Reports exact values as it's experimental Analysis.

☹ Platform dependent.
- Dependent on H/W and S/W environment.

② APRIORI ANALYSIS:
- Independent of H/W and S/W environment

Framework:
(i) Language for Algo — Pseudocode.
(ii) A Computation model.
(iii) A metric for comparision.
(iv) Representation of Complexity (Asymptotic Notations).

- Tractable Problems → Polynomial Time.
- Intractable Problems → Exponential Time.

- TYPES OF ANALYSIS / BEHAVIOUR OF ALGORITHM.

① WORST-CASE:
The input class for which the Algo. takes maximum time.
Eg: Quick sort behaves in worst case '$O(n^2)$' when the list is already sorted.

② BEST-CASE
The input class for which the Algo. takes minimum time.
E.g: Quicksort behaves in best case when the list is unsorted. $O(n \log n)$.

③ AVERAGE-CASE:
Derived in 3 steps.
(i) Define all possible input classes.
(ii) Determine take taken by algo. for all input classes.
(iii) Define prob. of choosing an input class, $\forall$ input classes.

$K$ = no. of i/p classes
$t_i$ = time taken by Algo for $i^{th}$ i/p class.

$$A(n) = \sum_{i=1}^{K} P_i * t_i$$

## ASYMPTOTIC NOTATIONS:

① BIG-OH ($O$): upperbound : $f(n) = O(g(n))$ iff $f(n) \leq C \cdot g(n)$

② BIG-OMEGA ($\Omega$): Lower Bound : $f(n) = \Omega(g(n))$ iff $f(n) \geq C \cdot g(n)$

③ BIG-THETA ($\Theta$): TIGHT Bound : $f(n) = \Theta(g(n))$ iff $C_1 g(n) \leq f(n) \leq C_2 \cdot g(n)$

④ Small-OH ($o$): Proper Upperbound : $f(n) = o(g(n))$ iff $f(n) < C \cdot g(n)$

⑤ Small-OMEGA ($\omega$): Proper LowerBound : $f(n) = \omega(g(n))$ iff $f(n) > C \cdot g(n)$

Subset of

- Properties of Asymptotic Notations:

① If $f(n) = O(g(n))$, then $a \cdot f(n) = O(g(n))$ $\forall a > 0$.

② If $f(n) = O(g(n))$ & $d(n) = O(e(n))$, then
(i) $f(n) + d(n) = \max(O(g(n)), O(e(n)))$.
(ii) $f(n) * d(n) = O(g(n) * e(n))$.

③ $\log n^x = O(\log n)$
④ $(\log n)^x = O(n^y)$
⑤ $n^x = O(a^n)$
$\forall x, y > 0$, $\forall a > 1$.

Generally,
$B(n) \leq A(n) \leq W(n)$

I. $B(n) = A(n) = W(n)$
→ Merge, heap, Selection sort.

II. $B(n) < (A(n) = W(n))$
→ Linear Search / Binary Search / Insertion sort.

III. $(B(n) = A(n)) < W(n)$
→ QuickSort.

• Dominance Relation: Constants $<_A$ Logarithmic $<_A$ Polynomic $<_A$ Exponential.

• Some Discrete Properties:

| | $O$ | $\Omega$ | $\Theta$ | $o$ | $\omega$ |
|---|---|---|---|---|---|
| ① Reflexive → | ✓ | ✓ | ✓ | × | × |
| ② Symmetric → | × | × | ✓ | × | × |
| ③ Transitive → | ✓ | ✓ | ✓ | ✓ | ✓ |
| ④ Transpose Symmetry → | If $f(n)=O(g(n))$, then $g(n)=\Omega(f(n))$ | | — | If $f(n)=o(g(n))$ then $g(n)=\omega(f(n))$ | |

NOTE: Asymp. Notations do not obey tricotomy property.

SPACE COMPLEXITY: It is the workspace required by the algorithm.

- WorkSpace refers to the Computational Space excluding the space allocated to hold the input.

Let $S(n)$ and $T(n)$ represent respectively the space & time Complexity then, $\boxed{S(n) = O(T(n))}$ is always valid, i.e., $S(n) \leq T(n)$.

- Stack is a computational Space.

NOTE: Only Polynomial time & space algo. is considered to be efficient.
→ An algo is space efficient, if space complexity $O(1)$ or at most $O(n)$.

① Concat. of two list in $O(1)$ time → Circular doubly linked list.
② Min-Gate-delay for any op to appear in the an array multiplier for multiplying 2 (n-bit) nos is $O(n^2)$.
③ Transitive closure is like finding solution to all pair shortest path which takes $O(n^3)$.
④ If Input is in ascending order (already sorted), then, Quick-selection sort behaves in the worst case $O(n^2)$ and Insertion sort takes $O(n)$ time in Best Case.
⑤ In a sorted doubly linked list, decrease key takes $O(N)$ to decrease the key & sort the LinkedList again.
⑥ Time complexity to uniquely determine a Binary tree using & postorder is $O(n)$

**Divide & Conquer:** Here, Divide is always compulsory but to Conquere is optional depending upon the problem.

```
Procedure DnC (A,P,Q){
    if(Small(P,Q)) then
        return (G(P,Q));
    else
        Divide (P1,P2,.... Pk)
        return(Combine(DnC(P1),
            DnC(P2)..... DnC(Pk)));
}.
```

- All DnC problems can be solved Recursively.
- It is never mandatory to solve all k-subproblems atleast one subproblems must be solved.

**TIME COMPLEXITY:**

① $T(n) = a\,T(n/b) + f(n)$
for Symmetric case (when size of subproblems are same)
Here a = no. of subproblems.
b = parts of — " —.

② $T(n) = T(\alpha n) + T((1-\alpha)n) + fn$.
for assymmetric case.
where $\alpha > 0$ but $< 1$.

③ $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + \dots + f(n)$ for assymmetric case
but $\alpha + \beta + \gamma + \dots \neq 1$.

**CASE STUDIES FOR D&C:**

① **Max-min:** Finding the max & min element of an array of size 'n'.

(a) N-DnC approach:
(i) Set a max & min variable initialized to suitable values.
(ii) Apply linear search and assign max & min to the variables.

∴ Time Complexity = O(n) in any case.
& Space Complexity = O(1) — " —

(b) DnC Approach.

$T(n) = 0$ for $n=1$; $1$ for $n=2$. } n is small.
$T(n) = 2T(n/2) + 2$, $n > 2$.

∴ Time complexity = O(n) in any case
& Space Complexity = O(log n) — " —

NOTE: No. of Recursive calls gives Time Comp. & depth of Recu. Tree gives S.C.

② **Merge-Sort:** Based on principle of merging and conquering.

$T(n) = c$ for $n=1$ → To divide
$T(n) = 2T(n/2) + bn$ — To merge (conquere).

NOTE: To merge only, the time Complexity is O(n).

∴ Time Complexity = $O(n \log n)$. & Space Complexity = $O(n + \log n) = O(n)$
(n: new Array, log n: Stacle.)

③ BINARY SEARCH PROBLEM: Here, Conquering (Combining) is not require. Only divide the problem set and solve only a particular set each time (leaving the others).

$T(n) = c$ when $n = 1$.

$T(n) = T(n/2) + a$ , $n > 1$

∴ Time Complexity = $O(\log n)$

Here, SPACE COMPLEXITY depends upon implementation.

(i) For Iterative, SC = $O(1)$.

(ii) For Recursive, SC = $O(\log n)$.

④ QUICKSORT (Partition Exchange Sort a.k.a. Hoare's Technique).

- Partitioning/Pivoting Process:

(i) It involves selection of a Pivot (partitioning element) in the given list. Generally, the first element is selected as Pivot.

(ii) Partitioning process accomplishes the following activities:

(a) Placing the pivot to its correct place as it should lie in the final sorted list.

(b) Move all elements less than pivot to its left known as left Partition.

(c) Move all elements greater than pivot to it right known as right partition.

(iii) The partitioning process recursively continues on the left and right partitions until the size of partition reduces down to 1 element where upon the list becomes sorted.

Worst Case
- when pivot is either at first or last position.

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

$T(n) = 1$ , $n = 1$.

Time Complexity = $O(n \log n)$ for unsorted → SC = $O(\log n)$

Time Complexity = $O(n^2)$ for sorted (either asc or dsc) → SC = $O(n)$

⑤ Matrix Multiplication:

(a) Non-D&C.

- Matrix addition takes $O(n^2)$ time but Matrix multiplication takes $O(n^3)$ time.

(b) D&C Mat. Mult.:

$T(n) = c$ , $n = 1$

$T(n) = 8T(n/2) + bn^2$ , $n > 1, b > 0$.

∴ Time Complexity = $O(n^3)$

i.e, same as Non-D&C Mat. Multiplication

(c) STRASSEN'S MATRIX MULTIPLICATION:

$T(n) = c$ , $n = 1$

$T(n) = 7T(n/2) + bn^2$ , $n > 1, b > 0$

∴ Time Complexity = $O(n^{2.81})$ better than D&C strategy.

## MASTER's Method for Solving D&C Recurrences:

Let $T(n) = a \cdot T(n/b) + f(n)$ ; $a \geq 1$, $b > 1$.

**Case 1:** If $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$, then.

$T(n) = \Theta(n^{\log_b a})$.

**Case 2:** If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$,

then $T(n) = \Theta(f(n))$.

**Case 3:** If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for any $k$.

- (i) If $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$
- (ii) If $k = -1$, then $T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$
- (iii) If $k < -1$, then $T(n) = \Theta(n^{\log_b a})$.

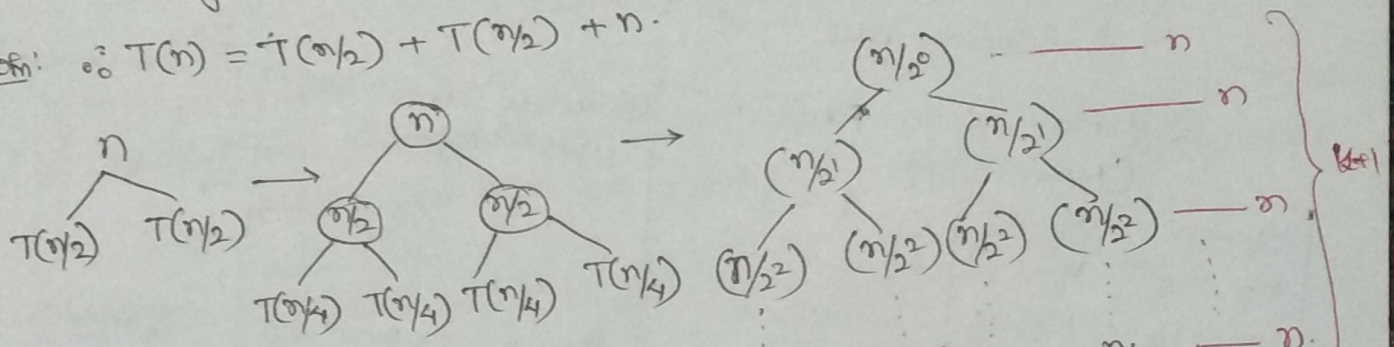
- D&C Recurrences can be solved using:
  1. Back Substitution – Gives value + order.
  2. Master's Method – (Mainly used for Symmetric) – Gives only order.
  3. Recursion Tree – (Mainly used for Asymm) – Gives only order.

---

## Recursion Tree method for Solving Recurrences:

(1) Solve Merge Sort problem using Recursion Tree: $T(n) = 2T(n/2) + n$.
$T(n) = c$ , $n = 1$.

Soln: $\therefore T(n) = T(n/2) + T(n/2) + n$.




Now, $T(n/2^k) = O(c)$

$\therefore n/2^k = 1$ $\therefore k = \log n$.

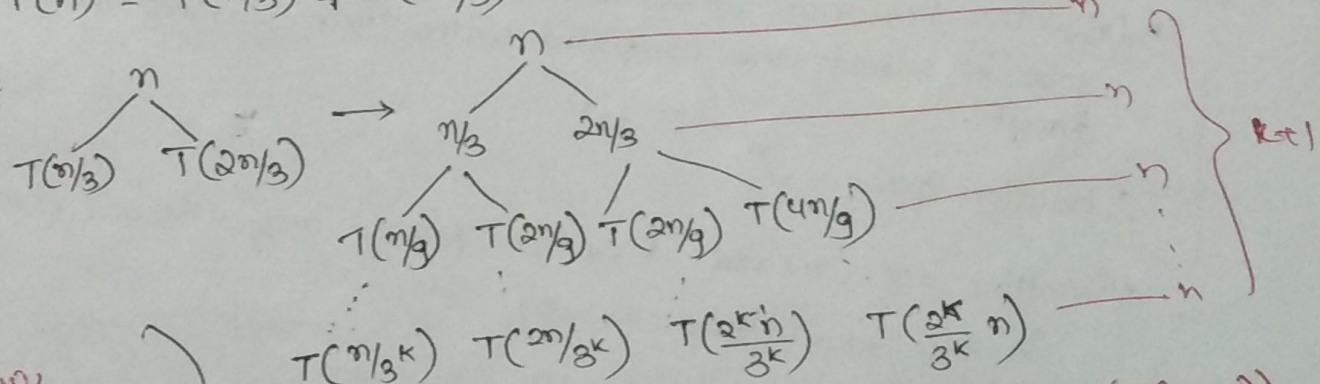
Total $= n(k+1)$

$\therefore T(n) = O(n(k+1)) = O(n \cdot (\log n + 1)) = O(n \log n)$.

---

(2) $T(n) = T(n/3) + T(2n/3) + n$

Soln:




Now,

$\frac{2^k}{3^k} n = 1$

$\Rightarrow \left(\frac{2}{3}\right)^k n = 1$

$\therefore k = \log_{3/2} n$.

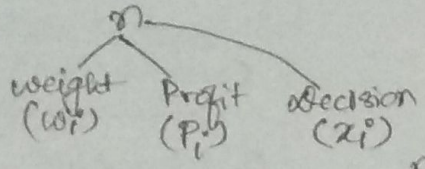
$\therefore T(n) = O(n(k+1))$

$= O(n \cdot (\log_{3/2} k + 1))$

$= O(n \log n)$

# GREEDY METHOD:

## ① Knapsack Problem:




- Explicit Constraints : $\sum_{i=1}^{n} w_i \geq M$ [Given that sum of all weights is more than knapsack capacity].
- Objective function : To maximize $\sum_{i=1}^{n} P_i x_i$ [Profit should be maximized based on decision]
- Subjected to $\sum_{i=1}^{n} w_i x_i \leq M$. [weight based on decision should not exceed the max. capacity of knapsack]

- when $0 \leq x_i \leq 1$, then it is known as Greedy/ fractional/ Real knapsack.
  ∴ Solution Space = $\infty$.
- when $x = 0/1$, then it is known as Binary knapsack.
  ∴ Solution Space = $2^n$.

ⓐ Greedy about profit.
(Most Profit)

| | |
|---|---|
| $w_1 = 18$ | $P_1 = 25$ |
| $w_2 = 15$ | $P_2 = 24$ |
| $w_3 = 10$ | $P_3 = 15$ |

$x_1 = 1$, $x_2 = \frac{2}{15}$, $x_3 = 0$

∴ $\sum w_i x_i = 20$

$\sum P_i x_i = 25 + \frac{48}{15} + 0 = 28.2$

ⓑ Greedy about weight
(least weight)

$x = 0$, $x = \frac{10}{15}$, $x = 1$ ∴ $\sum w_i x_i = 20$

$\sum P_i x_i = 15 + 24 \times \frac{10}{15} + 0$
$= 31.$

ⓒ** Greedy about $P_i/w_i$:
(By default)

$\frac{P_1}{w_1} = 1.88$ $\frac{P_2}{w_2} = 1.6$ $\frac{P_3}{w_3} = 1.5$

$x_1 = 0$ $x_2 = 1$ $x_3 = \frac{5}{10}$

∴ $\sum w_i x_i = 20$

$\sum P_i x_i = 24 + 7.5$
$= 31.5$

## ② Job Scheduling with deadlines: (CPU Scheduling)

- Single CPU with non-preemptive Scheduling
- OBJECTIVE FUNCTION: "Select a subset of given 'n' Jobs, such that jobs in subset are completed within their deadlines with maximal profit".

∴ Solution Space = $2^n$ → no. of jobs.

- Deadline = x, means it can be placed any where from 0 to x in the Gaant Chart.

- JSD (working like insertion sort) taking an order of $n^2$ time.

## (3) Optimal Merge Pattern:

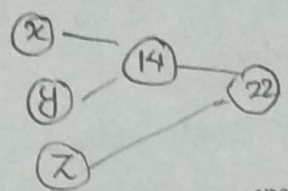
→ Merging of two files in a sorted fashion.

∴ Total no. of record movements = $n+m$ (n = no. of elements in $F_1$, m = no. of elements in $F_2$)

- Generally, the problem is defined for n files to be merged.
- Aim is to select a sequence where no. of record movements is Optimal (least).

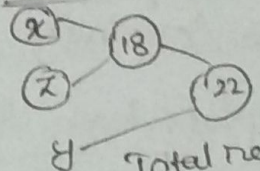
Eg: 3-files, with $f_x = 10$ records, $f_y = 4$ and $f_z = 8$ records.

Case (i)




Total rec. movements = 14+22 = 36

Case (ii)




Total records movement = 18+22 = 40.

Case (vi) . . . . . = 17+21 = 38

Time Complexity:
using LL → $O(n^2)$
using min-heap → $O(n \log n)$

Space Complexity: $= O(n)$.

(*) Total Comparisons = 13+21 = 34.

The min. no. of record movement is selected, i.e, 34 (yzx).

∴ Solution Space = $n!$.

NOTE: Only this problem has a feasible Soln space. No other Greedy Approach (Knapsack, N-Queens, etc.) has a feasible Soln Space.

SHORTCUT: Add least no. of records first to get Optimum Soln.

---

## (4) Huffman Coding: Data Encoding + Compression.

- Data Compression is never possible using uniform Encoding.
- Non-Uniform Encoding: Different no. of bits are used to encode different elements based on their frequency of occurance.

→ The elements that occur more frequently are encoded with less no. of bits.

---

## (5) Minimum Cost Spanning Tree:

- A Subgraph $T(V, E')$ of $G(V, E)$ where $E' \subseteq E$ is a spanning tree iff T is a tree (i.e, no cycles).
- Solution Space = $n^{n-2}$ (for $K_n$ only)
- Spanning tree is used to efficiently implement Multicast/Broadcast in Any Kind of network (Homogeneous / Heterogeneous Network).

(i) PRIM'S ALGORITHM: Select a node and Go greedy for min. weights.

- It always follows the property of a tree (not forest).
- Time Complexity to generate MST = $O(n^2)$. ← Adj. Matrix
  = $O((n+e)\log n)$ → sparse
  = $O(n^2 \log n)$ → dense
  ] Heap. Algorithms.

(ii) KRUSKAL'S ALGORITHM: Go greedy for edges & combine till no cycle is formed.

- It always follows the property of a forest (not tree).

Time Complexity = $O(e \log e)$.

NOTE: If a graph has distinct cost edges, then,

- MST by Prim's = MST by Kruskal's.

(iii) DIJKSTRA'S ALGORITHM: (to find MST).

Step 1: Start with any edge.
Step 2: Keep adding edges until a cycle is formed.
Step 3: Delete the edge with max. weight in cycle & Continue.

Time complexity = $O(n.e)$

---

⑥ SHORTEST PATH:

ⓐ Single Pair Shortest path: Distance between two points is computed.

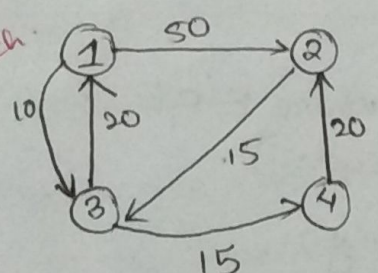
- Can be solved using A* search algorithm.

ⓑ Single Source Shortest Path: Shortest path from a fixed source to all other vertices is computed.

(i) Dijkstra's Algorithm: works only for +ve edges.

- Matrix-Method: for directed/undirected Graph.

Greedy Approach




$V_0 = 1$.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| (1) | - | 50 | (10) | ∞ |
| {1,3} | - | 50 | (10) | (25) |
| {1,3,4} | - | (45) | (10) | (25) |
| (1,3,4,2) | - | (45) | (10) | (25) |

→ Relaxation can only be carried for unselected vertices.

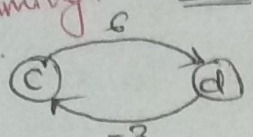
- MST-Method: for undirected Graph only.

☹ → Cost to other nodes is known, it doesn't show the stepwise path.

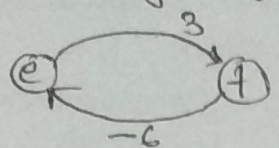
(ii) Bellman-Ford Algorithm: Can work with -ve edge weights but not -ve edge (wt.) cycles.

↙ Dynamic Programming Approach.

NOTE: No. algorithm can work with -ve edge cycles.




+ve weight cycle.




-ve weight cycle.

- The shortest path to the vertices in the -ve weight cycle or those vertices that are only reachable along with the -ve wt. cycle cannot be computed.
- BF Algo (based on principle of optimality) is used to explore all possible paths that a pair of vertices can have.
- Here, relaxation is done $\leq n-1$ times.

- Time Complexity = $O(n \cdot e)$ → same as dijkstra's Algo.
  For Complete Graph, $e = O(n^2)$ ∴ Time Complexity = $O(n^3)$.

---

(c) All Pairs Shortest Paths:

(i) Can be solved using single-source shortest path (Greedily) for n-times & changing the source n-times.

(ii) Floyd-Warshall-Algo: Let $A^k(i,j)$ represent the cost of the path from vertex 'i' to vertex 'j' with 'k' being the highest intermediate vertex between them.

↙ Dynamic Prog. Approach.

$i \xrightarrow{(k-1)} k \xrightarrow{(k-1)} j$

$$A^k(i,j) = \min_{1 \leq k \leq n} \{ \underbrace{A^{k-1}(i,k) + A^{k-1}(k,j)}_{\text{going through k.}}, \underbrace{A^{k-1}(i,j)}_{\text{not going through k.}} \}.$$

$$A^0[i,j] = C[i,j]$$

where, $C[i,j]$ = value in the cost matrix directly from i to j.

- Time Complexity = $O(n^3)$

NOTE: It helps to find transitive closure of a matrix.

---

NOTE: Shortest path between any two vertices is not always unique (even if the weights in graph is distinct).

- DFS can never be used to find shortest path.

→ (Single Source Shortest Path)

**Multistage Graph:** → Exactly 1 vertex at 1st & last stage and the remaining intermediate stage will have atleast 1 vertex
→ ∈ Vertices in Stage 'i' is adjacent to Stage 'i+1' only.
→ Most of the time Greedy/Dijkstra's Algo do not works, So, we can use either—

(i) Brute-force technique: — Enumerate the complete Solution Space
— Then find Optimum solution.
— Excessive time and Space Consumption.

(ii) By using Dynamic Programming:

**DYNAMIC PROGRAMMING:** Based on enumeration, often tries to cut down those decision sequences which will never lead to opt-soln. The sequence of decisions in D.P. are made by applying principle of optimality.

- **PRINCIPLE OF OPTIMALITY:** It states that whatever the initial state and decisions are, the remaining sequence of decision must be optimal.
- By Applying Prin. of Opt., A problem is divided into overlapping subproblems (in D&C the subproblems are independent) and the solution is expressed as a recurrence which is solved in bottom-up fashion.

E.g: fibonacci series : $T(n) = T(n-1) + T(n-2) + c = O(2^n)$
∵ The values of recursion is retained (preserved) as a feature of Dynamic programming.
∴ Time Complexity becomes $O(n)$.

**(1) DP on Multi-stage Graph:**

Let $C[i,j]$ be the edge cost between $V_i$ & $V_j$.
Let $COST[i,j]$ refers to cost of path from vertex 'j' in Stage 'i' to destination vertex 't';

$$\therefore COST[i,j] = \min_{\substack{k \in S_{i+1} \\ \langle j,k \rangle \in E}} \{ C(j,k) + COST[i+1,k] \}$$

$$COST[k-1,j] = c(j,t)$$

② TRAVELLING SALESMEN PROBLEM: → NP Complete Problem.

→ To find min. trip path from $V_0$ through all other nodes back to $V_0$.

→ Let $G[i, S]$ represent the cost of tour 'i' visiting all other vertices in set S' exactly once & terminating the tour at $V_0$.

$$G(i, \{S\}) = \min \{ C(i,k) + G(k, S - \{k\}) \}$$

$$G(i, \emptyset) = C(i, V_0)$$

Time Complexity = $O(n^2 \cdot 2^n)$ Space Complexity = $O(n \cdot 2^n)$.

③ Floyd-Warshall Algorithm.

④ Matrix-chain Product (MCP):

Generally, No. of scalar multiplications = $n^3$

∴ Time Complexity = $O(n^3)$ where n = order of matrices.

→ • To fully parenthesize a given chain of matrices such that the no. of scalar multiplication involved in getting the product matrix is minimum.

• Let $A_{i \ldots j}$ represent the product matrix obtained by multiplying the chain, i.e, $A_{i \ldots j} = A_i \times A_{i+1} \times A_{i+2} \times \ldots \times A_j$

• Let $M[i,j]$ represent the no. of scalar multiplication involved for obtaining matrix $A_{i \ldots j}$!

→ Any Optimal parenthesis must split the given chain of matrices between $A_k$ & $A_{k+1}$ such that the no. of multiplication is minimum.

$$M[i,j] = \min \{ M[i,k] + M[k+1, j] + P_{i-1} \cdot P_k \cdot P_j \} \quad \text{for } i \le k \le j-1$$

$$M[i,j] = 0 \quad \text{otherwise.}$$

∴ Time Complexity = $O(n^3)$ where n = no. of matrices.

⑤ Longest Common Subsequence: Every Substring ⟹ Subsequence

- For a string of length 'n', # Substrings $= \frac{n(n+1)}{2} = O(n^2)$
  # Subsequence $= 2^n = O(2^n)$
- Subsequences: A group of one or more characters of a string, that need not be contiguous but must be in order.

→ Given two strings, x & y each having n & m characters respectively. The problem of LCS is to determine a common subsequence of x & y of same length.

→ Let i & j be indices into the string x & y as shown below

$X = \langle x_0^0, x_1^1, x_2^2, x_3^3 \ldots\ldots x_n^i \rangle$

$Y = \langle y_0^0, y_1^1, y_2^2, y_3^3 \ldots\ldots y_m^j \rangle$

→ Let L[i,j] represent the length of the common subsequence of the string x & y as defined below:

CASE 1: X[i] = Y[j] then

$L[i,j] = 1 + L[i-1, j-1]$

CASE 2: X[i] ≠ Y[j]

$L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$.

Termination Condition: $L[-1,j] = 0$ when $i = -1$

$L[i,-1] = 0$ when $j = -1$.

Time Complexity = Space Complexity = $O(m \times n)$

⑥ 0/1 Knapsack:

$$f_n(m) = \max\{\underbrace{P_n + f_{n-1}(M - W_x)}_{x_n = 1}, \underbrace{0 + f_{n-1}(M)}_{x_n = 0}\}$$

$f_0(x) = 0$.

## GRAPH TECHNIQUES:

Tree traversal is always unique but Graph traversal is not.

① Exploring / Expanding Node (E-Node) → The node that is currently being explored or whose children is being generated.
→ There can be only one enode during traversal.

② Live node: Node that has not been fully explored.
→ There can be many live nodes during traversal.
→ BFS uses Queue to store live nodes & DFS uses stack to store live nodes.

③ Dead node: Node that has been fully explored.

Always +ve integers (>0)
- Discovery Time: d(x): The time at which the node is visited for first time.
- Finishing Time: f(x): The time at which the node becomes dead.
  → used to find Topological Sort in DFS of DAGs

### DEPTH FIRST SEARCH (DFS):

① For undirected Connected Graphs :→ DFS Spanning Tree.
② For undirected Disjoint Graphs :→ DFS Spanning Forest (a.k.a Depth first traversals (DFT)).

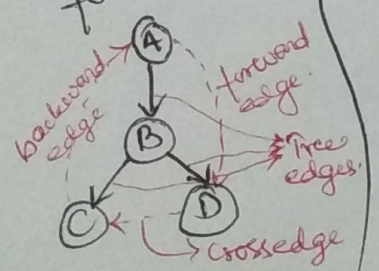
NOTE: Discovery time > Finishing time → Not a valid graph.

③ For Directed Graphs: → Either DFS Spanning Tree or Forest.
→ In the DFS Spanning tree/forest, the following edges may be present: (i) tree-edge :→ Part of DFS Spanning Tree/forest.
(ii) Forward-edge: Leads from a node to its non-child descendent.
(iii) Backward-edge: Lead from a node to its ancestors.
(iv) Cross-edge: Leads from a node to another node that is neither its ancestor nor its descendent.

④ For DAGs: → DFS in DAG leads to topological sort/ topological linear order.

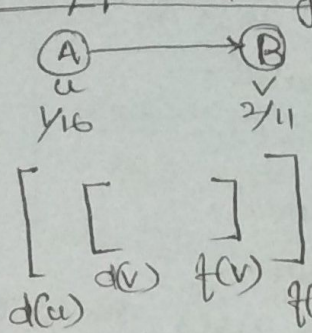
⇒ Algo Topo_Sort {
① DFS(V);
② Arrange the nodes in decr order of finishing time.
}.

Application of d(x) & f(x):
(i) Topology of Graph.
(ii) Topological Sorting.
(iv) Identify types of edges in DFS Span. Tree/ forest

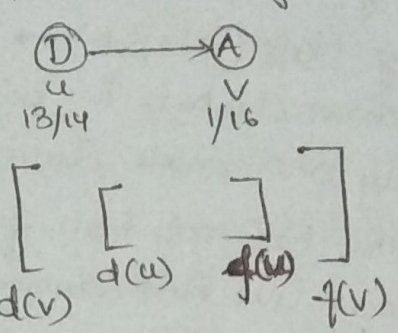


## Parenthization Theorem:

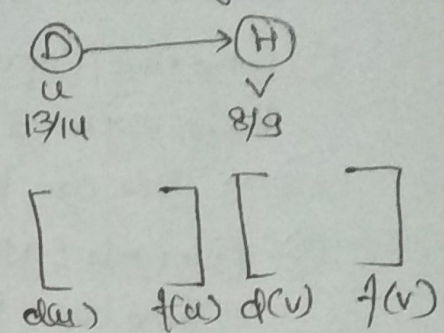
(i) Tree / forward edges.




(ii) Backward edge:




(iii) Cross edge:




**Breadth First Search:** Here, Queue DS is used to store live nodes.

→ Generally Queue behaves in FIFO manner. But when it behaves in LIFO manner it is called DFS using BFS.

## Properties or Application of Graph traversals:

(1) Time Complexity of DFS/BFS depends on the representation of the Graph.
- Adjacency matrix : T.C. $= O(n^2)$
- Adjacency list : T.C $= O(n+e)$.

(2) Both BFS and DFS can be used to determine/detect
- (i) A connected/disconnected graph.
- (ii) Cycle in a graph. (presence of back-edge).
- (iii) Connected/Strongly connected/Biconnected Components
- (iv) Articulation points & bridges.

(3) BFS can be used to generate min. Spanning Tree, but DFS cannot be used " " MST.

(4) DFS is used in implementing Backtracking design and BFS is used to implement Branch & Bound strategy.

(5) If a graph has unit weight edges, the BFS acts as the most efficient algorithm for solving the problem of single source shortest path.

# HEAP ALGORITHMS

- **Heap** → used to implement Priority Queue.
  ↳ Array Based implementation in the form of Binary Tree

(i) Min-Heap : Value of every node is lesser than all value of its child.
(ii) Max-heap : ——— greater than ———

(A) **Insertion Method**: Create a tree of n-elements starting from an empty tree and Insert 1 element at a time and adjust (if reqd).

Best Case, $T(n) = O(n)$ | Worst Case, $T(n) = O(n \log n)$

Insert Operation in Heap takes : $T(n) = O(1)$ → Best Case.
$T(n) = O(\log n)$ → Worst Case.

(B) **Build-heap or Heapify Method**:

→ Adjust the nodes level by level from top to down, by comparing it with the value of its largest child.

Time Complexity $= O(\log n)$ | Space Complexity $= O(n)$

---

Algo Heap_Sort {
1. Heapify (A, n) —— $\log n$.
2. for i ← n to 2 by −1 { —— n
   Swap (A[1], A[i]);
   Adjust [A[1]); —— $\log n$.
   }
}.

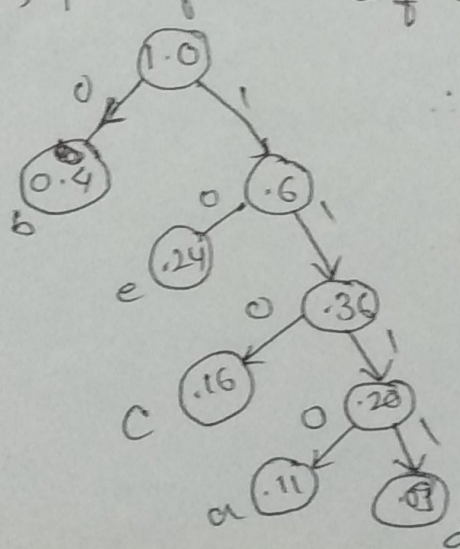
T.C = $O(n \log n)$.

- **Extract Max-element from Max-Heap**
→ Extract the element and replace it by last element and adjust.
T.C = S.C = $O(\log n)$.
- Increase / Decrease key.
T.C = $O(\log n)$.

---

## Points to Note:

(1) Given, prob of occurance of a = 0.11, b = 0.40, c = 0.16, d = 0.09, e = 0.24.




∴ Avg. length of huffman coding
= 1×0.4 + 2×0.24 + 3×0.16 + 4×0.11 + 4×0.09
= 0.40 + 0.48 + 0.48 + 0.44 + 0.36
= 2.16