Teacher's Sign / Remarks

ALGORITHM

Algo:- Set of steps to complete a task, described precisely enough such that computer can run it.

Ex:

GPS - shortest path

shopping apps - RSA

Characteristics:-

* Definiteness
* Finiteness
* Effectiveness
* Efficiency.
* 1/more inputs
* 1 output

Analysis:

Process of evaluating efficiency of algorithms focusing on time, space complexity.

⇒

	NOI/s	RT	steps
A:	10 billion $= 10\times10^9 = 10^{10}$	n^2	$2n^2$
B:	10 million $= 10\times10^6 = 10^7$	$n\log n$	$50\ n\log n$

billion = 10,00,000 $= 10^6$

billion = 10,00,00,000 $= 10^9$

$N = 10^7$ (array size)

$$\text{Time taken} = \frac{\text{No. of steps } (N)}{\text{Speed (NOI/s)}}$$

For A:- $\frac{2n^2}{10^{10}} = \frac{2\times(10^7)^2}{10^{10}} = 2\times10^4 = 20000\ s$

B. $t = \frac{50 n \log n}{10^7} = \frac{50 \times 10^7 \times \log_2 10^7}{10^7} = 50 \times 23.2534$

$= 1162.67$ s

$= 350$ s

$\therefore$ B has ↑ efficiency than A.

$\because$

Computational Tractability: Easy to deal.

Factor used to access/identify if specific problem statement at hand can be solved by an algorithm.

Approach:- Analyze before running code.

Platform independent: does not depend on computer capacity.

Instance Independent: does not depend on input size

Function of input size N :- $f(N)$

RT:-

Worst cases:-

↑ Time

Avg case:-

Expected time.

Best case:-

Swiftest time.

Brute force search:

Search solution from entire search space

RT bound:

Upper, lower limit on time complexity, as $F(N)$.

Upper → $O(f(N))$ i.e order of ( )

Lower → $O(\Omega)$ Ω → Omega.

Tightest → $O(\Theta)$

Models of computation:

1. Turing machine:

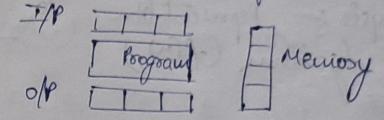
Linear search

Caveat:- Limitations

No RAM

2. Word RAM:

Random access machine does arithmetic, bitwise operations on word of w bits.



Caveat:- ↑ input won't work.

3. Brute force search:

Ex:- Search needle in haystack.

Check every possible solution.

scaling property:

$(\text{Max step})^{RT} \propto cN^d$, c, d – constants

N – input size

d – scaling factor exponent

c: No. of steps

d: Growth rate.

Middle school :
Prime factorization

Euclid's algorithm :- gcd(m, n) = gcd(n, m%n).

Integer checking : smallest no. than m, n.
Increment.

Time complexity :- <ctime>, <time.h>

Iteration : no stack (↓ time)
Recursion : stack (↑ time)

Asymptotic order :

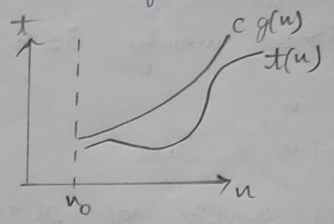
Asymptotic analysis :- Representation to find time complexity of algorithm (T(n))

Order of growth :
Way to measure how algorithms RT ↑ as Input size ↑

Asymptotic notations :- Mathematical way of representing time complexity.

notations :-
1. O - Big oh
2. Ω - Big omega.
3. Θ - Big theta.
4. o - small oh
5. ω - small omega.

N - input size
f(n) - growth of N wrt time
T(n) - time complexity
Any algorithm's worst case RT
C(n) - Basic operation count of algorithm
g(n) - Simple function to compare count order of magnitude.



Asymptotic Upper bound:
O(big oh)

$t(n) \in O(g(n))$
order of g(n)

c(g(n)) is upper bound for t(n). for n beyond n_0

Function g(n) is upper bound for t(n), if beyond some point, g(n) dominates t(n).

g(n) is function which describes order of magnitude.

g(n) not only dominates t(n), but some times of t(n).

* C is fixed constant.

Beyond n_0, t(n) lies below cg(n).

∴ cg(n) is upper bound of t(n).

$t(n) \in O(g(n))$

1) Prove $f(n) = 100n+5$ is $O(n^2)$ using Big O notation.

To prove :- $f(n) = O(n^2)$ i.e, find c, n_0 s.t $f(n) \leq cn^2$ $(\forall n \geq n_0)$

Given :- $f(n) = 100n+5$

Let $c=1$

For $n \geq 5$,

we modify $f(n)$ as

$100n+5 \leq cn^2$

$\Rightarrow 100n+5 \leq 100n+n$

$\Rightarrow 100n+5 \leq 101n$

$\Rightarrow n^2 - 100n - 5 \geq 0$

$\Rightarrow n = 100.05 = 101$

$= n_0$

($\because n \geq 5$ i.e $n \geq 5 > 1$ meaning $5 \leq n$)

$\therefore$ Replacing 5 with n,

$100n+n$ is larger

$\Rightarrow$ $100n+n$ can be simplified as $101n$.

$\because n^2 \geq n \quad \forall n \geq 1$,

we can replace n with n^2

$\Rightarrow 101n \leq 101n^2$

thus,

$100n + 5 \leq 101n^2 \quad \forall n \geq 5$.

Hence, to satisfy Big-O definition,

$f(n) \leq cn^2 \quad \forall n \geq n_0$

$\therefore c = 101$

$n_0 = 5$

$\therefore f(n) = O(n^2)$.

2) Consider $f(n) = 100n^2 + 20n + 5$. P.T order is $O(n^2)$ using Big O notation.

$f(n) \leq cn^2 \quad \forall n \geq n_0$

① Find upper bound.

n^2 term dominates

Express entire function as multiple of n^2.

$\Rightarrow 100n^2 + 20n + 5 \leq 100n^2 + 20n^2 + 5n^2$

$\leq 125n^2$

② Find n_0, c.

$c = 125$

Observing expression, $n \geq 1$.

$\Rightarrow n_0 = 1$.

$\therefore T(n) \in O(n^2)$.

LS :-

Best case :- $O(1)$

Worst :- $O(n)$

Average :- $O(n)$

BS :-

Best :- $O(1)$

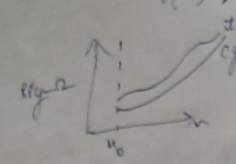
Worst :- $O \log_2(n)$

Avg :- $O \log_2(n)$ } Every step search space becomes half (Base 2 :: binary)

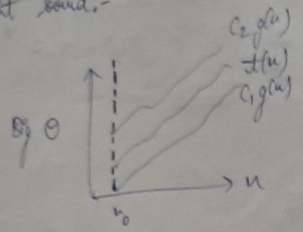
~~Tight Bound proof :-~~

Lower bound $\Omega(g(n))$

$f(n) \geq c\,g(n) \quad \forall n \geq n_0$



Tight bound:-



$C_1 g(n) \leq f(n) \leq C_2 g(n). \quad \forall n \geq n_0$

Proof:-

$f(n) = \Theta g(n)$.

Given: f, g are 2 functions such that.

$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and $c > 0$.

then, $f(n) = \Theta g(n)$.

If 2 functions $f(n), g(n)$ grow at same rate, as n becomes very large, then we say that $f(n) = \Theta g(n)$ ie $f(n), g(n)$ only differs by constant c, at 1 point, both converge at c.

Proof:

these eqns

$f(n) = O(g(n))$

$f(n) = \Omega(g(n))$

a limit and it is positive.

$\because \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$

Beyond n_0, ratio is btwn $\frac{1}{2}c$ and $2c$.

$f(n) \leq 2c\,g(n) \Rightarrow f(n) = O(g(n))$

$f(n) \geq \frac{1}{2}c\,g(n) \Rightarrow f(n) = \Omega(g(n))$ $\forall n \geq n_0$

By def of limits, calculus, for any +ve no, there exists large n_0 s.t. $\forall n \geq n_0$, the ratio of function stays within v small range C.

Properties:-

1. Transitivity:-

If we have 3 functions $f(n), g(n), h(n)$ we say that function is asymptotically upper bounded by function $g(n)$.

$g(n)$ is asymptotically upper bounded by $h(n)$ then, $f(n)$ is asymptotically upper bounded by $h(n)$.

$f(n) = O(g(n))$

$g(n) = O(h(n))$

$\Rightarrow f(n) = O(h(n))$

Proof:-

$f(n) \leq c\,g(n)$ ——— (1)

$g(n) \leq c'\,h(n)$ ——— (2)

$\exists\, c, n_0$ such that $\forall n \geq n_0$ eqn (1) satisfies

$\exists\, c', n_0'$ s.t $\forall n \geq n_0'$ eqn (2) holds

To prove transitivity we need to prove value of n s.t it is larger than both n_0, n_0'

for $n_0 = \max(n_0, n_0')$

Sub value of $g(n)$ in (3),

$f(n) \le c\, g(n)$.

$\le c(c' h(n))$

$\le cc' h(n)$

$\le C\, h(n)$

$\le O(h(n))$

1) Consider 3 functions $f(n) = 3n^2$, $g(n) = 5n^3$, $h(n) = 10n^4$. Prove property of transitivity.

$=$ $f(n) \le c\, g(n)$

$3n^2 \le c\, 5n^3$

$3 \le 5n$

Step 1:- check if $f(n) = O(g(n))$ i.e $f(n) \le c\, g(n)$ i.e find values for c, n_0 such that condition satisfies

$3n^2 \le 5n^3$

$n = 1, 2, 3, \ldots$

$n \ge 1 \quad \therefore n_0 = 1$

$5n^3 \le 10n^4$

$n = 1, 2, 3, \ldots$

$n \ge 1 \quad n_0 = 1$

$\Rightarrow 3n^2 \le 10n^4$ → ∴ transitivity satisfied.

∴ Growth Order $= n^4$

2. Sum of functions. → $f(n), g(n)$ both bounded by another function $h(n)$, then their sum is also bounded by $h(n)$.

$=$ $f(n) = O(h(n))$ — (1)

$g(n) = O(h(n))$ — (2)

$\Rightarrow f(n) + g(n) = O(h(n))$

Consider (1), $\exists\, c_1$ and n_0 $\forall\, n \ge n_0$

$f(n) \le c_1 h(n)$. — (3)

Consider (2), $\exists\, c_2, n_0'$ $\forall\, n \ge n_0'$

$g(n) \le c_2 h(n)$ — (4)

(3) + (4) $\Rightarrow$ $f(n) + g(n) \le c_1 h(n) + c_2 h(n)$

$\le (c_1 + c_2) h(n)$

$\le C h(n)$

1) $f_1(n) = 3n^2$

$f_2(n) = 5n^2$

$f_3(n) = 2n^2$

4. If 2 functions $f(n)$, $g(n)$ taking non-ve values such that $g(n) = O(f(n))$, then $f(n) + g(n) = O(f(n))$ i.e sum is asymptotically tight bound by $f(n)$

3. Generalized.

k – constant

$f_1, f_2, \ldots f_k, h$ – functions

$f_i = O(h)$

then, $f_1 + f_2 + \ldots + f_k = O(h)$

Asymptotic:-

≡ If $g(n)$ function grows same as $f(n)$ then, $f(n)+g(n)$ have same growth.

In other words, can define fastest growing functions that dominates.

Tractable → can be solved
Intractable → cannot be solved.

Asymptotic properties:-

1. Polynomial:

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0.$$

$$f(n) = a_0 + a_1 n + \dots + a_d n^d, \quad d > 0, \quad a_0 \neq 0$$

d.

order of growth = Highest order term in Big O.

2. Logarithmic:

Growth is smaller than very small polynomial functions.

$\log_b a \Rightarrow b^x = a$.

$\log_b n = O(n^x) \quad \forall b > 1, x > 0.$

3. Exponential.

1) Consider logarithmic function $\log_2 n$, $n^{0.01}$, compute, compare 2 functions for increasing value of n. P.T logarithmic function has slowest order of growth.

= For $n = 10$,

$$\log_2(10) = 3.3219$$

$$10^{0.01} = 1.0233$$

For $n = 100$

$$\log_2(100) = 6.6438$$

$$100^{0.01} = 1.0471$$

∴ Log function has ↑ growth order.

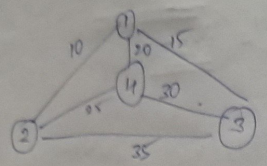
Every exponential funct^n outgrows polynomial function no matter how large polynomial exponent is for larges n values.

∴ Expo > Poly > Logarithmic

RT:-

$O(1)$	constant	Accessing array ele
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Max ele in array. Dividing into 2 halves. ← Quick sort in any, ↑ Merge sort/c ratio (can be half)
$O(n, \log n)$	Near linear	Bubble sort, Floyd Warshall.
$O(n^2)$	Quadratic	placing 1st target ele in correct pos, 2nd, 3rd, ... Matrix Multiplication.
$O(n^3)$	Cubic	
$O(2^n)$	Exponential	Recursive fibonacci
$O(n!)$	Factorial	Travelling salesmen problem.

TSP :-
Salesman used to find shortest route that visits all cities exactly once and return to starting city.



Stable matching problem :- Gale-Shapley algorithm.
Consider 2 sets $Y = \{m_1, m_2, \ldots m_n\}$, $X = \{w_1, w_2, \ldots w_n\}$. Each man has a ranking list of men. Each woman has ranking list of women.

Set of n pairs, where each pair is paired with exactly 1 woman. There should not be any blocking pairs.

Merge sort :-
TC = O(nlogn)
Step one it is divided into 2 half in every one is obtained.

Quick sort :-
Pivot :- Choose median (easy)
Partition :-
2 pointers one close / pointers leftmost, rightmost.
leftmost ++ until value is greater than pivot value.
rightmost -- until value is less than pivot.
Swap both.
Now pivot is at middle
Divide into 2 parts, again quick sort.

TC :- ~~nlog n~~ O(n log n) — pivot is at middle.
Best
worst → $O(n^2)$ → Division is uneven
Avg → O(n logn)

Continued....
Set of M pairs (m_i, w_j) where each pair is paired with exactly with 1 woman.
$M = \{(m_1, w_2), (m_2, w_3), (m_3, w_1)\}$
(1) (2) (3)

Blocking pair :- Unstable
A pair (m, w) is a blocking pair if
a) they are not matched in current pairing M, ie a man m, women w are not matched in M.
b) Man m prefers woman w more than the current partners.

(c)
Stable matching :- Matching is stable if there is no blocking pairs.

Instance of problem :-
2 sets of preference list is given or by ranking matrix where each man, women ranks acc. to preference.

Row represents men, columns represents Women

Men's preference:

	1st	2nd	3rd
Bob :	Lea	Ann	Sue
Jim :	Lea	Sue	Ann
Tom :	Sue	Lea	Ann

Matrix :-

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Step 0 :- At beginning all men, women are unpaired.

Step 1 :- Proposal
Bob. any free woman m.
. in proposes w

Response
(1) If women w is free, she might accept, get engaged.
(2) If women w is not free, she compares m with current mate. If she prefers m over current mate, she accepts m. she can also reject m.

TC :- It takes n^2 steps to ensure no blocking pairs
$\therefore O(n^2)$

Divide & Conquer : D&C

Merge sort :- MS

Input is reduced to 2 → $O(\log_2 n)$.
sort ele by comparing with each other → $O(n)$
Worst case → $O(n \times \log_2 n)$

Any D&C technique : BASE CASE FOR RECURSION
(Stopping condition) Condition at which algorithm is finished

Cases :- MS
Best :- Array already sorted. — $O(n \log_2 n)$.
Worst :- Array is in completely reverse order — $O(n \log_2 n)$
Avg :- All possible ways in which input is arranged.

D&C :- Reduce degree of polynomial.
Making it ↓ degree

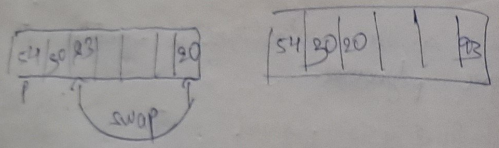
Recurrence relation :
RT as a function of its RT on smaller subproblems i.e representing RT as smaller degree polynomial.

Techniques to bound RT of Algorithm :-
(1) Recursion tree
(2) substitution method.
(3) Master's theorem

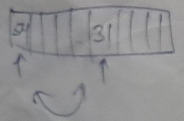
Quick Sort :-
1. Pivot : Element around which array is divided.
2. Index pointer : Tracks boundary where elements smaller than pivot should be placed.
3. Walker pointer :

Same array to identify elements smaller than/equal to pivot.
Later, swapped with index pointer ele

Algorithm:-
1. Pivot selection: Random ele to divide array in 2 parts
Median-3 technique: find median of 1st, middle, last elements
2. Partitioning: To make pivot ele, to fall exactly at middle.
3. Recursion: Repeat process on left, right subarrays until 1 element.



Base case: All small left, big right.



Efficiency:-
Quick > Merge.
In place sort ↑ memory space
Direct swapping
no subscript
↓ space.

DAC:-
Divide into multiple sub-problems
Solve each of them separately.
Merge, get required answer.

Reason:-
Brute force → Polynomial RT
DAC → Reduce this RT to lower polynomial.

Merge sort:-
1. $n=1 \Rightarrow$ Done (only 1 ele) $O(1)$
2. Recursively sort $A(1 \cdots \frac{n}{2})$, $A(\frac{n}{2}, \cdots n)$
3. Merge 2 sorted lists.

Recurrance relation:-
Relation involving repeated dividing and merging
Limits RT of algo recursively (∵ problem is divided) in terms of RT on smaller instances.

Master theorem:
Way to solve recurrance relations.

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d).$$

where, $T(n) \rightarrow$ Time complexity.
$a \rightarrow$ no. of subproblems into which original is divided.
$b \rightarrow$ reduction factor (factor by which problem size is reduced in each recursive call)

$O(n^d)$ → cost of merging, dividing at each level.

Cases:-

1. Divide dominates:-
$$d > \underbrace{\log_b a}_{\text{divide method}}$$
$T(n)$ order $= O(n^d)$.

2. Equal growth:-
$$d = \log_b a$$
$$\Rightarrow T(n) = O(n^d \log n)$$

3. Conquer dominates:-
$$d < \log_b a$$
$$\Rightarrow T(n) = O(n \log_b a)$$

Solve using Masters theorem

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

b) $T(n) = 8T\left(\frac{n}{3}\right) + n^2$

=

$T(n) \leq 8T\left(\frac{n}{2}\right) + cn$ where $n \geq 2$, $T(2) \leq C$

$T(n)$ → Worst case RT

n = I/P size.

I/P size is reduced to 2, recursion stops @ $O(\log n)$.

Sort elts by comparing them $O(n)$.

∴ Worst case → $O(n \log n)$.

∴ Cases:- Best → Array is sorted.

Worst → Reverse order

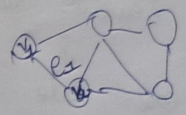
Avg → All possible ways in which I/P elts can be arranged

Merge sort :- $a = b = 2$

Graph:- collection of nodes/vertices & edges.

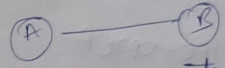
Pairwise relationships.

Vertices → V, edges → E



$e_1 = (v_1, v_2)$

Symmetric relationship : UNDIRECTED graph.



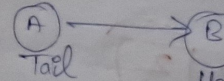
A ⇌ B, B ⇌ A.

Consider 2 vertices A&B. If there is an edge btwn A&B, A is connected to B, B is connected to A.

Asymmetric relationship:

Edges indicate 1 way relationship

DIRECTED graph.

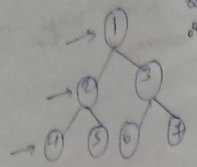


Ex:- Maps, LAN, Internet, Social Media, Project management Networks: Transport, Communication, Information, Social Dependency.

Path : sequence of nodes. denoted by P, where nodes = $\{v_1, v_2, \ldots v_k\}$

cycle : path where sequence of nodes comes back to where it began.

Breadth First Search :- Wide.
level by level.
Shortest path (1 node visit once)
Queue → STORE REFERENCE NODES
so we can go back.
1, 2, 3, 4, 5, 6, 7.



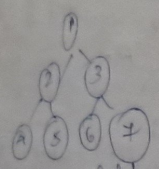
Ex:- Social media, Networking.

~~Depth First Search~~ Back tracking using empty list.

Traversal ends when all elements are traversed.

TC = O(n) : visiting each node once.
SC = O(n).

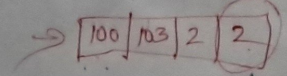
Depth First Search :- PRE-ORDER



1, 2, 4, 5, 3, 6, 7.

Ex:- Puzzles, Maze, Sudoku.

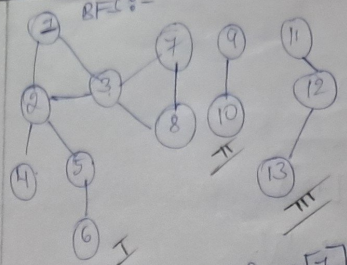
Check connectivity.



100	103	2	2
2	103	2	100
2	100	2	103
2	2	100	103

Rooted graph = Tree

1)



BFS :-

4 not 7
∵ 2 is first added compared to 3.
∴ complete 2's children first.

= start at 1 → queue [1]
{1} visited. ✓

visit 1 → neighbours → queue [2, 3]
{1, 2, 3} ✓

visit 2 → 1, 3, 4, 5 → queue [3, 4, 5].
{1, 2, 3, 4, 5}.

visit 3 → 1, 2, 7, 8 → queue [4, 5, 7, 8]
{1, 2, 3, 4, 5, 7, 8}.

visit 4 → 2 → queue [5, 7, 8]
{1, 2, 3, 4, 5, 7, 8}.

visit 5 → 2, 6 → queue [7, 8, 6]
{1, 2, 3, 4, 5, 7, 8, 6}.

visit 6 → 5 → queue [7, 8]
{1, 2, 3, 4, 5, 7, 8, 6}.

visit 7 → 3, 8. → queue [8]
{1, 2, 3, 4, 5, 7, 8, 6}.

visit 8 → 3 → queue [ ]
{1, 2, 3, 4, 5, 7, 8, 6}.

1 → 2 → 3 → 4 → 5 → 7 → 8 → 6.

start 9 → queue [9]
{9} ✓

visit 9 → 10 → queue [10]
{9, 10}.

visit 10 → 9 → queue [ ]
{9, 10}.

7 → 10

9 → 10

start 11 → queue[11]
{11}.

visit 11 → 12 queue[12]
{11, 12}

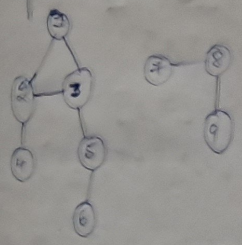
visit 12 → 11, 13 queue[13]
{11, 12, 13}.

∴ 11 → 12 → 13

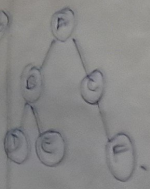
∴ 1 → 2 → 3 → 4 → 5 → 7 → 8 → 6 → 9 → 10 → 11 → 12 → 13

2) Implement BFS traversal :-

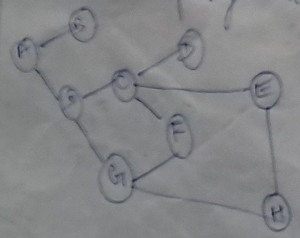
a)



b)



c) Output BFS spanning tree :-



2) a) start 1 → q[1]
{1}. ✓

visit 1 → 2, 3 q[2, 3]
{1, 2, 3}. ✓

visit 2 → 3, 4 q[4]
{1, 2, 3, 4} ✓

visit 3 → 1, 2, 5 q[5]
{1, 2, 3, 4, 5} ✓

visit 4 → 2 {1, 2, 3, 4, 5}.

visit 5 → 3, 6 q[6]
{1, 2, 3, 4, 5, 6}.

visit 6 → 5 {1, 2, 3, 4, 5, 6}.

start 7 → q[7]
{7}.

visit 7 → 8 q[8]
{7, 8}.

visit 8 → 7, 9 q[9]
{7, 8, 9}.

∴ 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

b) start A → q[A]
{A}.

visit A → B, C q[B, C]
{A, B, C}.

visit B → D, E q[D, E]

visit C → A, F {A, B, C, D, E}.

D → q[F]
{A, B, C, D, E, F}

E → " "

F → " "

d. A → B → C → D → E → F

3

DFS:- Go deep into path.

Connected graph:
Path btwn any 2 vertices

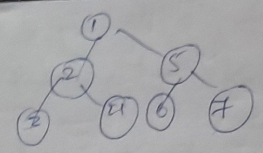
~~Disconnect 1 pair~~

Disconnected graph:
Atleast 1 pair of vertices have no path btwn them.

Strongly connected graph:
A directed graph is strongly connected if there is directed path btwn every pair of vertices. EVERY VERTEX IS REACHABLE FROM EVERY OTHER VERTEX

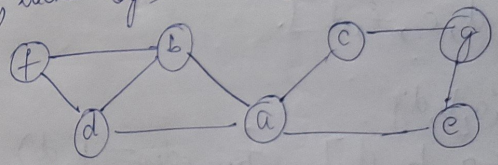
Weakly connected graph:
Directed graph where there is path btwn every pair of vertices, regardless of direction of edges

DFS:



1 → 2 → 3 → 4 → 5 → 6 → 7

1) Consider graph given in figure. No cycles
a) ~~Starting~~ at vertex A ~~resolving ties~~ by vertex order, traversal graph by ~~DFS~~.
b) Construct ~~BFS~~ spanning tree.
c) Show traces of traversal stack for ~~BFS~~ & specify order in which vertices are pushed onto & popped off from stack
d) Identify back edges if any.
e) "All DFS forests will have same no. of tree edges, back edges". True/False? Justify.



= Adjacency list:-
a: b, c, d, e.
b: a, d, f
c: a, ~~e~~, g
d: a, b, f
e: a, g
f: b, d
g: c, e.

starting vertex = a :-
visited - {}
stack - [ ]
DFS trees edges - [ ]
DFS traversal order - [ ]

Iterations :-

① Stack : [a]
Visited : {a}
DFS : [a]
a neighbours : b, c, d, e.
Alphabetical order
Push into stak : e, d, c, b.

② Stack : [e, d, c, b]
Pop : b
Visited : {a, b}
DFS : [a, b] Tree edge = (a, b).
b neighbours : a, d, f
Push into stak : f, d

③ stack [e, d, c, f, d].
Pop : d
visited : {a, b, d}
DFS : [a, b, d] Tree edge = (b, d)
d neighbours : a, b, f.
Push : f

④ stack : [e, d, c, f, f]
Pop : f
Visited : {a, b, d, f} Tree edge : (d, f)
DFS : [a, b, d, f]
f neighbours : b, d → All visited
Push : -

BACK TRACK to d. → no unvisited neighbours
BACK TRACK to b → All neighbours visited
BACK TRACK to a
From a, visit c.

⑤ stack : [e, d, c, f]
Pop : f (Already visited ⇒ Ignore).

⑥ stack : [e, d, c]
Pop : c → visited
Visited : {a, b, c, d, f, g}.
DFS : [a, b, d, f, c]
Tree edge : (a, c)
c Neighbours : a, g ⇒ g → visited.
Push : g

⑦ stack : [e, d, f, g]
Pop : g (visited)
visited : {a, b, c, d, f, g}.
DFS : [a, b, d, f, c, g].
Tree edge = (c, g)
g : neighbours = c, e → e visited.
Push : e.

⑧ stock : [e, d, e]
Pop : e
visited : {a, b, c, d, e, f, g}
DFS : [a, b, d, f, c, g, e]
Tree edge : (g, e) e Neighbours : c e
Push :

Finally, DFS spanning tree edges.
(a, b) (b, d) (d, f) (a, c)
(c, g) (g, e).

Code trace (Push, pop order)

Push order:- Wrapper
a → b → c → d → e → f → d → f → g → e

Pop order:-
a → b → d → f → f(skip) → c → g → e
→ e(skip) → d(skip).

Back edge:- Edge from node to ancestor in DFS tree, forms a cycle. Connect to already visited ancestors.
Immediate parent of node should not be considered.

Identified back edges:-
From traversal,
*(d, a) → Back edge (a is already visited, a is not d's parent.)
*(b, f) → Back edge (f is visited via d)
*(e, a) → Back edge.

~~Adj list~~ TC both same: O(V+E)

Topological sorting:- DFS
Linear ordering of vertices s.t. for every directed edge u, u appears before v.

① Directed DAG. graph represented by adjacency list.
② Starting node.
③ Visited set to track visited nodes.
④ Stack

O/P:- sorted.

steps:-
① Mark current node as visited. Add node to visited set.
② Visit all unvisited neighbours of current node. If neighbour not visited, in graph node, if neighbour is visited, repeat process

③ Previous Ques

Start a
· N = [b, c, d, e] → B

From b
N = [a, d, f] → D.

From d
N = [a, b, f] → F

From f
N = [b, d] → BACKTRACK.

BACKTRACK to d } All neighbours visited.
BACKTRACK to b.
BACKTRACK to a.

From a
N = [c] → C

From C
N = [a, g] → G

From g
N = [c, e] → E

From e
N = [a, g]
→ BACKTRACK TO G, C, A.

DFS order:

a → b → d → f → c → g → e

DFS spanning tree: BFS

a→b b→d a→e
a→c b→f
a→d c→g
e→g

For DFS spanning tree, include only edges used to reach for 1st time.

Ignore edge used during backtracking / cycles

start at a

Visit b → edge : a→b

From b
visit d → edge : b→d.

From d
visit f → edge: d→f

Backtrack to a
visit c → edge: a→c

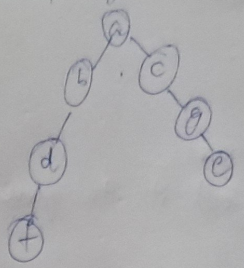
From c
visit g → edge: c→g.

From g
visit e → edge: g→e.

DFS spanning tree edges:-

a→b
a→c c→g
b→d g→e
d→f

DFS spanning tree



Stack tracing:- ✓

Push order: visited node order

a → b → d → f → c → g → e

Pop order:

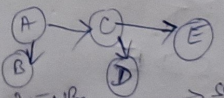
f → d → b → e → g → c → a

How to find back edges:-

* Do DFS
* keep track of DFS tree
* ∀ edge in digraph, check if ① it's not a tree edge ② it connect current node to ancestor

Topological sorting:- Directed Acyclic

=>



USING → SOURCE REMOVAL

= Kahn's algorithm:- BFS + In degree counting

* Nodes

A
B
C
D
E

In degree

	From
0	
1	(A)
1	(A)
1	(C)
1	(C)

* Add all nodes with In degree = 0 to queue.

Queue = [A]

* while queue is not empty.
  a) Remove node from queue.
  b) Add it to result
  c) Reduce indegree of Neighbours
  d) if Nbr indegree become 0, add it to queue

Pop A → result : [A]
B → In degree = 0 ⇒ Add to queue
C → In degree = 0 ⇒ Add to queue
∴ Queue = [B, C].

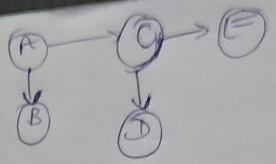
Pop B ⇒ result = A, B.

No outgoing edges from B.
Queue : [C]

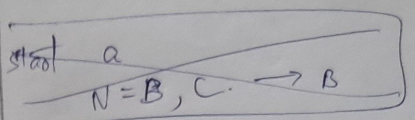
Pop C ⇒ result = A, B, C
D → indegree = 0 ⇒ Add to queue
E → indegree = 0 ⇒ Add to queue
Queue : [D E]
Pop D ⇒ result = A B C D
Pop E ⇒ result = A B C D E

Topological sorting Using DFS

Step 1: Perform DFS from each unvisited node.
Step 2: After visiting all descendants of a node, add node to a stack.
Step 3: At end, reverse the stack to get topological order.



Edges :-
A → B
A → C
C → D
C → E



* Visit A :
  Mark A as visited
  A's N = B, C.

Visit B : From A
  Mark B as visited, B has no neighbours, Add to stack B = [B]

Visit C : From A
  Explore C's neighbours - D, E.

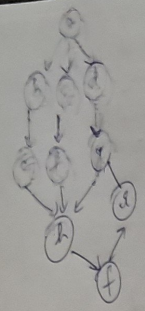
Visit D : From C.
Mark D as visited
D has no neighbors.
Add D to stack
stack = [B, D].

Visit E : From C.
  Mark E as visited.
  E has no neighbours.
  Add E to stack → stack = [B, D, E]

Back to C :
  All neighbours visited → Add C to stack.
  stack = [B, D, E, C].

Back to A :
  All N visited
  stack [B D E C A].

∴ TO = [A C E D B.]



Unit-3

Binary tree :- ~~Data structure~~ o/2 children

Heap DS has a shape such that each node has upto 2 children

Each node in tree store a key.

A key is a value to determine order of heap.

Each node in tree has exactly 1 value i.e key.

Types :-

~~Max~~ Heap :- key at each node is >= key of its children

min key :- " " <= " "

min Ex :- 

max Ex :- 

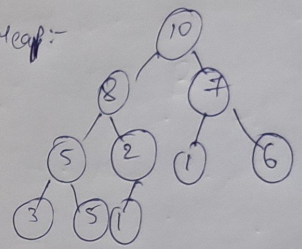
Properties :-

1. Shape :- All level are full except last level.
2. Parental dominance :- Parent > child
   key in each node is >= keys of children

Array :-

Index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1
						Parent		Leaves.			

Heap :-



~~Insertion~~ :- $O(\log n)$ ∴ Max 2 directions.