In DFS, back edge is an edge that goes from a node to the ancestor in DFS tree form a cycle in the graph
(write whatever wasn't in the DFS spanning tree)

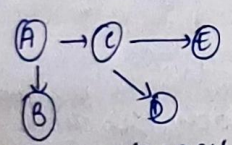
a→d, a→e, b→f, e→g

To check if these are back edge
→Do DFS
→Keep track of the DFS tree
→for every edge in the ~~DFS~~ original graph i) check if it is not a tree edge ii) it connects the current node to an ancestor already visited & still active

→ should be a DAG

Topological sorting

↳ Two methods
i) Kahn's alg. (BFS + in-degree counting) → no. of incoming edge for a given node
ii) DFS ↳(Source removal alg.)



Step 1) Find in-degrees *{write step by step → in deg. conne- → graphs. }

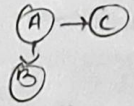
Nodes	In-degree
A	0
B	1 (from A)
C	1 (from A)
D	1 (from C)
E	1 (from C)

Step 2)
Add all the nodes with in-degree value 0 to a queue
Queue = [A]

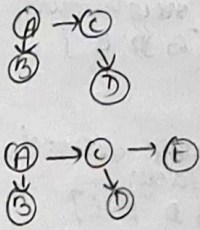
Step 3)
while queue is not empty
i) remove the node from queue
ii) add it to the result
iii) reduce the in-degree of the neighbours
iv) if in-degree of neighbours is 0, add it to queue

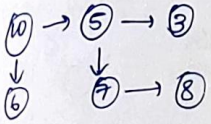
pop A → in degree becomes 0 → add it to queue (A)
C → in degree = 0 → add to queue
Q = [B, C]
pop B → result : [A, B]
no out going edge from B
Q = [C]

pop C → result [A, B, C]
D → in degree = 0 Add to Q
E → =0 Add to Q
Q = [D, E]



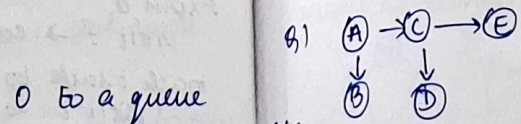
pop D → result [A, B, C]
no out going edge from D
pop E → result [A, B, C, D]
no out-going edge from E
result [A, B, C, D, E]





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

Q1)



BDECA

Soln
Edges: A→B, A→C, C→D, C→E
→ visit A
mark A as visited
explore A's neighbours → B, C
→ visit B (from A)
mark B as visited
~~explore~~ B's has no neighbours
stack: [B]

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

Stack trace (push order & pop order)
push order: a → b → c → d → e → f → d → f → g → e
(visited node list)
pop order: a → b → d → f → f (skip) → c → g → e → e (skip)
order of traversal:- a → b → d → f → c → g → e

Back edges connect to already visited ancestors but immediate parents of the node shouldn't be considered

Identified back edges
From the 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

O(V+E) → time complexity
not possible to tell which is faster (same time complexity)

Directed acyclic graph (DAG) → Topological sorting
A linear ordering of vertices such that for every directed edge (u → v), u appears before v

Algorithm
→ Inputs → directed graph represented by an adjacency list
→ a starting node
→ a visited set to keep track of the visited node
→ Stack (to store the sorted nodes)

Steps
1) mark the current node as visited & add the node to the visited stop
2) visit all the unvisited neighbours of the current node, ~~for each neighbour in the graph~~ node, if the neighbour is not visited, do the topological sorting
3) Push After visiting all the unvisited node, append the node to the stack

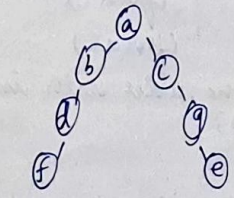
1) neigh = [b, c, d, e] → pick b
2) from b
N = [a, d, f] → pick d
3) from d
N = [a, b, f] → pick f
4) From f
N = [b, d] → backtrack
5) Backtrack to d → no more (unvisited neighbours)
6) Backtrack to b → next neighbour is f → backtrack to a
7) from a,
N = [c]
N = [a, g] → pick g
8) From g
N = [c, e] → pick e
9) From e
N = [a, g] → Backtrack g, c, then a

Spanning tree	
a → b	b → d
a → c	b → f
a → d	c → g
a → e	e → g

For DFS spanning tree construction
- include only the edges used to reach the node the 1st time
- ignore the edges 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

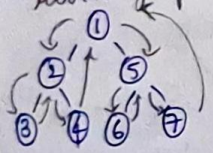
∴ DFS spanning tree edges
a → b d → f
a → c c → g
b → d g → e



push order (visit) → visited node list
a → b → d → f → c → g → e
pop order
f → d → b → e → g → c → a

DFS is a graph traversal
→ go deep into the path until you can't go any further (then go for next path)
→ to check if graph has connectivity through the path
Connected graph: → graph is said so if there is a path btw any two vertices
Disconnected graph: Atleast one pair of vertex has no path btw them
Strongly connected graph: A directed graph is strongly connected if there is a path from A to B but not vice-versa

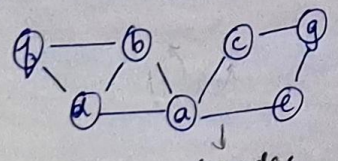
DFS → preorder like traversal (Root Left Right)
rooted graph



→ source node → unvisited
→ mark as visited
→ go deep (until you reach leaf node)
→ then backtrack & go to adjacent subtree

Stack → LIFO

Q) consider the graph given in the figure
i) Starting at vertex A, resolve the ties by vertex (alphabetical) order, traverse the graph by DFS.
→ construct a DFS spanning tree
→ show the traces of traversal stack for DFS & specify the order in which the vertices were pushed onto & popped off the stack → find the tree edges
→ identify back edges if any
→ "All DFS forest will have same no. of tree edges & back edges" → verify if true | false, justify your answer



Adjacency list
a: b, c, d, e
b: a, d, f
c: a, g
d: a, b, f
e: a, g
f: b, d
g: c, e

start from vertex a
visited: [ ]
Stack: [ ]
DFS tree edges: [ ]
DFS order of traversal: [ ]

Iteration 1:
→ Stack [a]
→ V {a}
→ DFS order: [a]
→ neighbours of a: b, c, d, e (in alphabetical order)
→ push to stack: e, d, c, b (reverse order (LIFO))

Iteration 2:
→ stack [e, d, c, b] pop last elem
→ pop: b [visited]
→ visited: {a, b}
→ DFS order: [a, b]
→ tree edge [a b]
→ neighbours of b: a, d, f → visited already
→ push to stack: f, d

Iteration 3: b already popped
→ Stack [e, d, c, f, d]
→ pop: d [visit]
→ visited: {a, b, d}
→ DFS order: [a, b, d]
→ tree edge [b, d]
→ neighbours of d → a, b, f → visited
→ push to stack: f

Iteration 4:
→ Stack: [e, d, c, f, f]
→ pop: f [visit]
→ visited: {a, b, d, f}
→ DFS order: [a, b, d, f]
→ tree edge [d, f]
→ Iteration 5 → all visited
→ Stack: [e, d, c, f] Backtracking
→ pop f: already visited, ignore

Iteration 6
→ Stack: [e, d, c]
→ pop c [visit]
→ visited: {a, b, d, f, c}
→ DFS order: [a, b, d, f, c]
→ tree edge: [a, c]
→ neighbours of c: a, g → visited
→ push to stack: g

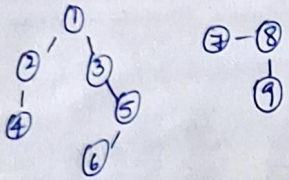
Iteration 7
→ Stack: [e, d, g]
→ pop g (visit)
→ visited: {a, b, d, f, c, g}
→ DFS order: [a, b, d, f, c, g]
→ tree edge: [c, g]
→ neighbours: c, e
→ push to stack: e

Iteration 8
→ Stack [e, d, e]
→ pop e (visit)
→ visited: {a, b, d, f, c, g, e}
→ DFS order: [a, b, d, f, c, g, e]
→ tree edge: [g, e]
→ neighbours: a, g → both visited

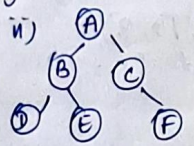
Iteration 9
→ Stack [e, d,] → all visited so ignore

Implement a BFS traversal of graph

i)



ii)



iii) start → A

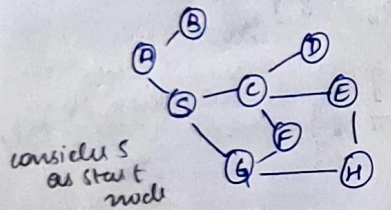
A	Q [A]	V [A]
B, C	Q [B, C]	V [A, B, C]
B	Q [C D E]	V [A B C D E]
A, D, E		
C	Q [D E]	V [A B C D E F]
A, F		
D	Q [E]	V [A B C D E F]
B		
E	Q [F]	V [A B C D E F]
B		
F	Q [ ]	V [A B C D E F]
C		

order: A B C D E F

Implement

i) print the BFS traversal order
ii) construct the BFS spanning tree (constructed from traversal order)

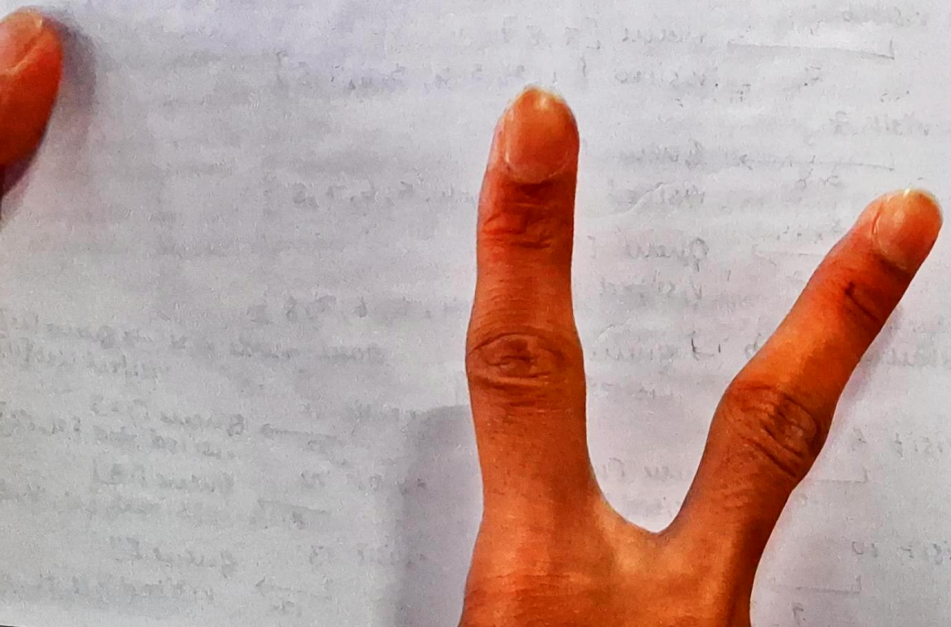
a start node,

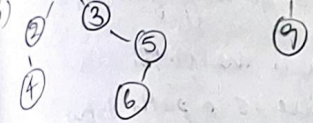


considers S as start node

S G C F H E D A B ?

S A G L B F H E ?





start node → 1 Q [1] V [1]

1	Q [2 3]	V [1 2 3]
2, 3		
2	Q [3 4]	V [1 2 3 4]
1, 4		
3	Q [4 5]	V [1 2 3 4 5]
1, 5		
1	Q [5]	V [1 2 3 4 5]
2		
5	Q [6]	V [1 2 3 4 5 6]
3, 6		
6	Q [ ]	V [1 2 3 4 5 6]
5		

start → 7 Q [7] V [7]

7	Q [8]	V [7, 8]
8		
8	Q [9]	V [7, 8, 9]
7, 9		
9	Q [ ]	V [7 8 9]
8		

the two vertices a & b & there exists an edge btw them,
mmetric → a is connected to b & vice versa
two way connection

mmetric relations: One-way relationship & is
presented by a directed graph & that consists & a set
nodes V & set & directed edges E

u e' ∈ E' is an ordered pair (u, v) where
positions & u & v are not interchangeable
u → tail & edge, v → head & edge
edge e' leaves node u & enters node v

- transportation, communication, information, social & dependency network

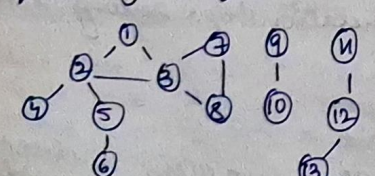
peration: traversing a sequence & nodes connected by edges.

Path: in an undirected graph G=(V, E) is a sequence &
nodes $v_1, v_2, \ldots, v_k$

Properties:
→ each consecutive node is joined by an edge
→ path is simple if all vertices are distinct from one another

cycle: sequence & nodes cycle back to where it began
↳ A cycle in a graph is defined as a path v_1 to v_k in which the sequence & nodes cycles back to where it began

A path in an undirected graph G(V, E) is a sequence & nodes defined by the symbol ?



multiple paths, cycles, disconnected components → graph (complex & undirected)

- Suppose start node = 1 → Queue [1]
  visited list [1]

~~• visit 1~~

• visit 1 neighbours 2, 3 → Queue [2, 3]
visited {1, 2, 3}

take traversal order from queue
1 2 3 4 5 7 8 6 + 9 10 + 11 12 13

• visit 2 neighbour 1, 3, 4, 5 → Queue [3, 4, 5]
visited {1, 2, 3, 4, 5}
↑ visit only neighbours & do not visit the node you're visiting

• visit 3 1, 2, 7, 8 → Queue ~~[7, 8]~~ [4, 5, 7, 8]
visited {1, 2, 3, 4, 5, 7, 8}

• visit 4 2 → Queue ~~[7, 8]~~ [5, 7, 8]
visited {1, 2, 3, 4, 5, 7, 8}
complete 2's children only then go to 3's children

• visit 5 2, 6 → Queue [6, 7, 8]
visited {1, 2, 3, 4, 5, 6, 7, 8}

• visit 6 5 → Queue [7, 8]
visited {1, 2, 3, 4, 5, 6, 7, 8}

• visit 7 3, 8 → Queue [8]
visited {1, 2, 3, 4, 5, 6, 7, 8}

• visit 8 3, 7 → Queue [ ]
visited {1, 2, 3, 4, 5, 6, 7, 8}

start node = 9 → Queue [9]
visited list [9]

• visit 9 10 → Queue [10]
visited {9, 10}

• visit 10 9 → Queue [ ]
visited {9, 10}

start node = 11 → Queue [11]
visited list [11]

• visit 11 12 → Queue [12]
visited ~~list~~ {11, 12}

• visit 12 11, 13 → Queue [13]
visited {11, 12, 13}

• visit 13 12 → Queue [ ]
visited {11, 12, 13}

Case 1: Divide dominates

If d is greater than $\log_b a$ ($d > \log_b a$) then we can conclude time complexity $T(n)$ will be of the order of $\underbrace{O(n^d)}_{\text{cost}}$ (method)

$T(n) \rightarrow$ some where = cost of dive & me

Case 2: Equal growth

If $d = \log_b a$ then $T(n) = O(n^d \log n)$

Case 3: Conquer dominates

If $d < \log_b a$ then $T(n) = O(n \log_b a)$

(8) Solve the following recurrence relations using masters theorem

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

ii) $T(n) = 2T\left(\frac{n}{3}\right) + n_2$

Q) Apply merge sort alg. to the following elem.

→ {5, 2, 4, 6, 1, 3, 2, 6}

→ {20, 10, 5, 15, 25, 30, 50, 35}

Analyzing the merge sort Alg. using recurrence

$T(n) <= 2T(n/2) + cn$ where $n > 2$ & $T(2) <= c$

$T(n) \rightarrow$ worst case RT $n =$ I/P size

I/P is reduced to size 2, recursion stops. [O(log n)]

sort the elems by comparing them to each other [O(n)]

Worst case analysis → [O(n log n)]

Best case: Array is sorted (MS divides the array recursively into halves & merges them back, even if merging is trivial it still requires O(n))

Worst case: Array is completely reverse order

Average case: Considers all possible ways the I/P elems can be arranged

Base case: limiting cond. where soln. occurs recursion

[O(log n) → the problem gets divided into 2, at logarithmic rate

O(log n) → comparisions

i) pivot: pivot is the elem around which the array is divided

ii) index pointer: the pointer keeps track of the boundary where elements smaller than the pivot should be placed.

iii) walker pointer: pointer scans the array to identify elems smaller than or equal to the pivot, which are then swapped with elems at the index pointer.

When the i/p reaches pivot, array divided at that pt. & process repeats

Alg.

→ choose a pivot

→ rearrange the array so that the elems smaller than the pivot are on the left side & elem > pivot on right side

→ repeat until they contain only 1 elem in each & left & right sub-array.

Adv. of Quick Sort over merge Sort-

→ uses the concept of in place sorting. QS sorts the array without creating any addition Data struct. Unlike MS which creates temp sub arrays for merging. QS rearranges elems within the same array.

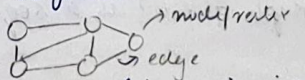
MS → each subproblem → creation of sub-array → space complexity inc.

QS → less space complexity

Substitution method

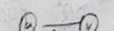
→ guessing a soln. & then

→ using MI to prove

Graph → 

→ It is a way of encoding pairwise relationships among a set of objects

→ consists of vertices & edges

→ collection of nodes joined using edges E. 

→ edge of a graph ($e \in E$) represented as a two elem. subset of V of $e = \{u, v\}$ for some $u, v \in V$, where u & v are the ends i.e. of e

→ Symmetric relationships in graphs: indicated by an undirected graph where the edges in the graph indicates a symmetric relationship btw their ends

Step 0: At the beginning, all [illegible] are free
meaning free

Step 1: Proposal: Select any free man m. m proposes to based on his preference list

Response: a) if woman w is free, she accepts the proposal & get engaged

b) if woman w is not free, ~~she~~
↳ she compares m with her current mate
if she prefers m over her current mate, accepts m & rejects her current mate
↳ she rejects m

Step 2: Continues until all are paired

Analysis: terminates after n^2 iterations

Time complexity & Space com Q's
Design & Analyze algorithm

1b)
↳ Asymptotic notation
what is AN, mention the notations
↳ graph + Eq.
↳ one line about it

Properties
↳ name them, eg.
↳ meaning of Eq (1 line)

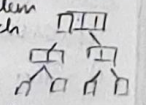
1c) a → d → e → f → g
↳ spanning tree using this

Divide and conquer
→ divide into multiple sub-problem
→ solve each sub problem seperately
→ merge & get the required array

Why is it applied
brute force → polynomial running time
D & C → reducing this running time to a lower polynomial

Merge sort A[1....n]
1. if n=1, done (only 1 elem) ← Base case O(1)
2. Recursively sort A[1....⌈n/2⌉] and A[⌈n/2⌉+1....n] ← Sort both halves
3. "Merge" the 2 sorted lists ↳ divide ends when last level has 1 elem each

Key subroutine:



Recurrence relation
→ Analyzing the running time of D & C algorithm involves solving a recurrence relation
→ limits the run time of alg. recursively in terms of RT on smaller instances
→ expresses RT of an alg. as a fn. of its RT on smaller subproblems
⇒ [• recursion tree • substitution method • masters theorem
↳ to find UB for the alg.

Masters theorem ~~to solve recurr~~
→ provides a way to solve recurrence relations that arise in D & C ~~algo~~ alg.

$$T(n) = aT(n/b) + O(n^d)$$

T(n) → time complexity
a = no. of sub problems into which the original problem is divided
b = reduction factor (factor by which the problem size is reduced in each recursive call)
$O(n^d)$ → cost of merging & dividing at each level

MS → a = b = 2 (dividing into two)

Logarithms

Growth rate: Even the smallest polynomial fn. grows faster than a logarithm fn. as 'n' becomes fn.

$\log_b n$, is the no. x such that $b^x = n$

Eg. $\log_2 8 = 3$ because $2^3 = 8$

1) $\log_2 1000 = 9.9657$ | $2^{1000} =$

$\log_2 0.01 = -6.6438$ | $2^{0.01} = 1.00695$

St: For every base b, the fn. $\log_b n$ is asymptotically UB by every fn. of the form n^ε, $\varepsilon > 0$

8) Consider a logarithmic fn. $\log_2 n$ and $n^{0.01}$. Compute & compare these 2 fn's for some values of n. List your observation & prove that log is having a slower rate of growth

Soln.

	$\log_2 n$	$n^{0.01}$
n = 1000	9.96578	1.0715
n = 5000	12.2877	1.0889

Exponential fn.

Every exponential fn. outgrows the polynomial fn. no matter how large the polynomial exponent is for larger values of n

Exponential GF > polynomial GF > logarithmic GF

(growth fn.)

— x — running time

Common running time

O(1) → constant → accessing an array elem.

O(log n) → logarithmic → binary search

O(n) → linear → finding max in an array

O(n log n) → linear → merge & quick sort

$O(n^2)$ → quadratic → bubble sort & Floyd warshall

$O(n^3)$ → cubic → matrix multiplication

$O(2^n)$ → exponential → recursive fibonacci

O(n!) → factorial → traveling salesman problem

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 women similarly each woman has a ranking list of men

A set of n pairs (m_i, w_i) where each pair is paired with exactly one woman. Our goal is that there should not be any ties

Assumption:

$M = \{(m_1, w_2), (m_2, w_3), (m_3, w_1)\}$ matching pairs

Blocking pair: A pair (m, w) is a blocking pair if,

→ They're not matched in the current pairing

meaning if a man m & woman w have no match yet

→ man m prefers woman more than the current partner

Stable match: A matching is stable if there is no blocking pairs

An instance (Eg) of the problem is represented in 2 ways

→ Two sets of preferences are given or a ranking matrix where each man has ranked on the order of preference & vice versa

The row will represent men, column → woman, both will be given names.

Men's preference list

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

Men's preference 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)

Bob's 2nd pref is Ann 2 (2,3)

Ann's 3rd pref is Bob 3

Soln. Step 1: check if $f(n) \le c \cdot g(n)$ meaning we need to find a value of for c & n_0, such that the condition satisfies

$3n^2 \le 5n^3$ → put $n=1, 2, \ldots$

↳ satisfies for $n \ge 1$

↳ min $n=1$ (n value)

$5n^3 \le 10n^4$ → order of fn.

↳ put $n=1, 2, \ldots$

↳ satisfies $\forall\, n \ge 1$

↳ min $n=1$ (n value)

$3n^2$ has an order of growth of ϕ

$3n^2 \le O(10n^4)$

Sum of fn.'s

If two fn.'s $f(n)$ & $g(n)$ are both bounded above by another fn. $h(n)$, then their sum is also bounded by $h(n)$

↳ 'growing at same rate' → UB

↳ $(f(n)+g(n))$

If $f(n) = O(h(n))$ → (a)

If $g(n) = O(h(n))$ → (b)

then $f(n)+g(n) = O(h(n))$

order of 3rd fn. is same as order of other two fn

Proof: Consider Eq (a) $f(n) = O(h(n))$ meaning $\exists$ constants c_1 and n_0 such that $\forall\, n \ge n_0$

$f(n) \le c_1(h(n)) \rightarrow (1)$

similarly,

$g(n) \le c_2(h(n)) \rightarrow (2)$

$\therefore$ (1) + (2)

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

$f(n) + g(n) \le (c_1 + c_2)\, h(n)$

$f(n) + g(n) \le$ [illegible] $(h(n))$

8) Consider [illegible] $f(n) = 3n^2$ and $g(n) = 5n^2$

Prove that

8) $f_1(n) = 3n^2$, $f_2(n) = 5n^2$, $f_3(n) = 2n^2$ → all have n^2 so order of growth $O(n^2)$

$f_1(n) + f_2(n) + f_3(n)$

$= 3n^2 + 5n^2 + 2n^2$

$= 10n^2$

order of growth doesn't change hence proof.

sum of many fn's

$g(n) = 2n^2$ $g(n) = 3n^3$ $g(n) = 4n^2$ → polynomial, so for ↳ take highest degree

Property 4: if two fn's $f(n)$ & $g(n)$ are taking non-ve values such that $g(n) = O(f(n))$ then $f(n) + g(n) = \Theta(f(n))$

growth rate of $f(n)$ & $g(n)$ → same

If one fn. $g(n)$ grows atmost as fast as another fn. $f(n)$ then sum of $f(n)$ & $g(n)$ will have same growth as fn. In other words, the fastest growing fn. dominates the sum of the fn.'s

Asymptotic bounds for some common fn.

→ some fn.'s come up repeatedly

↳ polynomial, logarithmic, exponential

Polynomials → $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$ $a_j n^d$ $j < d$

d → degree of the polynomial, $d > 0$, a_d → non zero

Their asymptotic order of growth / rate of growth is determined by their "highest order term"

$pn^2 + qn + q \rightarrow d = 2$

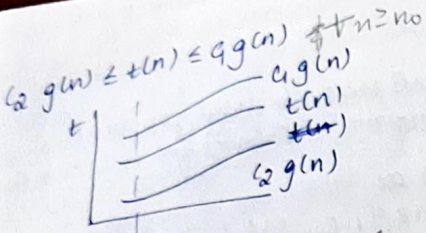
Conclusion:

→ the dominant term dictates the growth rate

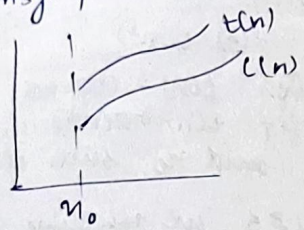
→ lower degree term like n^{d-1}, n^{d-2} etc, grow much slower than n^d & become significant in Big O notation

→ since Big O ignores constant factors, we only consider the highest degree term's growth

→ sum of multiple $O(n^d)$ terms is still $O(n^d)$



Asymptotic lower bound



Θ type bound, $t(n)$ grows btw c_1 & c_2

Proof 1: To obtain an asymptotically tight bound for a limit
St: An asymptotically tight bound can be obtained directly by computing a limit as n goes to ∞. If the ratio of fn.'s $f(n)$ & $g(n)$ converges to a +ve constant as n goes to ∞, then
$f(n) = \Theta g(n)$

Let f & g be two fn.'s such that
$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ ⟷ mathematically written as
exists & is equal to some no. $c>0$, then $f(n)=\Theta g(n)$

If two fn.'s $f(n)$ & $g(n)$ grow at the same rate, as n becomes very large then we say that $f(n)=\Theta g(n)$ meaning $f(n)$ & $g(n)$ differs by a constant c

Proof
There exists a limit & it is +ve,
now $f(n)=O(g(n))$ & $f(n)=\Omega(g(n))$ } limit
↳ upper bound ↳ lower bound

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ (+ve), there is some no beyond which the ratio is always btw $\frac{1}{2}c$ & $2c$ (As per calculus)
Thus, $f(n) \le (2c)\, g(n)$ (+ve) $\forall n > n_0 \Rightarrow f(n) = O(g(n))$
$f(n) \ge \left(\frac{1}{2}\right) c\, g(n)$ $\forall n > n_0 \Rightarrow f(n) = \Omega(g(n))$
($c_1 \to 2c$, $c_2 \to \frac{1}{2}c$)

By the defn. of limits in calculus for any small +ve no, $\exists$ some large no such that $\forall n \ge n_0$ the ratio of the fn. stays within a small range of c

Transitivity
St: If we have three fn.'s $f(n)$, $g(n)$ and $h(n)$, we say that the fn. $f(n)$ is asymptotically upper bounded by a fn. $g(n)$ if $g(n)$ in turn is asymptotically upper bounded by a fn. $h(n)$ then $f(n)$ is asymptotically upper bounded by $h(n)$.
a) if $f(n) = O(g(n))$
b) if $g(n) = O(h(n))$
then we conclude $f(n) = O(h(n))$

Proof: $f(n) \le c\, g(n) \to (1)$
$g(n) \le c'\, h(n) \to (2)$

Big O → $f(n) \le \Theta(g(n))$

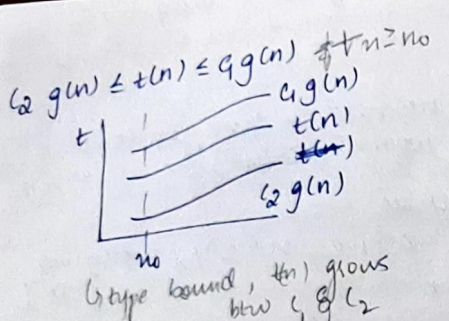
There exists constants c and n_0 such that $\forall n \ge n_0$, Eq (1) satisfies
There exists constants c' and n_0' such that $\forall n \ge n_0'$, Eq (2) satisfies
To prove the transitivity property we have to find value for n such that
$n >$ ~~max~~ both n_0 & n_0' ($n_0 = \max(n_0, n_0')$)

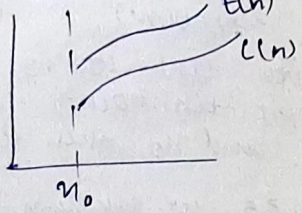
min value for Eq (1)

Substitute (2) in (1)
$f(n) \le (\, g(n)$
$\le c(c'\, h(n))$
$f(n) \le (cc')(h(n))$
$f(n) \le c\, h(n)$ → constants
$f(n) = O\, h(n)$ hence proved

8) Consider 3 fn.'s ; $f(n) = 3n^2$; $g(n) = 5n^3$, $h(n) = 10n^4$
Prove transitivity across the three fn.'s
$3n^2 \le c\, 5n^3$
$5n^3 \le c'\, 10n^4$ → $3n^2 \le cc'\, 10n^4$

- Asymptotic lower bound





c_2 type bound, $t(n)$ grows btw c_1 & c_2

Proof 1: To obtain an asymptotically tight bound for a limit
St: An asymptotically tight bound can be obtained directly by computing a limit as n goes to ∞. If the ratio of fn's $f(n)$ & $g(n)$ converges to a +ve constant as n goes to ∞, then,

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

Let f & g be two fn's such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ (↔ mathematically written as) exists & is equal to some no. $c>0$, then $f(n)=\Theta(g(n))$

If two fn.'s $f(n)$ & $g(n)$ grow at the same rate, as n becomes very large then we say that $f(n)=\Theta(g(n))$ meaning $f(n)$ & $g(n)$ differs by a constant c

Proof
There exists a limit & it is +ve,
now $f(n)=O(g(n))$ & $f(n)=\Omega(g(n))$
(upper bound, lower bound) } limit

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$ (+ve), there is some no beyond which the ratio is always btw $\frac{1}{2}c$ & $2c$ (As per calculus)

Thus, $f(n) \le 2c\, g(n)$ ($2, c, g(n)$ → +ve) $\forall n > n_0 \Rightarrow f(n) = O(g(n))$

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

$\frac{1}{2}, c \to \Omega$

By the defn. of limits in calculus for any small +ve no, ∃ some large no such that $\forall n \ge n_0$ the ratio of the fn. stays within a small range of c

Transitivity
If we have three fn's $f(n)$, $g(n)$ and $h(n)$, we say that the fn. $f(n)$ is asymptotically upper bounded by a fn. $g(n)$ & if $g(n)$ in turn is asymptotically upper bounded by a fn $h(n)$ then $f(n)$ is asymptotically upper bounded by $h(n)$.

eg if $f(n) = O(g(n))$
& if $g(n) = O(h(n))$
then we conclude $f(n) = O(h(n))$

Big O → $f(n) \le c\, g(n)$

Proof: $f(n) \le c\, g(n) \to (1)$
$g(n) \le c'\, h(n) \to (2)$

There exists constants c and n_0 such that $\forall n \ge n_0$ (min value for n_0), Eq (1) satisfies

There exists constants c' and n_0' such that $\forall n \ge n_0'$, Eq (2) satisfies

To prove the transitivity property we have to find value for n such that

$n > \max(n_0, n_0')$ (both n_0 & n_0')

substitute (2) in (1)

$f(n) \le c\, g(n)$
$\le c(c'\, h(n))$
$f(n) \le (cc')(h(n))$
$f(n) \le c\, h(n)$ → constants

$f(n) = O(h(n))$ hence proved

8) consider 3 fn.'s ; $f(n) = 3n^2$; $g(n) = 5n^3$, $h(n) = 10n^4$
Prove transitivity across the three fn.'s

$3n^2 \le c\, 5n^3$
$5n^3 \le c'\, 10n^4$ → $3n^2 \le cc'\, 10n^4$

These are absolute constants $c>0$, $d>0$ so ... for every instance of size N, the running time is bounded by cN^d primitive computational steps (almost proportional) { N gets bigger, RT grows at the rate of N^d but not faster

Running time $\propto N^d$

RT of an algorithm is controlled by a mathematical rule that limits how many steps the algorithm can take to accomplish a task.

$c \rightarrow$ scaling factor / fixed no. } do not depend on N

$d \rightarrow$ (growth rate / exponent } size of i/p

↳ adjusts (no. of steps

↳ determines how algorithm's ~~run~~ RT increases as the i/p N grows

cN^d = max no. of steps for i/p of n wrt time t

Asymptotic analysis / growth / notations

↳ mathematical way to find time complexity of an algorithm

$N \rightarrow$ i/p size

$f(n) \rightarrow$ growth size

$T(n) \rightarrow$ time complexity [Any algorithm's worst case running time]

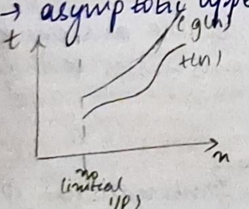
$C(n) \rightarrow$ Basic operation count

$g(n) \rightarrow$ it is a simple fn. to compare the count or order of magnitude

- $O \rightarrow$ big oh
- $\Omega \rightarrow$ big omega
- $\theta \rightarrow$ big theta
- $o \rightarrow$ little oh
- $\omega \rightarrow$ little omega

Big O $\rightarrow$ asymptotic upper bound

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



$n_0 \neq 0$

$n_0 \geq 1$ (Assume)

$f(n) \leq c\, g(n)$

We say that a fn. $g(n)$ is an upper bound for another fn. $t(n)$ if beyond some (n_0) point, the fn. $g(n)$ dominates $t(n)$. Here we told $g(n)$ is a fn. which describes the order of magnitude. There is a fixed constant c & beyond some time t

Q) Prove $t(n) = 100n+5$ is in the order of $O(n^2)$ using big-O notation

$t(n) = O(n^2)$

Soln: given: $t(n) = 100n+5$

TPT: $t(n) = O(n^2)$ i.e., we need to find constants c and n_0 such that $100n+5 \leq cn^2$ ~~$O(n^2)$~~ for all $n \geq n_0$

for $n \geq 5$, we modify $t(n)$ as

$100n + 5 \leq cn^2$ [∵ $n \geq 5$ i.e., $n \geq 5 > 1$, meaning $5 \leq n$]

i.e., $100n + 5 \leq 100n + n$

so replacing 5 with n makes $100n + n$ larger

→ $100n + n$ can be simplified as $101n$

Since $n^2 \geq n$ for all $n \geq 1$ we can replace n with n^2

So, $101n \leq 101n^2$

Thus,

$100n + 5 \leq 101n^2$ for $n \geq 5$

Hence to satisfy the Big-O defn. $t(n) \leq c.n^2$ for all $n \geq n_0$

choose $c = 101$

$n_0 = 5$

This confirms $100n + 5 \leq 101n^2$ for all $n \geq 5$

$t(n) = O(n^2)$

Q) consider $t(n) = 100n^2 + 20n + 5$, prove that it is in the order of $O(n^2)$ using big-O notation.

$$\frac{100n^2 + 20n + 5}{n^2}$$

$$100 + \frac{20}{n} + \frac{5}{n^2}$$

$100n^2$

A → n^2 growth rate ↑ so efficiency ↓

B → n log n GR ↓ so efficiency ↑

Unit - I

→ <u>Computational tractability</u>: It is a factor that is used to assess or identify if a specific problem st. at hand can be solved with an algorithm

→ Does by determining the resource requirements (time & space needed for <u>execution</u> of the algorithm)

I/P n ↑ space & time ↑ resources $ ↑

~~resources~~ resource requirement ↑ size of I/P ↑

<u>Efficiency</u>: An algorithm is efficient, if when implemented, it runs quickly on <u>real I/P instances</u>

redefined defn: An algorithm is efficient, it is platform independent, instance independent, and of predictive values wrt increasing I/P sizes

mathematical defn: given a problem st. with I/P size N, an algorithm can be described & then analyzed for performance by computing the running time mathematically as a fn. of the I/P size N. f(N)

Worst case running time

→ worst case: max amt of time/resources an algorithm will take to complete a task

→ Avg case: expected running time of an algorithm, (uses random instances)

→ Best case: swiftest time an algorithm takes...

All depends on I/P's → I/P's in the real world follow specific pattern/struct, not truly random

- sorting: partially sorted not jumbled with no relation
- web search: follow lang rules, pattern, trends, we can't type random letters

Disadv. of avg case

→ performs well for one set of I/P & not for others

→ real time I/P not produced from a random distribution

∴ worst case is efficient

How to ... worst case running time?

Step 1: calc. the upper bound on the running time of the algorithm

↳ largest possible running time an algorithm can have over all I/P's of a given size N is calculated.

↳ Done by identifying case that causes the execution of max. no. of operations / longest time

Step 2: Observe how running time scales with I/P 'N'

Brute force search: all possible soln to a PS from which we find the best one → slow, gives no insight into the struct. of the problem

running time bound: upper (→ O( )), lower (→ Ω(n)) or tightest (→ Θ(θ)) limit on the time complexity of the algorithm as a fn. of the I/P size n (↳ f(n))

↳ how to decide if its impressive or weak?

→ decided based on <u>models of computation</u> ↳ help understand the limitations ↳ meaning what can be solved with algorithms

- ↳ turing machine
- ↳ word RAM
- ↳ Brute force

(should study)

Efficiency of algorithm can be defined wrt → terminology worst case performance parameters

An algorithm is efficient if it achieves <u>qualitatively better</u> (→ significantly better) worst case performance, at an <u>analytical level</u> than brute-force search ↳ analyzing alg. using mathematical tools like Big O notation & not by relying on two I/P's

meaning

→ even in the worst case, if it solves a problem faster than brute force

→ brute force search checks all possible ans one by one

→ a better algorithm - finds the ans faster with smart techniques Eg: permutation, path etc. - problems that involve choosing & arranging items in diff ways

Polynomial time as a defn. of efficiency

PS: no. of possible soln.'s for any combinational (optical) problem grows exponentially in the size N of the I/P (2^n, n!)

i.e., when I/P size inc by 1, (even a small inc. in N causes a huge jump in the no. of possibilities)

(N+1) → no. of possible soln.'s inc multiplicatively

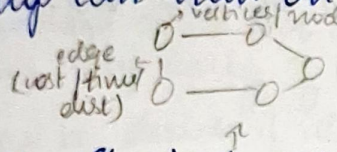
needed soln.: a good algorithm with a better scaling property is required

Travelling salesman problem: visit n cities exactly once & return to no. of possible tours (n-1)! → N=5 → 24 routes, N=6 → 120 routes; if N=N+1 ↳ multi... inc

Design and Analysis of Algorithms

Algorithm → set of steps to complete a task
Provides a step by step procedure that converts an I/P into a desired O/P
→ should have a logical structure,
→ receives few I/P's (I/P data)
→ performs a series of operations (processing)
→ produces a desired result (O/P data)

comp algorithm - a set of steps to accomplish/complete a task that is described precisely enough such that the comp can run on it



↳ solving problem, optimizing soln, automating tasks
↳ reduce time/resources
↳ improve efficiency (for

gps - shortest path algorithm
online shopping - RSA algorithm [cryptography technique]
↳ secure transfer ↳ transfers packets across internet

characteristics of Comp Algorithm
→ definiteness - instructions should be clear & unambiguous
→ finiteness - should terminate after a no. of steps
→ effectiveness - simple inst.
→ efficiency - [time & space] → computational resources

Analysis of algorithms - process of evaluating the efficiency of algorithms, focusing mainly on time & space complexity
↳ it helps in: how the algorithm's running time / space requirements grow in size

processor speed
running time
1) A → n^2 → 10 billion (10^{10}) inst/sec → $2n^2$ inst (N) ← no. of steps
B → n log n → 10 million (10^7) inst/sec → 50 n log n inst (N)
] compute time taken & conclude efficiency

I/P size → 10 million (10^6) no.'s → why? (10^7)

$$\text{time taken } (T(N)) = \frac{\text{no. of steps } (N)}{\text{speed}}$$

why? ← $\log_2$ → take base as 2

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

For B = $\frac{50 n \log_2 n}{10^7} = \frac{50 (10^7) \log(}{10^7}$ = 1162.67

∴ B is more efficient

LPP goal: i) optimize a linear fn. Go find ... soln. for a fn. ii) optimal val can be either the max / min val ...

How to solve LPP?
Step 1: mark the decision var's in the psblm.
Step 2: build the obj fn. of the psblm & check if the fn. needs to be minimized / maxi..
Step 3: write down all the constraints of the linear psblm
Step 4: Ensure non-neg. restrictions of decision var's
Step 5: Now solve LPP

Simplex method
→ represent as a std. form.
  ↳ must be a maxi... psblm
  ↳ all the constraints must be in the form of linear eq. with non-negative
  ↳ all var's should be non-neg.

To consolidate simplex alg.
Step 1: set up the psblm
  2: make the first simplex table
  3: check the bottom row (Z-row)
  4: choose the entering var.
  5: choose the leaving var. (min ratio test)
  6: do the pivot (update the table)
  7: repeat until optimal

Floyd's alg (All pair's shortest path problem)

used to find the shortest path distances btw every pair of vertices in a weighted graph.

Works for both directed & undirected graph.

Alg

I/P: weight matrix W of a graph with no negative-length cycle → W[i,j]
- ↳ is the direct edge cost from vertex i to vertex j
- ↳ ∞ to direct edge

O/P: ~~the~~ dist matrix D which gives shortest dist from i to j

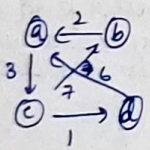
D ← W (weight matrix) // D: dist matrix

for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      D[i,j] ← min [D[i,j] , D[i,k] + D[k,j]]
return D

neg. edge → a → b (edge weight -ve)

neg cycle → a → b → c → a, sum = -2 (sum of edge dist is -ve)

dist matrix is updated by either keep the current known shortest path from i to j or by going through an intermediate vertex k & see if that path is shortest after n such iterations all shortest pairs are found.



digraph

a→a no loop
a→b no direct path

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{bmatrix} a & b & c & d \\ 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}$$

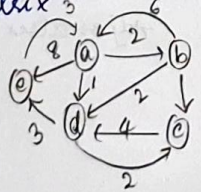
weight matrix

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{bmatrix} a & b & c & d \\ 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

	Path	cost
a → a		0
a → b	a → c → b	10
a → c	a → c	3
a → d	a → c → d	4

Floyd's will not work if there are negative cycles

Q) Solve the all pair shortest path problem for the digraph with the following weight matrix



$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}\begin{bmatrix} a & b & c & d & e \\ 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \quad D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \infty & 4 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix} \quad D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix} \quad D^{(4)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \infty & \infty & 0 & 4 & 7 \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ 10 & 12 & 0 & 4 & 7 \\ 7 & 9 & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$ final answer

Iterative approach

- → start with a soln. that satisfies all the constraints of the problem
- → proceed to improve by repeated applications of some simple step.
- → involves a small, localized change yielding a feasible soln. with an improved val. of the objective fn.

Linear programming: a mathematical concept that is used to find the optimal soln. of the linear fn. It uses all available resources in a manner such that they produce the optimum result.

components of LP
- ↳ decision var. – var's determined to achieve optimal soln.
- ↳ objective fn. – mathematical eq. that represents the goal you want to achieve
- ↳ constraints: limitations that your decision var's follow
- ↳ non-negativity restrictions: decision var's can't be neg.

Special case of LP: max. flow problem – maximize the amount of flow that can be sent through a network with links of limited capacities

Alg. analysis

O(nW), suitable for reasonable sizes of weights wi.

Bellman ford's alg. – min. cost path prblm → use: toll booth in roads (negative charge)

Definition: find shortest path in Directed graph. each edge (i, j) has an associated weight (wij)

find neg. cycle, minimum cost path,

i) ∃ ij ∈ P cij < 0 — check if it has neg cycles

ii) ∃ ij ∈ P cij — if no neg. cycle exist, find path P from origin to dest.

iii)

Hence also called as min-cost path prblm & the shortest path prblm.

Designing & Analyzing an alg

Disadv. of dijkstra's – may not yield the correct shortest path

for an edge u → v with weight w, if the known dist to u plus w is less than the know dist to v, update

dist[v] > dist[u] + weight(u, v)

update: dist[v] = dist[u] + weight(u, v)

no. of times to Relax edges → |V| − 1 updating current dist & with updated val)

eg. dist[1] = 0

Edge 1 → 2, weight = 3

dist[2] = ∞

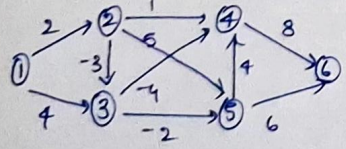
now, dist[2] > dist[1] + 3

∞ > 0 + 3 → relax it

dist[2] = 3

Steps

1) initialize a table to store min. costs for paths
2) fill the table iteratively using previously computed costs
3) return the final cost for the shortest path



Warshall's alg. → alg (finding connectivity) only for directed graph

To calc. transitive closure of a directed graph

~~A digraph with warshall's alg~~ the transitive closure of a directed graph with n vertices can be represented as a n×n boolean matrix T = {tij}, in which the element in the ith row & the jth column is 1 if there exists a non-trivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, tij is 0.

Alg

I/P: adj matrix A of a digraph G with n vertices

O/P: transitive closure of the digraph.

$R^{(0)} \leftarrow A$ (adj matrix initialized as first I/P for R)

i = source vertex
j = dest. vertex
k = intermediate nodes

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R(k)[i,j] ← R(k-1)[i,j] or (R(k-1)[i,k] and R(k-1)[k,j])
return R(n)
```

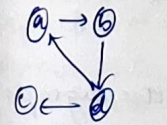
For each intermediate vertex k from 1 to n, ~~outermost loop is iterating from~~
For each source vertex i from 1 to n
For each dest vertex j from 1 to n

If there was already a path from the source vertex i to the dest. vertex j without using the intermediate vertex k or there is a path from the source vertex i to the intermediate vertex k & a path from the intermediate vertex k to the dest. vertex j

After processing all k return $R^{(n)}$

Time complexity O(n3) → 3 loops

Path	
a → a	a → b → d → a
a → b	a → b
a → c	a → b → d → c
a → d	a → b → d



Digraph

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adj matrix

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

transitive closure

this suggests finding the optimal soln. on the intervals (1, 2, 3..., n) involves looking at the optimal soln. of smaller problm. of them (1, 2... j)

Opt(j) → max value for a subset

Best case: opt(0) = 0 (no intervals)

Two choices for j

include j:

gain value v_j + best val of non-overlapping sol-opt

exclude j:

take opt(j-1)

formula opt(j) = max(v_j + opt(p(j)), opt(p(j-1))

inclusion cond:

Alg:

compute opt(j)

if j == 0:

return 0

else:

return max(

v_j + computeopt(p(j)), value by including j

compute opt(j-1) value by excluding j

)

To prove correctness of alg: MI

Best case: opt(0) = 0

inductive step: for any j > 0, we use recurrence

opt(j) = max(v_j + opt(p(j)), opt(j-1))

it checks whether including / excluding j gives a better result

Issue with alg

→ exp. time complexity

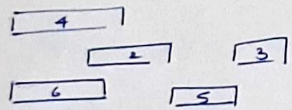
memoization: pruning technique used to speed up recursive alg. by storing the results of expensive fn. calls & reusing them when the same i/p's occur again.

Unit-4

- GA are simple & intuitive but don't always guarantee correct/optimal soln.
- many real world pblm → don't have natural GA

Limitations
- GA fails when local choices don't lead to global optimum
- DC → for complex pablm → dosen't reduce exp time to polynomial time
  unseful only in MS & QS

Dynamic programming → instead of solving pablm by splitting them into independent subproblem
- solves overlapping subproblems & reuses soln's to avoid re-computation



width = time
no on bar = enjoyment score
GA → picking next event with ↑ enjoyment score

Problem
- Event 6 overlaps with others: prevents from selecting multiple smaller overlapping events that tog. have a higher total score
- greedy picks 6 & misses the better overall combination

Dynamic pagming
- considers all combinations of non-overlapping events
→ use previously solved subpblm to determine the optimal subset
→ 4, 2, 5, 3 do not overlap & tog give a score of 10 > 6

why does GA fail? Because it makes locally optimal choices
why does DP succeed? considers global optimality through subproblem reuse.

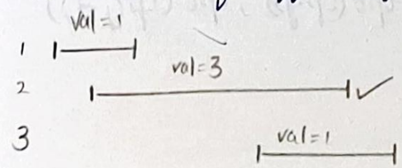
DP
→ dividing the problem into subproblem
→ solving & storing soln.'s to subproblem
→ combining them to build soln.'s to bigger pablm.

Weighted interval scheduling
Stage 1: as a recursive procedure - closely resembles brute force search
Stage 2: converting recursive to iterative alg. building soln. from smaller sub-pablm.

- Earlier: start & end time
- Now: Earlier + weight/value

Goal: select non-overlapping interval with max total value.
Why greedy fails: in weighted case, GA dosen't guarantee optimality
Eg. choosing interval with earliest end time might skip a higher value combination

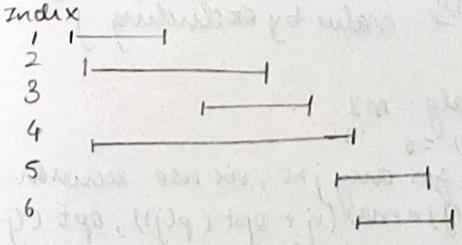
Why use DP
- recursive → explore optimal substructures / subps.
- convert to iterative → for efficiency



weighted int. - weight/val (v_i)

Goal: Design a recursive alg. using DP for weighted int.
Objective: Select non overlapping intervals to maximize total weight/val
each interval has start time s_i, finish time f_i & weight w_i
compatible intervals: are the ones that do not overlap
sorted by finish time: $f_1 \le f_2 \le \dots \le f_n$

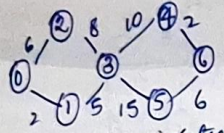
$p(j)$ → no of intervals completed before it



p(1) = 0
p(2) = 0
p(3) = 1
p(4) = 0
p(5) = 3
p(6) = 4

To decide whether to include the last interval in the final ans
→ if n ∈ optimal sel-
  intervals p(n)+1 to n-1 must be excluded
  meaning: you cannot pick any overlapping intervals.
  include optimal soln. for (1, ..., p(n)): pick best possible soln. from earlier, non-overlapping interval, then add val. of n to that.
→ if n ∉ optimal sel-
  optimal soln. lies in (1, .., n-1): exclude n from soln.

8.) REDO



connected, weighted, undirected graph.

initially $w = \infty$

source node → 0

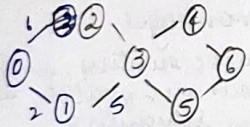
from 0, two adj nodes → go to ① (min weight)

a)

Dist: visited = {0}

0 : 0 ✓
1 : ∞
2 : ∞
3 : ∞
4 : ∞
5 : ∞
6 : ∞

b)

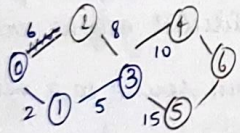


visited = {0, 1}

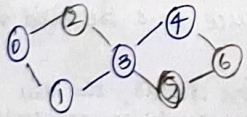
Dist.

0 : 0 ✓
1 : 2 ✓
2 :
3 :
4 :
5 :
6 :

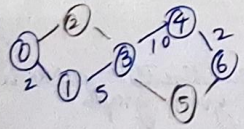
c)



d)



e)



8 + 5 + 10 + 2 = 19

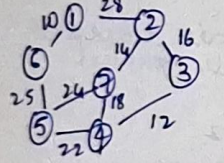
huffman trees and codes

- It is a lossless data compression alg.
- The idea is to assign variable length binary codes to the I/P chars based on the freq's of corresponding chars.
- We can use fixed length encoding that assigns to each symbol a bit string of the same length. This is exactly what the std. ASCII code does.

Kruskal's alg.

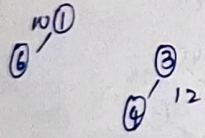
1) Sort the edges by their weight in ascending order
2) Add edges to MST starting with smallest edge & moving to larger edges
3) At each step, check if adding next edge would create a cycle
4) If it creates a cycle, ignore that edge
5) Continue the process until all vertices are traversed

Q)

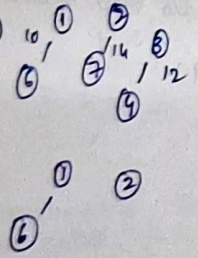


1)

2)

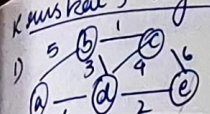


3)



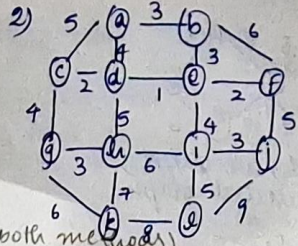
Kruskal (4 steps)

1)



consider 'a' as the source vertex

2)



→ priority queue & adjacency matrix (both methods)

Dijkstra's algorithm (weighted, connected with non-neg. weights)

- Solve the single-source shortest paths problem
- source vertex: in a WG, given vertex is called source
- Aim: To find shortest paths to all the other vertices from the given source vertex
- NOTE: We are NOT finding a SINGLE shortest path.
- Instead finds a FAMILY OF PATHS each leading from source to different vertex
- We catch these, some paths may have common edges
- Application: transportation planning, packet routing, finding shortest path in social network, speech recognition, doc formatting, robotics, compilers, airline crew scheduling

Floyd's alg: used for more general all pairs shortest paths problem

Dijkstra's alg: applicable to directed & undirected graphs with non-negative weights only.

~~Logic~~ Step 1: find the shortest paths from source to a vertex nearest to it (cond min weight)

Step 2: find the shortest path from source to a second nearest vertex.

~~Step 3:~~ in general, before the i^{th} iteration starts, the alg. already identifies the shortest paths to i-1 other vertices nearest to the source

Subtree: of a given tree
the vertices, source & the edge of the shortest paths leading from the source from a subtree T_i of the given graph.

Fringe vertices: as all the ~~vertices~~ weights are non-negative, the next vertex nearest to the source vertex can be found among the vertices adjacent to the vertices of the subtree T_i

candidates from which DA selects the next vertex nearest to the source

Schedule to minimize lateness : an exchange alg.

- Given a set of n jobs each with processing time t_i & due date d_j
- each job scheduled to single machine, st. time = 0
- Objective: schedule all jobs to minimize the maximize lateness (no preemption)

lateness of job i = completion time of i − d_i

lateness = $\max_i (c_i - d_i)$

Sorting jobs based on (by) inc due date & schedule them

Job	processing time	Due date
A	3	4
B	2	2
C	1	6

B → A → C (based on inc. due date 2, 4, 6)

B = 2 − 2 = 0
A = 4 − 3 = 1
C = 6 − 1 = 5

B starts at 0, ends at 2 → lateness = 2 − 2 = 0
A starts at 2, ends at 5 → lateness = 5 − 4 = 1
C starts at 5, ends at 6 ? → lateness = 6 − 6 = 0

max lateness = 1

Proof:
1) Assume S is an optimal schedule not ordered by due date
2) identify the first inversion in S: jobs i & j such that:
   → $d_i > d_j$ but job i comes before job j
3) Swap i & j in the schedule

i A 4 d_i
j B 2 d_j } swap $d_i > d_j$

A minimum spanning tree (MST) is a subset of edges of a connected, weighted, undirected graph that-
i) connects all the vertices i.e., forms a spanning tree
ii) have no cycles i.e., forms a tree
iii) has the min possible edge weight-

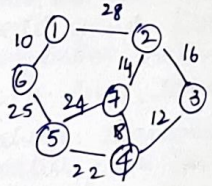
Prims algorithm (connected, weighted)

// prims alg. for constructing a MST
// input: a weighted connected graph $G = \{V, E\}$
// output: E_T, the set of edges composing a MST of G.
$V_T \leftarrow \{v_0\}$ // set of tree vertices, can be initialized with any vertex
$E_T \leftarrow \phi$

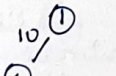
for i ← 1 to |V| − 1 do
  find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$
  $V_T \leftarrow V_T \cup \{u^*\}$
  $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Cond: all nodes traversed, no cycles formed

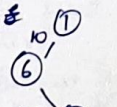
8)



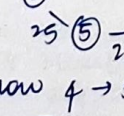
Start node = 1
1 →(28) 2 1 →(10) 6, choose 1 →(10) 6 (min weight)



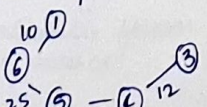
check btw 6 & 1 which has a min connected edge so 6 →(25) 5



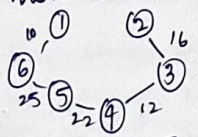
check 1, 6, 5 so 5 →(22) 4



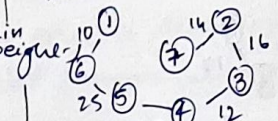
now 4 → 3



now 3 → 2



now 2 → 7



all nodes traversed so stop.

weight = 99

8) Construct a heap for the list 17, 21, 105, 2, 3, 11, 22

transform & conquer
↳ constructed as BT → then transformed to heap → then conquer (sorting)
change of DS

greedy algorithm
- A problem st. → has n no. of feasible soln. but can have only one optimized soln.
- Approach used for PS which needs an optimized soln. (Best soln.)
  ↳ A PS that requires a minimum/max result

Interviews	Start time	End time
A	1	4
B	3	5
C	0	6
D	5	7
E	8	9
F	5	

Goal: attend as many interviews as possible
condition: no overlapping
go to least interview end time

Starting with min start time & (ET, ST) is small
A, D, E no overlapping, each interview starts one after the other
optimal soln. → 3 interviews (Minimum Optimization)

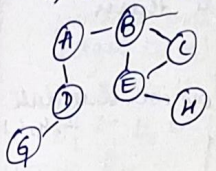
knapsack problem
- knapsack has a capacity (weight) w
- you have several items, each item has weight & benefit (cost)
- fill up the sack
  → knapsack weight capacity is not exceeded
  → total benefit is maximal

(Max optimization)

Identifying the key components for a greedy alg.
- candidate set: set composing of all the possible soln's for a PS
- selection fn. - used to choose / filter out best candidate to be added to the soln.
- feasability fn. - applying a local cond. or pruning cond thereby to determine whether a candidate can be used to contribute to an effective soln.
-

- Objective soln: fn used to assign a value to a soln. or a partial soln. at each step of alg. helps the alg. make local optimal choices with a goal of finding a global optimum
- soln fn - used to indicate whether a complete/best possible soln. has been reached

minimum spanning tree
↳ connected weighted undirected graph



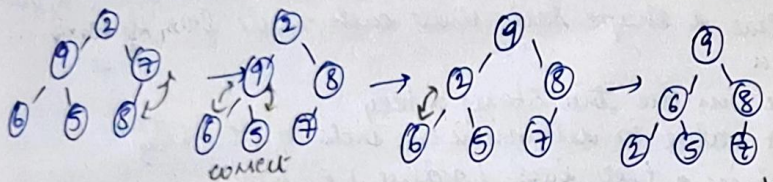
traverse the graph from source to dest. such that the weight is minimum.
→ Drawn from a parent-graph
Spanning tree can be drawn even if the graph is directed & disconnected

Koushikal's algorithm
constructs the spanning tree based on the ~~arranged~~ order of weights (ascending)
↓
minimum ST should be
cond: acyclic

greedy alg (how does it produce an optimal soln?)
→ 1st approach: establishing that the soln "stays ahead"
→ 2nd approach: "Exchange argument"
  ↳ take all possible soln. → use the greedy cond. to get a soln.
    Eg: spanning tree (minimal)
  ↳ step by step process → decisions / I/P's at every step helps get a soln. better than the other. Eg: Interview

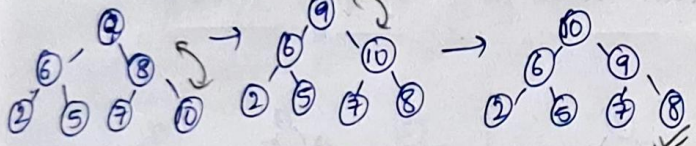
Interview scheduling:
R → set of all requests & A → all accepted requests
while R is not empty:
  choose a request i ∈ R that has "smallest finishing time"
  add request i to A
  delete all req. from R that are not compatible with req i (is don't meet "smaller finishing time cond")
end while
return set A

2, 9, 7, 6, 5, 8



Bottom up construction : checking happens from bottom.

Add 10

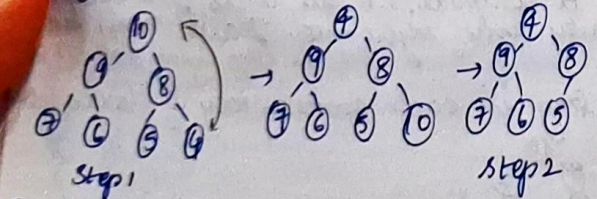


time complexity → $O(\log n)$ → for insertion

↳ insertion can't take more key comparisions than the heap's height. ↳ it is a BT & two children gets added at each node

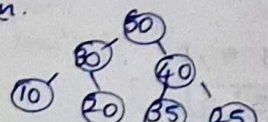
Maximum key deletion from a heap

→ exchange the root's key with last key K of heap [step1]
→ dec. the heap's size by 1 [step2]
→ 'heapify' the smaller tree by shifting K down the tree exactly in the same way we did it in the bottom up heap const. algorithm [Bottom up heap const.] [step3]

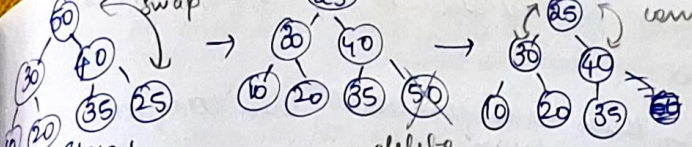


Q) Consider the heap represented as an array [50, 30, 40, 10, 20, 35, 25]

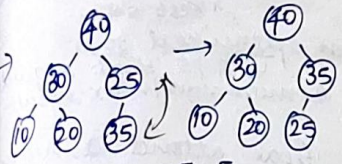
Soln.



construct the BT, check if it's a heap
↳ yes



[50, 30, 40, 10, 20, 35, 25] [25, 30, 40, 10, 20, 35, 50] [25, 30, 40, 10, 20, 35, 25]

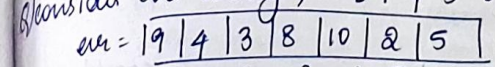


[40, 30, 25, 10, 20, 35] [40, 30, 35, 10, 20, 25]
heapified

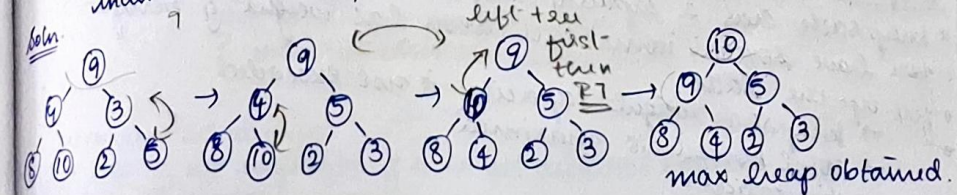
Heapsort

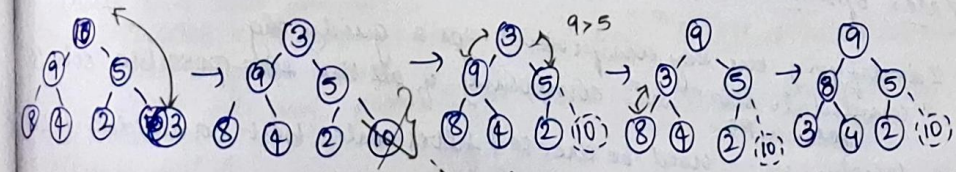
Stage 1: heap const. : construct heap from given array
Stage 2: max key deletion : apply the root deletion operation n-1 times to the remaining heap

Q) Consider the array, as = [9 4 3 8 10 2 5], perform heapsort



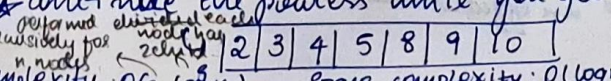
Soln.





(Write array rep for every step)

★ continue the process until you get the sorted order



time complexity: $O(\log n)$ → recursive

visit C (from A)
mark C as visited
explore C's neighbours → D, E

visit D (from C)
mark D as visited
D has no neighbours
stack : [B D]

visit E (from C)
mark E as visited
E has no neighbours
stack [B D E]

Back to C

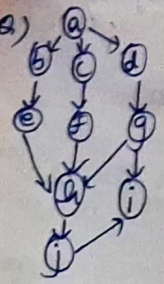
all neighbours visited, add to stack
stack : [B D E C]

Back to A

all neighbours visited
add to stack
Stack : [B D E C A]

Reverse the stack : A C E D B

Q)



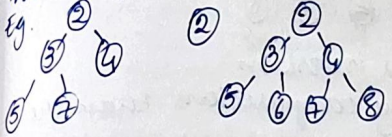
i j h e b f g d a

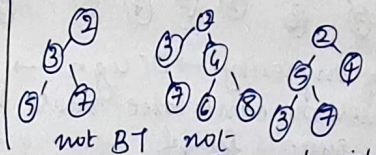
→ A Binary tree is a DS which has atmost 2 children (can't have odd no. of children)
→ Heap DS has a shape such that each node has upto 2 children
→ Each node in the tree stores a key
→ Key is a value to determine the order of the key in the tree
→ Each node in a tree has 1 value i.e., key
- min heap
- max heap

(a) The key at each node is ≥ keys of its children
—— H —— ≤ —— H ——

min heap

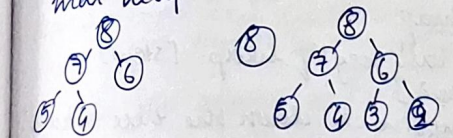
eg.





not BT not BT ↳ violates min heap prop.

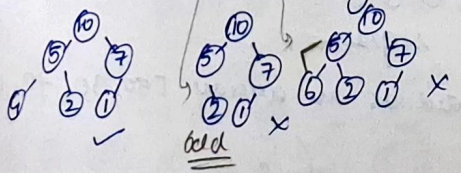
max heap

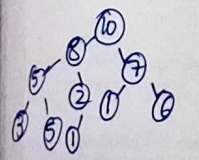


Properties of heap

Shape : the BT should be complete i.e all levels full except last level where only some right-most leaves may be missing.
(next order)
↳ last level of Rigid subtree can have only a left child

Parent al dominance : key in each node ≥ key of its children





index 0 1 2 3 4 5 6 7 8 9 10
val. | | 10 | 8 | 7 | 5 | 2 | 1 | 6 | 3 | 5 | 1 |

parent leaves

order : Root L R

make sure of the two properties (shifting if it doesn't satisfy)