Divide & Conquer:

→ Strategy is based on finding solution to a given problem via its one sub-problem solution.

→ An algorithm, following divide & conquer technique, involves following steps:

↳ Divide the problem into a set of sub-problems.

↳ Solve every sub-problem individually by recursive approach.

↳ Merge the solution of sub-prob into a complete solution of the problem.

Ex: Binary Search
Quick Sort
Merge Sort
Strassen's Matrix Multiplication
Closest pair of Points.

Greedy Technique:

→ This solves an optimisation problem by expanding a partially constructed solution until a complete solution is reached. (optimal global solution)

↳ Greedy Choice is the best alternative available at each step is made in hope that a sequence of locally optimal choices will yield optimal solution

to the entire problem.

↳ This approach works in some cases but fails in others. Usually it is not difficult to design a greedy algorithm itself, but the more difficult task is to prove that it produces an optimal solution.

Ex: Kruskal's MST
Prim's MST
Dijkstra's shortest Path
Knapsack Problem.

Dynamic Programming:

→ A technique widely used to solve optimisation problem.

→ Optimisation problem is used to find the either min or max result out of all possible outcomes.

Ex: Problems can be solved:
Floyd-Warshall and Bellman-ford
0/1 Knapsack Prob.
Chain Matrix Multiplication
Travelling Salesman Prob.

BackTracking Method: (brute force, not used in optimization prob.)

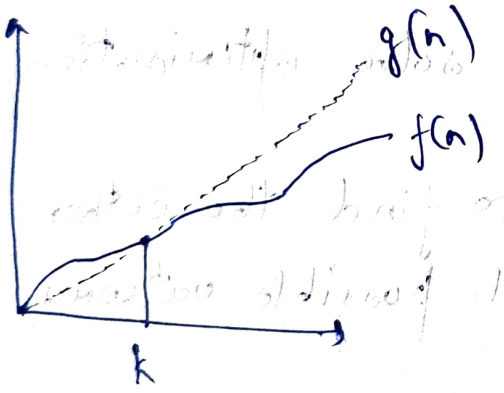
→ Suitable for solving prob. having multiple results & out of which, all or some of them are acceptable.

Ex: Map Coloring
Eight Queen Puzzle
Sudoku

Asymptotic Notations

O Notation (Big oh Not.)

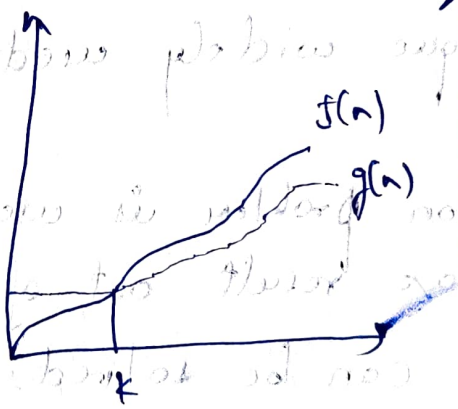
- Express upper bound of algo. running time
- worst case time complexity or longest time it takes.



Ex: $O(f(n)) = \{g(n)$: there exists $c > 0$ & n_0 such that $f(n) \leq c \cdot g(n)$ for all $n > n_0\}$.

Ω Notation (Omega Not.)

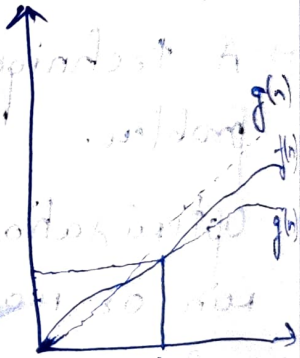
- Express lower bound of algo. running time
- best case time complex. or best time it takes



Ex: $\Omega(f(n)) = \{g(n)$: there ex. $c > 0$ & n_0 such that $g(n) \geq c \cdot f(n)$ for all $n > n_0\}$.

Θ Notation (Theta Not.)

- Express both upper & lower



Ex: $\Theta(f(n))$: $\{g(n)$ if & only if $g(n) = O(f(n))$ & $g(n) = \Omega(f(n))$ for all $n > n_0$.

fractional Knapsack Prob.

→ FKP is one of techniques which are used to solve Knapsack Prob. In Frac. Knapsack, the items are broken in order to maximize profit. The prob. in which we break the item is known as FKP.

→ There is other kind of Knapsack Prob, termed as 0-1 knapsack prob., in which objects are not be considered in fraction.

→ To understand this prob., consider the following instance of KP :

number of obj : $n = 3$

Capacity of knapsack : $W = 20$

$(p_1, p_2, p_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

→ To solve this prob., Greedy method may apply :

↳ from remaining obj, select obj. with max. profit

↳ from remaining obj, select obj with min. weight.

↳ from remaining obj, select obj with max p_i / w_i

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$(1, \frac{2}{15}, 0)$	$18+2+0 = 20$	28.2
2	$(0, \frac{2}{3}, 1)$	$0+10+10 = 20$	31.0
3	$(0, 1, \frac{1}{2})$	$0+15+5 = 20$	31.5

Complexity of FKP:

Example: Suppose There is a knapsack Capacity 5kg, we have 7 items with weight & profit:

Obj	1	2	3	4	5	6	7
weight	2	3	5	7	1	4	1
Profit	10	5	15	7	6	18	3

Selection Criteria will be to select the obj with highest profit per weight. do:

Obj	1	2	3	4	5	6	7
w	2	3	5	7	1	4	1
p	10	5	15	7	6	18	3
p/w	5	1.3	3	1	6	4.5	3

→ According to highest profit per weight, obj 5 is selected. So remaining capacity of knapsack is 4kg. Next, obj 1 is included as available capacity of knapsack is larger that total weight of obj 1. Now, left with 2kg. Next obj 6 but remaing is 2kg so, will take a frac. of obj 6 i.e. 2/4 portion of obj 6. Now, capacity is 0 So, no further obj. will be included

Total weight of included obj. = $1\times2 + 0\times3 + 0\times5 + 0\times7 + 1\times1 + \frac{2}{4}\times4 + 0\times1$

$= 2 + 1 + 2 = 5$

Total profit of included obj. = $1\times10 + 0\times5 + 0\times15 + 0\times7 + 1\times6 + \frac{2}{4}\times18 + 0\times3$

$= 10 + 6 + 9 = 25$

Huffman Coding :

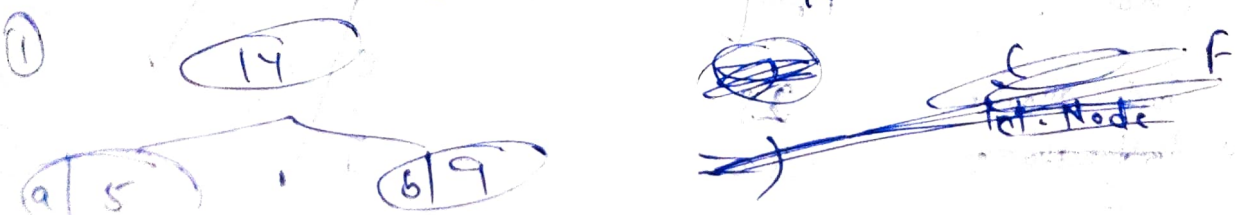
→ The method which is used to construct optimal prefix code is Huffman Coding.
→ Algo. builds a tree in bottom up manner. Can denote by T.

Steps to build Huffman Tree :

→ Input is array of unique character along with their frequency of occurences and output is HT.
↳ Create a leaf node for each unique character & build a min heap of all leaf nodes.
↳ Extract two nodes with min. frequency from min heap.
↳ Create new internal node with freq. equals to the sum of two node freq.
↳ Repeat 2. & 3 until heap contains only one node. The remaining node is root node & tree is complete.

Ex:

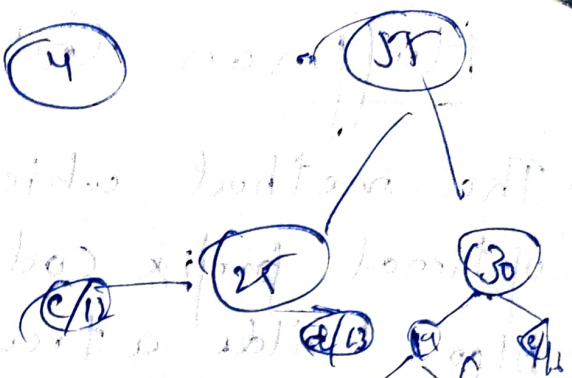
Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



=>

Char.	Freq.
c	12
d	13
Int. Node	14
e	16
f	45

(4)

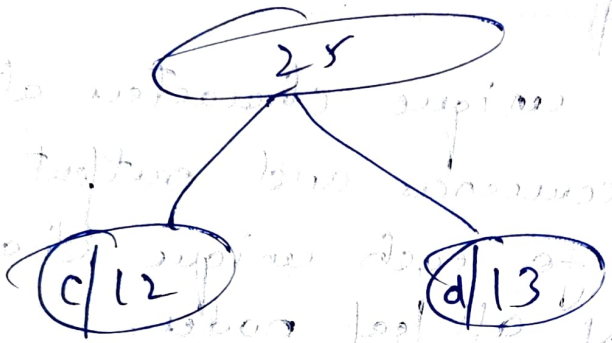


=>

Char.	Freq.
f	45
Int. Node	55

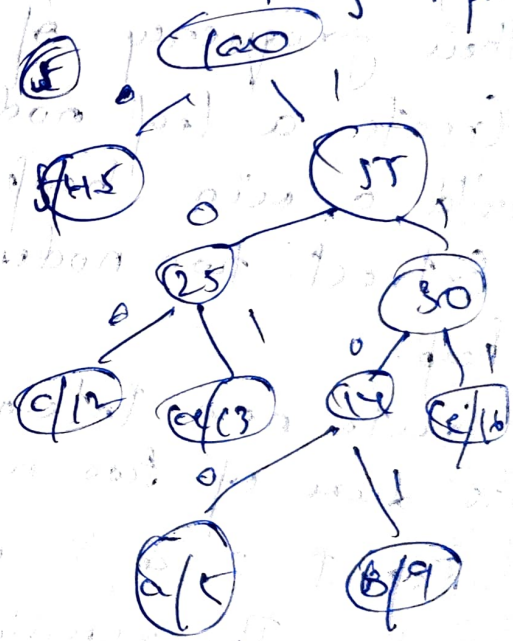
Now, min heap only contain one node, also stop

(2)

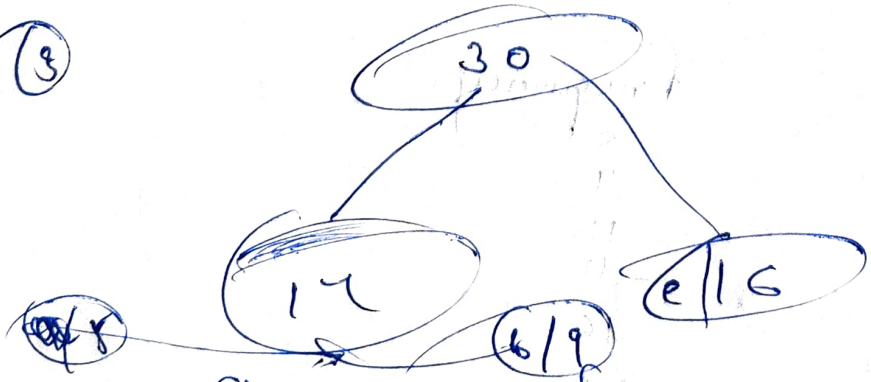


=>

Char.	freq.
Int. Node	14
e	16
Int. Node	25
f	45



(3)



=>

Char.	freq
Int. Node	25
Int. Node	30
f	45

Coded:

Char.	freq
a	
b	
c	
d	
e	
f	

Codes

Char.	Code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Recurrence Relation of Div & Conquer :

$$T(n) = \begin{cases} \theta(1), & \text{if } n \le c \\ aT(n/b) + f(n) \end{cases}$$

where,

$T(n)$ = Time required to solve a prob. of size n

a = no. of partitions made to a prob.

$T(n/b)$ = running time of solving each sub-prob. of size (n/b).

$f(n) = D(n) + C(n)$ – time required to divide prob & Combine solutions.

If prob. size is small enough say $n \le c$ for some const. c we have best case ; which can be solved in a const. time: $\theta(1)$

Otherwise, divide a prob of size n in sub-prob., each of $(1/b)$ size.

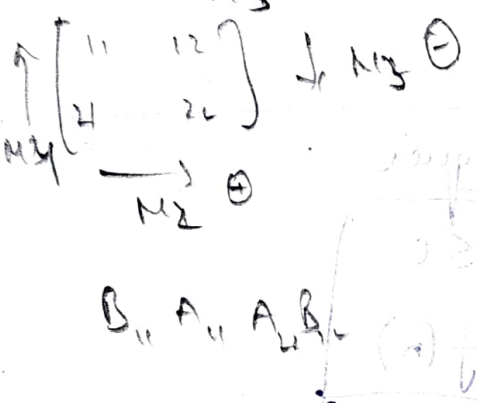
Strassen's Matrix Multiplication Algo. :

→ Uses same divide & conquer approach.

↳ Divide the input matrices A and B into $n/2 \times n/2$ sub-matrices, which takes $\Theta(1)$ time.

↳ Now calculate 7 sub-matrices M1 - M7 by using below formulas:

$$M1 = (A11 + A22)(B11 + B22)$$
$$M2 = (A21 + A22)\,B11$$
$$M3 = A11\,(B12 - B22)$$
$$M4 = A22\,(B21 - B11)$$
$$M5 = (A11 + A12)\,B22$$
$$M6 = (A21 - A11)(B11 + B12)$$
$$M7 = (A12 - A22)(B21 + B22)$$



↳ To get sub-matrices,

$$C11 = M1 + M4 - M5 + M7$$
$$C12 = M3 + M5$$
$$C21 = M2 + M4$$
$$C21 = M1 - M2 + M3 + M6$$

Using above steps, we get

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Example: To perform multiplication of A & B

$$AB = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix}\begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix}\begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 4 & 4 \end{bmatrix}\begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

Graph Traversal Algo:

→ GTA are techniques used to visit and explore all the nodes in a graph.

→ The are two key graphs traversal Depth first Search (DFS) & breadth first Search (BFS) algo.

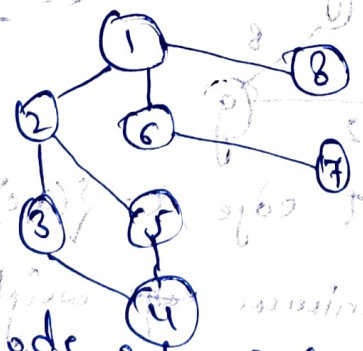
Depth first Search (DFS):

→ A popular graph traversal algo. that explores as far as possible along each branch before backtracking.

→ It starts at a given node and visit its unvisited neighbors recursively until it reaches a leaf node. Then, it backtracks to the previous node and continues the process until all nodes have been visited.

→ Traversal process terminates when all the vertices are traversed.

→ Ex: Graph for DFS & BFS traversal



→ DFS visits each node only once in the adjacency list. Therefore time to complete visiting all edges & associated vertices is $O(V+E)$

→ If a graph is represented through its adjacency matrix, the time to determine all vertices which are adjacent to starting vertex v is V. Since at most V vertices are visited, the total running time is $O(V^2)$

Kruskal's Algo. ⟹ (This is a Greedy Algo.)

→ Sort all edges of given graph in increasing order. Then it keeps on adding new edges & nodes in the MST if the newly added edge does not form a cycle.

→ It picks the minimum weighted edge at first & the max. weighted edge at last.

Disjoint-Set data structure:

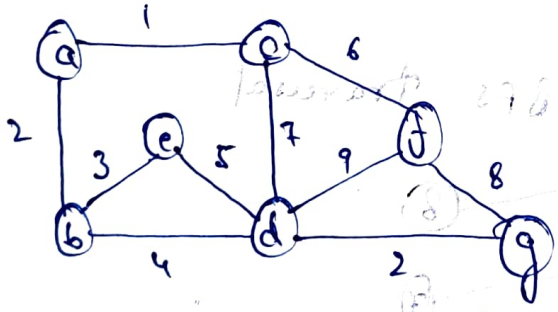
→ It is a datastructure that stores a collection of non-overlapping sets. Following are operations:

↳ MAKE-(v): Create new set

↳ FIND-(u): Returns element

↳ UNI-(u,v): Combines dynamic sets

Ex:

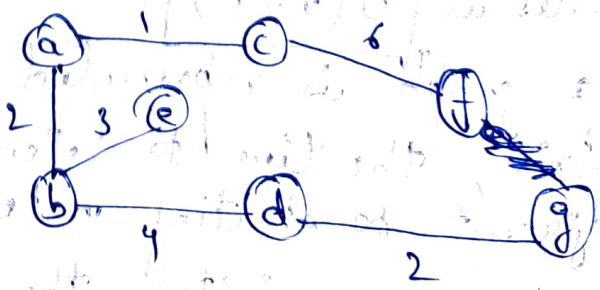


Sorted edges of graph

Consider smallest weight edge, X = {(a,c), (a,b), (d,g), (b,e), (b,d), (c,f)}

Note: (e,d) have minimum ~~edge~~ weight. But when we select that edge it will create a cycle (e→b→d). So we discard & select next.

final MCST of graph:



Cost of spanning tree

= 1+2+2+3+4+6

= 18

Running time for this: $T(n) = O(E \log V)$

Prim's Algo. ⇒ (Greedy Approach to identify MCST)

→ Basic idea of P. Algo is very simple, it finds safe edge and keep it in the set K.

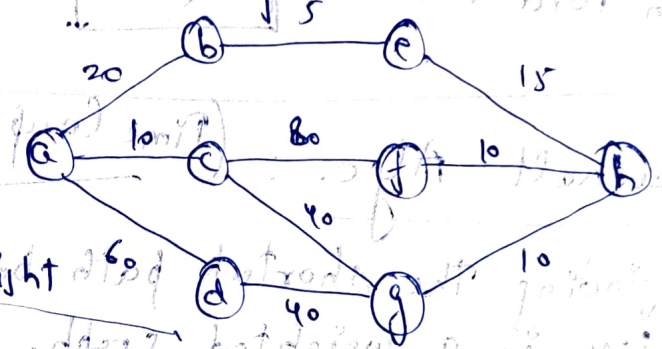
→ Time Analysis = $O(V^2)$

Single Source shortest Path ⟵ Dijkstra's Algo. / Bellman ford Algo.

Dij. Algo ⇒ (Greedy Algo.)

→ solves the single source shortest path problem when all edges have non-negative weights.

→ Example:



consider min. weight

→ Shortest path a → h via a → c → g → h and weight is 10 + 40 + 10 = 50.

→ Always choose the path that are optimal right now not for future consequences.

Time Complexity of Dijkstra's Algo: Depends on priority queue.

The total running time: $((V+E)\log V) = O(E \log V)$

Bellman ford Algo: (Doesn't use Greedy)

→ solves single source shortest-paths problems in case where edge weights may be negative or there is a negative edge weight cycle in the graph.

→ The algo. returns a boolean value TRUE if given directed graph contains no negative cycle that are reachable from the source vertex s, otherwise it returns FALSE.

Time Complexity:

Bellman ford runs in $O(VE)$

Floyd Warshall Algo.: (Time Complexity $O(n^3)$)

→ Algo. for finding the shortest path between all the pair of vertices in a weighted graph.

→ This algo. works for both the directed and undirected weighted graphs. It doesn't work for graphs with negative cycles.

→ Uses Dynamic Programming.

→ Works on concept of recursion.

→ Uses Intermediate vertex (except source & dest. vertex) in shortest path.

→ Recursive fn for FWA can be derived as:

$$d_{i,j}^{(k)} = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min\{d_{i,j}^{(k-1)} + d_{x,j}^{(k-1)}\} & \text{if } k > 0 \end{cases}$$

$d_{i,j}^{(k)}$ ← Weight of shortest path from i to j

$w_{i,j}$ ⇒ Edge weight from i to j

Naive Or Brute force Algo:

→ designed to find the position of a shorter text pattern within a larger text, such as paragraph, book, or any other source.

Time complexity: Let us rewrite the algo:

1) n = length of a text T.
2) m = length of a pattern string P
3) n-m = maximum valid shifts of a pattern in a text
4) s = shift index
5) for (s = 0, s ≤ n-m, s++)
6) if P[1...m] == T[s+1....s+m]
7) Display "Occurence of a pattern string with shift" s

Best Case:

→ It happens if pattern matches in the first m positions of the text. Total no. of comparisons = m (size of the pattern string)

∴ Best time Complex. = O(m)

In worst case scenario, total no. of comparisons: m(m-n+1), ∴ worst case time conf. = O(mn)

Rabin Karp Algo:

→ Concept is to accelerate the pattern matching process by computing hash fn.

→ Algo. calculates hash values for:

① pattern string of m-characters

② m-character substring of a text.

→ If hash values of ① & ② are equal, the algo. will perform brute force comparison b/w pattern string and m-character text substring.

→ In simpler terms, a hash fn takes a large input string and converts it into a smaller value. A good hash fn should have properties such as: efficiently computable & should uniformly distribute the keys & do not spurious hits. (hash fn ⇒ rolling hash fn)

Ex: Text T = baecdddabcdef

Patter P = ecd

Step 1: Hash (P) = $5\times10^2 + 3\times10^1 + 4\times10^0 = 534$] No match

Step 2: Hash (T) = $2\times10^2 + 1\times10^1 + 5\times10^0 = 215$

Hash (bae) (aec) = $1\times10^2 + 5\times10^1 + 3\times10^0 = 153$]

Step 3: will take next 'ecd'

Hash (ecd) = $[1\times10^2 + 5\times10^1 + 3\times10^0] - 1\times10^2]\times10] + 4\times10^0$

$= 53\times10 + 4$

$= 534$ // matches

Robin-Karp Complexity:

Best Case ⇒ $O(n)$, where n is length of a text.

Worst Case ⇒ $O(mn)$, where m is length of a pattern string & n is length of text string.

Knuth Morris Pratt Algo.:

→ KMP is an algo., which checks the characters from left to right.

→ When a pattern has a sub-pattern appear more than one in sub-pattern, it uses that property to improve the time complexity, also for in worst case.

→ Complexity is O(m+n) where m and n are the length of a pattern string and text string respectively.

→ This happens because the KPS algo. avoids frequent backtracking in the text string as it is done in naive algo.

→ The key idea in KMP algo. is to build a LPS array to determine from which point in the pattern string to restart comparing for pattern matching in a text in case there is a mismatch of a character without moving the text pointer backward.

→ Suppose LPS array value of (i-1)th character is 2. This number defines the length of largest prefix which is also a suffix in a pattern string.

→ If length of a pattern string is m then only (m-2) characters will be compared with a text.

Ex: Text: abcfabcxabfabcxabcz

Pattern: abcxabcz

fourth position ⇒ Mismatch (between f & x)

Time Complexity of KMP Algo.:

Worst case time complexity: O(m+n)