Algorithm

LAST MINUTE NOTES

Algorithm – Input, Output, Definiteness, Finiteness, Effectiveness

Analyze an algorithm

1) **Worst Case Analysis (Usually Done**) - calculate upper bound on running time of an algorithm (a situation where algorithm takes maximum time)

MAX

$\boxed{f(n) \le c.g(n)}$

2**) Average Case Analysis (Sometimes done**) - we take all possible inputs and calculate computing time for all of the inputs.

3) **Best Case Analysis** - calculate lower bound on running time of an algorithm.

MIN

Best Case − Minimum time required for program execution.
Average Case − Average time required for program execution.
Worst Case − Maximum time required for program execution.

I/P → O/P DS

2+3 5

4.99

ATM Money withdraw

Assume

1) Building

Best Case

2) Worst Case

Central

Harbour

ATM

3) Average

Central → Vashi

n n/2

Asymptotic Notation

** theorem

O-order

Big O

- O Notation - The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

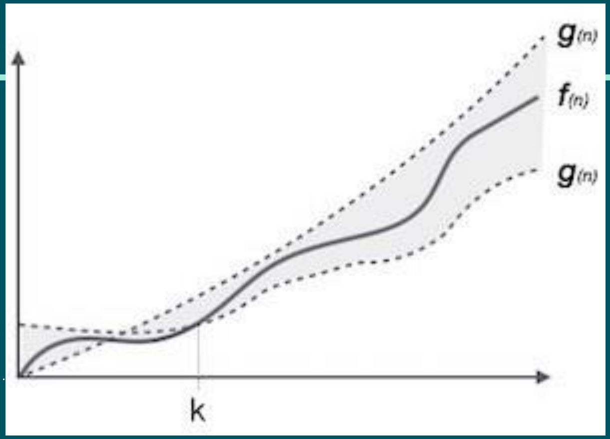
worst case

omega

- Ω Notation - The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

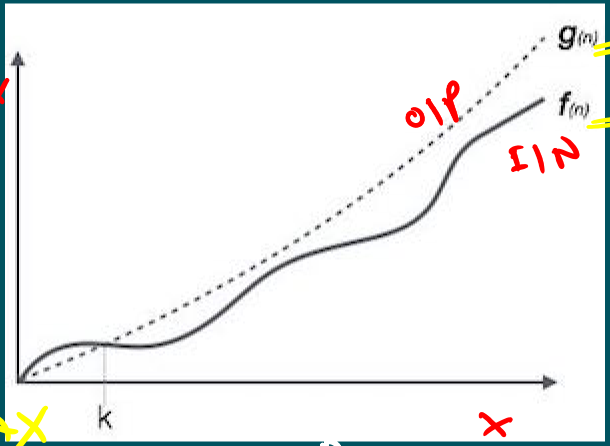
theta

- θ Notation - The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.



θ

$\theta(f(n)) = \{\ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0\}$



O/P

I/N

L

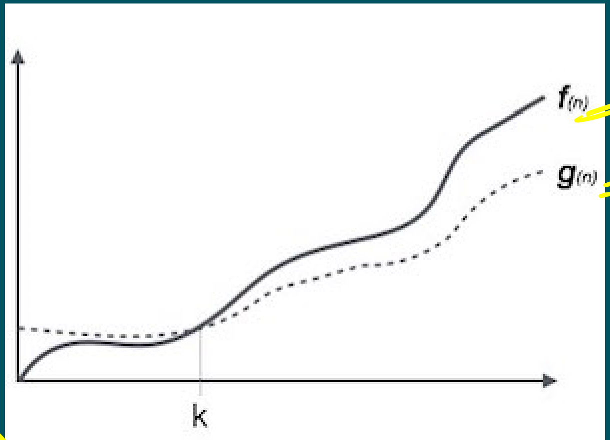
MAX

O(n)

← I input

R.H.S

(1) $\boxed{f(n) \le c.g(n)}$



$\Omega(n)$

(2) $\boxed{g(n) \le c.f(n)}$

1,2,3,4

$\boxed{x = 2+y}$

dependent

indep.

learn

constant	O(1)	less
logarithmic	O(log n)	
linear	O(n)	power 1
n log n	O(n log n)	
quadratic	$O(n^2)$	power 2
cubic	$O(n^3)$	power 3
polynomial	$n^{O(1)}$	n^n
exponential	$2^{O(n)}$	2^n high

Proof

Q1 $n^2 \, 2^{3\log n} \rightarrow O(n^5)$ T/F

Competitive

$2^{\log n} \approx n^{\log 2}$

$\Rightarrow n^2 \, n^3 \log 2$

$\log 2 = ?$ Constant $\Rightarrow$ $\Rightarrow n^2 \cdot n^3 \Rightarrow n^5$

ignore

Worst | Best | Average

Q2 L.H.S $A = n^2$ $f(n)$ R.H.S $B = n^3$ $g(n)$

Big O $O(n)$: L.H.S $\leq$ R.H.S

Omega $\Omega(n)$: L.H.S $\geq$ R.H.S

Theta $\Theta(n)$: L.H.S = R.H.S

$n=1$ $A=1$ $B=1$ Theta

$n=2$ $2^2=4 \leq 2^3=8$ Big O

$n=3$ $2^3=8 \leq 2^4=16$ Big O

$n=n$ n^n n^n Theta

main()
{ for(i=0 ; i< ~~5~~ n ; i++) 5

500 lakh | crore

{
for (   )
}

}

}

logic

execute < finite → O(1)

infinite ∞

n → a

finite → countable

DS → vast

concept → content

cos 0,1

$\log 2 = 1$

→ $\log 4 = \log_2 2^2 = 2\log 2 = 2 \times 1 = 2$

$\log 16 = 4 = \log 2^4 = 4\log 2 = 4 \times 1 = 4$

$a^{\log_n b} = b^{\log_n a}$

$b^{\log_b a} = a$

$\log_b a^n = n \log_b a$

$\log_b\left(\frac{1}{a}\right) = -\log_b a$

$\log_b a = \frac{1}{\log_a b}$

Master algo. easy ⇒

Q1 $T(n) = 8T(n/2) + n^2 \Rightarrow$ Complexity ?

recurrence relation

a, b, $f(n)$

Interview

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

$\leq g(n)$ $O(n)$

$\geq g(n)$ $\Omega(n)$

Binary search

- Sorted
- DAC
- → $O(\lg n)$

n-elem.

$n/2$

$O(\lg n)$

$T(n) =$

Recurrence Relation

$a = 8$

$b = 2$

$f(n) = n^2$

$a, b, f(n) = \boxed{n^{\log a}}$

$= n^{\log 8} = n^{\log 2^3}$

$= n^3$

$f(n) = n^2$

$g(n) = n^3$

$f(n) \leq g(n)$ worst

$\Rightarrow \theta(n^3)$

Q

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

Complexity = ?

Worst Case = ?

$a = 2$

$b = 2$

$f(n) = n^2$

$g(n) = ?$

$n^{\log_b a}$

$g(n) = n^{\log_2 2} = n^{\log 2} = n^1 = n$

$\log 2 = 1$

$$\boxed{f(n) \quad g(n)}$$

$n^2 \geq n$

answer

worst case

$\Omega(n^2)$

$\Theta(n^2)$

$O(n^2)$

Complexity = ?

(1) Recurrence relation

(2) Master theorem (easy)

(3) " Tree

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

$T(n)$ — size of problem; a — no. of subproblem; $\frac{n}{b}$ — size of each subproblem; $f(n)$ — cost

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$

Worst — Case 1 : $f(n) < \underbrace{n^{\log a}}_{g(n)} \Rightarrow T(n) = \Theta(n^{\log a})$ (avg)

Best — Case 2 : $f(n) > n^{\log a} \Rightarrow T(n) = \Theta(f(n))$

Average — Case 3 : $f(n) = n^{\log a} \Rightarrow T(n) = \Theta(f(n) \cdot \log n)$

Q

$$T(n) = 16T\left(\frac{n}{4}\right) + n$$

$$a\,T\left(\frac{n}{b}\right) + f(n)$$

$a = 16 \geq 1$

$b = 4 \geq 2$

$f(n) = n$

$$n^{\log_b a} = n^{\log_4 16} = n^{\log_4 4^2} = n^{2\log_4 4} = n^2$$

$$\boxed{\begin{array}{ccc} f(n) & & g(n) \\ n & < & n^2 \end{array}}$$

ans $= \Theta(n^2)$

$n^{\log_b a}$ $\log 3 = 1.73$

Q1) $T(n) = 3T(\frac{n}{5}) + n$ $= n^{\log_5 3} = n^{1.73}$ $f(n) = n$, $g(n) = n^{1.73}$ $f(n) \simeq g(n)$ $\theta(f(n) \cdot \log n) = \theta(n \log n)$ ans

Q2) $T(n) = 2T(n/2) + 1$ $= n^{\log_b a} = n^{\log_2 2} = n^1 = n$ $f(n) = c$, $g(n) = n$ $\theta(n)$

Q3) $T(n) = 1 \cdot T(n/2) + n$ $= n^{\log_2 1} = n^{0.166} \simeq n^1$ $\theta(\log n) < \theta(n)$ $= \theta(n)$

Q4) $T(n) = 1 \cdot T(n/2) + 1$ $= n^{0.166} > c = \theta(\log n)$

Q5) $T(n) = 2T(n/2) + n \log n$ $= n^{\log_2 2} = n^1 < n^1 \log n \Rightarrow \theta(n \log n)$

Q6) $T(n) = \sqrt{2}\, T(n/2) + \log n$ $= n^{\log_2 \sqrt{2}} = n^{\log_2 \frac{1}{2}} = \sqrt{n} > \log n$

$\theta(\sqrt{n})$

Remark $= n^1 = n$ $n^{0.7} = \log n$

$= n^{1.01} = n$

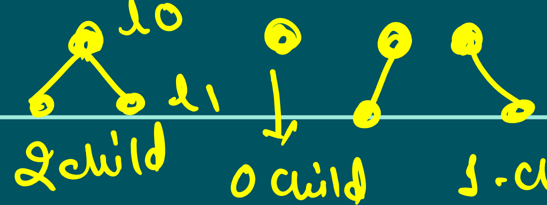
$= n^{1.5} = n$

$2\, n^2$

$.5 \; \sqrt{n} \; 1$

Trees

→ Properties, Root → 1   Versions, Binary Tree



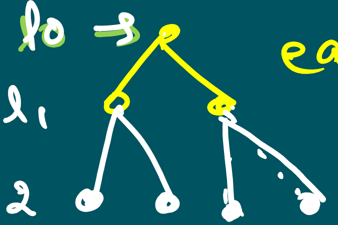
Binary Search Tree (BST)

There must be no duplicate nodes.

Root Node

left less   Right greater

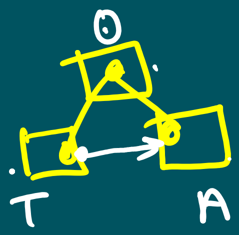
CBT → SBT



each level should be complete

Interview   Skewed   Competitive

AVL Tree- self-balancing Binary Search Tree (BST) , where the difference between heights of left and right subtrees cannot be more than one for all nodes.



Searching / Indexing

B-tree } B+tree → leaf node pointers → list

R&D

Graph

- similar to tree, cycle, directed, undirected
connected | disconnected

MST → Kruskal, Prims, Dijkstra

Traverse → DFS | BFS, TSP

Complexity

Graph

learn

array

linked list

Algorithm	Adjacency Matrix	Adjacency List
DFS	O(V * V)	O(V+E)
BFS	O(V * V)	O(V+E)
Dijkstra	O(V^2)	O(E log V)
Prim's		
Kruskal's		

Graph

Algorithm	Worst Case	Average Case	Best Case
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Bubble	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Quick	$\Theta(n^2)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Merge	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$
Heap	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$

SBI

O(n)

Algorithm	Worst Case	Average Case	Best Case
Topological sorting	$O(V+E)$		
	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
	$\Theta(n^2)$	$\Theta(nlgn)$	$\Theta(nlgn)$
	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$
	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$

SEARCHING

1st position search

n/2

Algorithm	Worst Case	Average Case	Best Case
Linear Search	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Binary Search	$O(\log n)$	$O(\log n)$	$O(1)$

DAC

array half

$O(n) > O(\log n)$

$O(1) > O(\log 1)$

$O(2) > \log 2 = 1$

$O(3) > \log 3$

$n = 1$

$n = 2$

$n = 3$

DAC

1.Divide: Break the given problem into subproblems of same type.

2.Conquer: Recursively solve these subproblems

3.Combine: Appropriately combine the answers

2) <u>Quicksort</u> is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

Divide & Conquer	Recurrence Relation	Time Complexity
Binary Search		O(nLogn)
Quick Sort		O(nLogn)
Merge Sort		O(nLogn)
Closet pair of points		O(nLogn)

1) <u>Binary Search</u> is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element

3) <u>Merge Sort</u> is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) <u>Closest Pair of Points</u> The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum.

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem*.

1) Kruskal's Minimum Spanning Tree (MST): In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

2) Prim's Minimum Spanning Tree: In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3) Dijkstra's Shortest Path: The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4) Huffman Coding: Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

Greedy Approach	Recurrence Relation	Time Complexity
Kruskal's		O(nLogn)
Prim's		O(nLogn)
Dijkstra		O(nLogn)
Huffman Coding		O(nLogn)

Greedy algorithm

Application

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Colouring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

- DP (Dynamic Prog)
- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

- DAC (Divide & Conquer)
- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again

1. **Overlapping Subproblems:** Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed.

2. **Optimal Substructure**: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Dynamic Programming	Recurrence Relation	Time Complexity
Floyd warshall		O(nLogn)
Bellman Ford		O(nLogn)
Longest Common Subsequence		O(nLogn)
		O(nLogn)

BackTracking

Stack

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items .Meaning of item varies from problem to problem.

Example: Hamiltonian Cycle

Search

Interpolation search is an improved variant of binary search. algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed. **O(log (log n))**

$O(\log n)$

$O(\log \log n)$ small