3/09/20 # Design & Analysis of Algorithms (10-12 M)

Text-books:

→ Introduction to Algorithms
- CLR (Coremen)

→ Fundamentals of Algorithms
- Sahani

Syllabus:

1. Analysis of Algorithms

→ Algorithm concept & lifecycle

→ Need for analysis

→ Methodology of analysis

→ Types of analysis

**→ Asymptotic Notations (ASN)
- Definitions & properties
- Problem solving

*→ Framework for analyzing recursive & non-recursive algorithms.

*→ Analysis of Program segments with loops.

→ Space complexities

2. Design Strategies:

i) Divide & Conquer

→ Control Abstraction

→ Max Min

→ Merge Sort

→ Quick sort

→ Binary Search

→ Matrix Multiplication

→ long integer multiplication

→ Master's theorem

(ii) * Greedy Method

→ Control Abstraction

→ Job seq with deadlines

→ knapsack problem

→ Merge patterns

→ Huffman coding

→ Spanning trees

→ Shortest paths

(iii) ** Dynamic Programming

→ General method

→ DP vs GM v D&C

→ Multistage Graphs

→ 0/1 knapsack

→ All pairs shortest path

→ Travelling salesperson.

→ Matrix chain Product

→ Longest common subsequence

→ Bellman Ford algorithm.

→ OBST

→ Reliable System Design

→ kadane Algorithm.

(iv) Graph Techniques

→ Traversals

→ Parenthesization theorem

→ Components; Articulation points.

(v) Heap algorithms

(vi) Sorting techniques

→ Classification

→ Case studies

——— o ———

Algorithm: Consists of finite set of steps / stmts to solve a given problem.

→ Each step/stmt may involve one/more fundamental operation.

→ Every operation must be

(i) Definite (i.e., clear) (unambiguous)

(ii) Effective (i.e., finite time)

→ Every algorithm must halt in finite time.

→ Every algorithm may accept 0 (or) more inputs and must produce atleast one output.

Lifecycle steps:

1. Problem definition
2. Requirements.
3. Design [logic]
4. Develop the algorithm (Pseudo code)
5. Validation (Proving its correctness)
6. Analysis
7. Implementation
8. Testing & Debugging

Analysis

need for analysis

- to determine resource consumption <time, memory, power, cost (registers etc.)>
- Performance Analysis (Comparision blw diff algorithms)

Methodology of analysis :

Aposteriori Analysis (Experimentation analysis)

→ This analysis done by converting algorithm into a program and we analyze by running it on a platform.

This analysis depends on the processors, memory speed, OS, programing language (compiler) i.e., platform dependent.

Advantages:

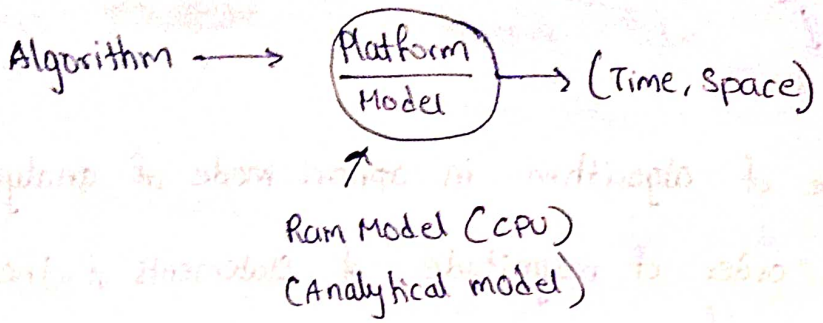
* Gives exact values in units of time/space.

Drawback:

* reports different values from platform to platform
* difficult to make performance comparision.
* Difficult to determine the exact times (∵ OS & other programs could run in parallel).

Hence

Apriori Analysis:- (Platform Independent Analysis)



This is a hypothetical model with infinite memory and we assume that it takes one unit of time for every fundamental operation.

→ This gives estimated time.

Adv:
→ This analysis do help in performance comparision.
→ Easy to carry out.

Components of analytic framework used in apriori analysis

→ A language for describing algorithm (Pseudocode)
→ A computational model (ram model) that algorithms execute within it.
→ A metric for measuring running time of algorithm.
i.e., basic / fundamental operation
→ An approach for characterizing running times of algorithms (i.e., Asymptotic notation)

Algorithm Test — time-apriori mode (Step count method)

```
{
1. x ← y+z ..... 2          (2 → assign)
2. for i ← 1 to n ..... 1+(n+1)+n   (1 → assign, (n+1) → comparisions, n → increaments)
       x ← y+z ..... n+n
3. for i ← 1 to n ..... 1+(n+1)+n
       for j ← 1 to n ..... n+n(n+1)+n.n
           a ← b+c; ..... n.n+n.n
}                                   5+8n+4n²
```

∴ time required is

$T(n) = 4n^2 + 8n + 5$

↓

i/p size

The running time of algorithm in apriori mode of analysis is determined by order of magnitude of statements of the algorithm.

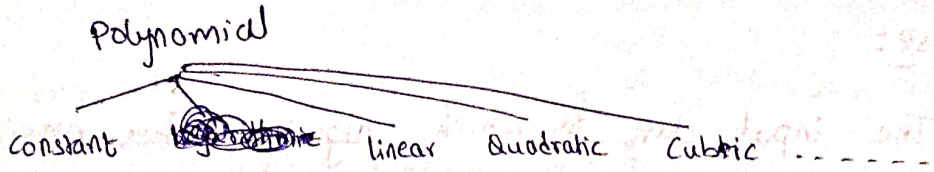
Here, order of magnitude refers to the frequency of the fundamental operation in the statement.

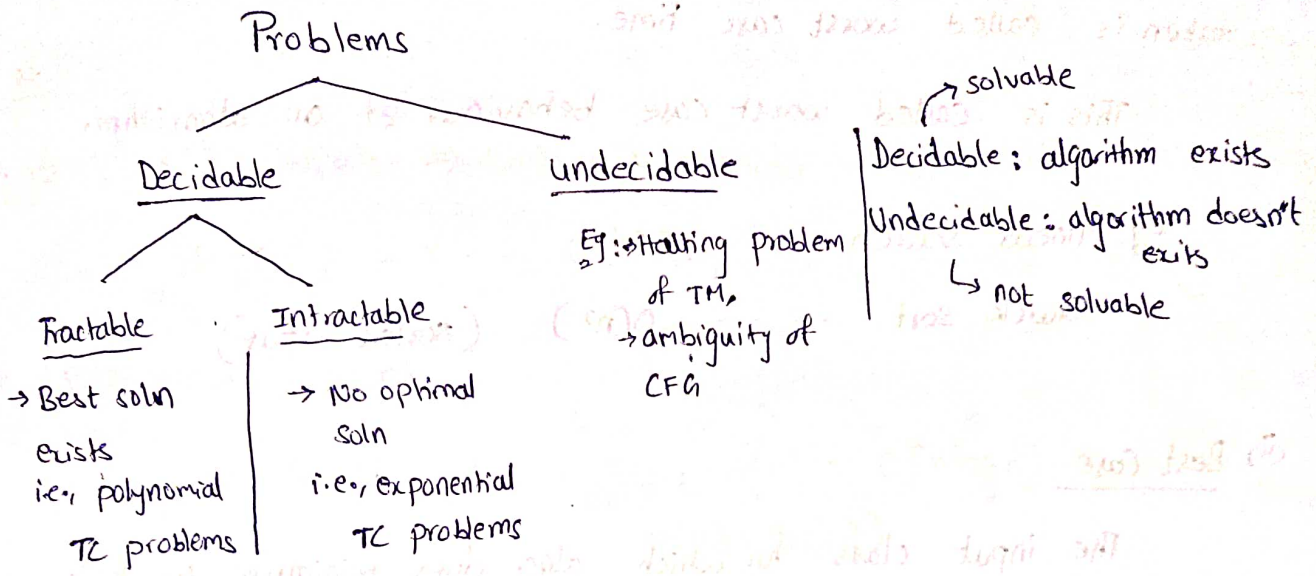
Algorithm Test	Order of magnitude
{	
1. $x \leftarrow y+z$	1
2. for $i \leftarrow 1$ to n { $x \leftarrow y+z$ }	n
3. for $i \leftarrow 1$ to n { for $j \leftarrow 1$ to n { $a \leftarrow b+c$; } }	n^2
}	$T(n) = n^2+n+1$

The objective of Apriori analysis representing running time of algorithm is to represent the time of an algorithm as a mathematical function of input size.

↳ derived or expressed using step count (or) order of magnitude

→ Polynomial func have lesser rate of growth and exponential functions have faster rate of growth.





→ Tractable problems fall under the class of P (Polynomial)

→ NP Problems (Non-Deterministic Polynomial)

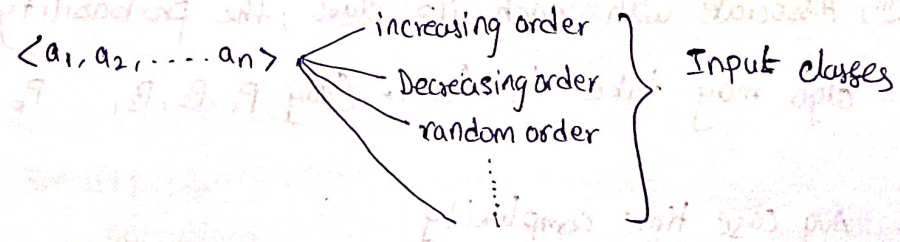
Those are problems that are solvable in polynomial time on non-deterministic TM.

However they take exponential time on deterministic TM.

→ Most of the intractable problems are NP.

Types of Analysis / Behaviour of algorithm:

→ Input of fixed size (say 'n')



Let no of input classes be k.

Based on this we broadly classify the input classes into 3 categories

(i) Worst-Case:

The input class for which algorithm does maximum work is called worst-case input. The corresponding time taken is called worst case time.

This is called worst case behaviour of an algorithm.

Eg: linear search ------ O(n)

Quick sort ------ $O(n^2)$ (sorted input)

(ii) Best Case:

The input class for which algo does minimum work is best case & corresponding time is best case time.

Eg: Linear search ---- O(1)

Quick Sort ------ O(nlogn)

(iii) Average Case Analysis: (Not needed for gate)

This is carried out in 3 steps:

(i) Determine all the input classes.

$\langle I_1, I_2, I_3, \ldots I_k \rangle$ (say size 'n')

(ii) Determine the time taken by the algorithm for each input class. (say $t_1, t_2, \ldots t_k$)

(iii) Associate with each i/p class, the probability with which algo may take i/p from. (say $P_1, P_2, P_3, \ldots P_k$)

Avg case time complexity

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

→ Let $W(n)$, $B(n)$, $A(n)$ be worst case, best case, avg case timings.

$$\Rightarrow \boxed{B(n) \leq A(n) \leq W(n)}$$

Note:

→ $B(n) = A(n) = W(n)$ ⇒ Uniform Behaviour

Eg: Merge Sort ... $O(n \log n)$.

→ $[B(n) = A(n)] < W(n)$

Eg: Quick Sort .... $B(n) = A(n) = O(n \log n)$

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

→ $B(n) < [A(n) = W(n)]$

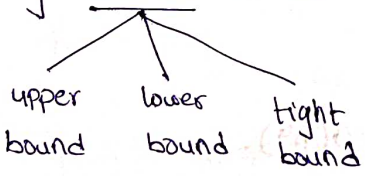
Eg: linear search .... $B(n) = O(1)$

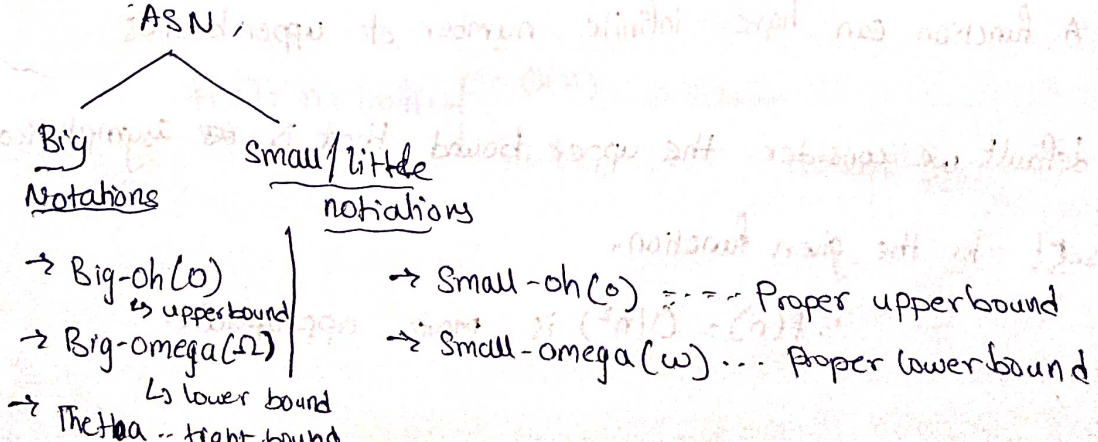
$A(n) = W(n) = O(n)$

→ $B(n) < A(n) < W(n)$

Asymptotic Notations (ASN):

Asymptotic notation is a mathematical tool which is used for representing bounds of a function.





Let 'f' & 'g' be functions from set of real numbers to real numbers.

(i) Big-Oh (O): (upper bound)

$f(x)$ is $O(g(x))$

iff $f(x) \leq c \cdot g(x)$ for some $c > 0$ whenever $x > k$

(k is a constant)

Eg: $f(n) = 1 + n + n^2 = O(n^2)$

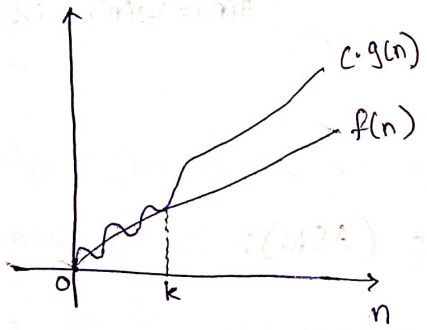
$1 + n + n^2 \leq n^2 + n^2 + n^2$

$\therefore f(n) \leq 3 \cdot n^2, \quad n > 1$

$\downarrow \quad \searrow$

$c \quad g(n)$

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



also $f(n) = 1 + n + n^2 < 3n^3$

$\Rightarrow f(n) = O(n^3)$

$\therefore f(n)$ is $O(n^2), O(n^3), O(n^4) \ldots$

$\therefore$ A function can have infinite number of upper bounds

By default we consider the upper bound that is, asymptotically closest to the given function.

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

(ii) Big-omega

$f(x)$ is

iff

$\therefore$ A fu

$\therefore \Omega$

$\therefore$

Eg: f(n

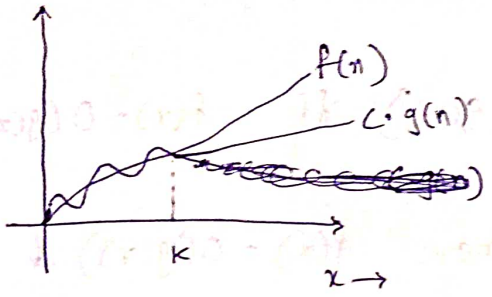
f(n

f(

(ii) Big-omega (Ω): (Lower bound of a function)

$f(x)$ is called $\Omega(g(x))$

iff $f(x) \geq C \cdot g(x)$ for some $c > 0$ &
whenever $x > k$



$f(n) = 1+n+n^2 \geq n^2$

$\Rightarrow f(n) = \Omega(n^2)$ -

$\therefore f(n) = 1+n+n^2 > 1 \Rightarrow f(n) = \Omega(1)$.

$f(n) = 1+n+n^2 > n \Rightarrow f(n) = \Omega(n)$

$\therefore$ A function can have several lower bounds

$\therefore \Omega(n^2)$ is asymptotically closer to $f(n)$

$\therefore f(n) = \Omega(n^2)$ is more appropriate.

Eg: $f(n) = n \rightarrow \begin{cases} O(n) \\ \Omega(n) \end{cases}$

$f(n) = 2^{100} \rightarrow \begin{cases} O(1) \\ \Omega(1) \end{cases}$

$f(n) = n + \log n \rightarrow \begin{cases} O(n) \\ \Omega(n) \end{cases}$ .... $\because n + \log n > n$

(iii) Theta(θ) : (Tight bound)

$f(x)$ is equal to $\theta(g(x))$

iff $c_1 \cdot g(x) \le f(x) \le c_2 \cdot g(x)$ $\quad c_1 > 0 \;\&\; c_2 > 0$ whenever $x > k$

In other words

$f(x) = \theta(g(x))$ iff $f(x) = O(g(x)) \;\&\; f(x) = \Omega(g(x))$

∴ In the cases where $f(x) = O(g(x)) \;\&\; f(x) = \Omega(g(x))$ it is more appropriate representation would be

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

Eg :

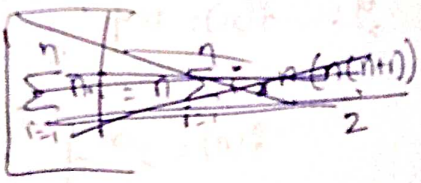
$f(n) = 10^{10} \lessgtr \begin{matrix} O(1) \\ \Omega(1) \end{matrix}$ $\quad \therefore \theta(1)$

$f(n) = n^2 + n\log n + 2^{100} \lessgtr \begin{matrix} O(n^2) \\ \Omega(n^2) \end{matrix}$ $\quad \therefore \theta(n^2)$

$f(n) = \sum_{i=1}^{n} 1 \lessgtr \begin{matrix} O(n) \\ \Omega(n) \end{matrix}$ $\quad \therefore \theta(n)$ $\quad \boxed{\because \sum_{i=1}^{n} 1 = n}$

$f(n) = \sum_{i=1}^{100} 1 \lessgtr \begin{matrix} O(1) \\ \Omega(1) \end{matrix}$ $\quad \therefore \theta(n)$

$f(n) = \sum_{i=1}^{n} n \cdot 1 \lessgtr \begin{matrix} O(n^2) \\ \Omega(n^2) \end{matrix}$



$f(n) = \sum_{i=1}^{n} i \lessgtr \begin{matrix} O(n^2) \\ \Omega(n^2) \end{matrix}$

$f(n) = \sum_{i=1}^{n} O(n) \lessgtr \begin{matrix} O(n^2) \\ \Omega(n^2) \end{matrix}$ $\quad \because \sum_{i=1}^{n} O(n) = O(n) \sum_{i=1}^{n} 1 = O(n^2)$

$f(n) = \sum_{i=1}^{100} O(n) \lessgtr \begin{matrix} O(n) \\ \Omega(n) \end{matrix}$

→ $f(x) = \sum_{i=1}^{n} i^2$ $\begin{cases} O(n^3) \\ \Omega(n^3) \end{cases}$

→ $f(n) = \sum_{i=1}^{n} i^3$ $\begin{cases} O(n^4) \\ \Omega(n^4) \end{cases}$

* → → $f(n) = \sum_{i=1}^{n} \frac{1}{x}$ $\begin{cases} O(\log n) \\ \Omega(\log n) \end{cases}$

$\therefore \sum_{i=1}^{n} \frac{1}{x} = \int_{1}^{n} \frac{1}{x}\, dx$

$= (\log x)_1^n = \log n$

→ $f(n) = \sum_{i=1}^{n} 2^i$

$= 2^1 + 2^2 + \cdots + 2^n$

$= 1 + 2 + 2^2 + \cdots + 2^n - 1$

$= \frac{2^{n+1} - 1}{2 - 1} - 1$

$= 2^{n+1} - 2$

$= O(2^n)$

$\leq 2^2 \cdot 2^n$

$= O(2^n)$

$\therefore f(n)$ is $O(2^n)$

$f(n)$ is $\Omega(2^n)$

Note:

$\sum_{i=1}^{n} O(1) \Rightarrow$ for i=1 to n
a=a+b

$\sum_{i=1}^{n} O(n) \Rightarrow \sum_{i=1}^{n} \sum_{j=1}^{n} O(1)$

$\Rightarrow$ for i=1 to n
for j=1 to n
a=a+b

→ $f(n) = \sum_{i=1}^{n} i \cdot 2^i$

$f(n) = 1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n$

$2 \cdot f(n) = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + n \cdot 2^{n+1}$

$2f(n) - f(n) = -2 - 2^2 - 2^3 - \cdots - 2^n + n \cdot 2^{n+1}$

$\Rightarrow f(n) = 1 - (1+2+2^2+\cdots+2^n) + n \cdot 2^{n+1}$

$f(n) = 1 + n \cdot 2^{n+1} - \frac{2^{n+1}-1}{2-1}$

$f(n) = 2 + n \cdot 2^{n+1} - 2^{n+1} = \underline{(n-1)2^{n+1} + 2}$

$\Rightarrow f(n) = O(n \cdot 2^n)$

$f(n) = \Omega(n \cdot 2^n)$

$$\sum_{i=1}^{n} i \cdot 2^i = (n-1)2^{n+1} + 2$$

$* \rightarrow f(n) = \sum_{i=1}^{n} 1/2^i$

$$= \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^n}$$

$$= \frac{2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2^1 + 1}{2^n}$$

$$= \frac{2^n - 1}{2^n} = 1 - \frac{1}{2^n}$$

$$\sum_{i=1}^{n} 1/2^i = 1 - \frac{1}{2^n}$$

$\therefore f(n) = O(1) \quad \because 1 - \frac{1}{2^n} \leq 1$

$\rightarrow f(x) = \sum_{i=0}^{\infty} x^i$ & when $x < 1$

$\Rightarrow f(x) = \frac{1}{1-x}$ $\left(\because 1-x > 1\right)$

$\therefore f(x) = O(1)$ (~~$\because x$ is a constant~~)

$\rightarrow f(n) = \sum_{i=j}^{n} a^i = \frac{a^{n+1} - a^j}{a-1}$

$\Rightarrow f(n) = O(a^n)$

$\rightarrow f(n) = \prod_{i=1}^{n} 1 = O(1)$

→ $f(n) = \prod_{i=1}^{n} i$

$\Rightarrow f(n) = n!$

$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n < n \cdot n \cdot n \cdots n$ (n times)

$\Rightarrow f(n) < n^n$

$\therefore f(n) = n! = O(n^n)$

However $f(n) \neq \Omega(n^n)$

$n! = \Omega(n)$
$= \Omega(n^2)$
$= \Omega(2^n)$
$= \Omega(an)$

→ $f(n) = \log n!$

$\log n! < \log n^n$

$\Rightarrow \log n! < n \log n$

$\Rightarrow \log n! = O(n \log n)$

$f(n) = n! = O(n^n)$
$n! \neq \Omega(n^n)$
$n! = \Omega(n)$
$= \Omega(n^2)$
$= \Omega(2^n) = \Omega(3^n)$.
$= \Omega(an)$

→ $f(n) = \sum_{i=1}^{n} \log i$

$= \log 1 + \log 2 + \cdots + \log n$

$= \log n!$

$= O(n \log n)$

However $f(n) \neq \Omega(n \log n)$

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

$\sum_{i=1}^{n} \sqrt{i} = \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}$

$\int_{1}^{n} \sqrt{x} \cdot dx = \frac{2}{3} x^{3/2}$

$\therefore O(n^{3/2})$

→ $f(n) = (\log n)!$

$< (\log n)^{(\log n)}$

$\Rightarrow f(n) = O((\log n)^{\log n})$

• also $(\log_e n)^{\log_e n} = n^{\log_e (\log n)}$

$\Rightarrow f(n) = O(n^{\log(\log n)})$ $\quad (\because a^{\log_c b} = b^{\log_c a})$

→ Smaller functions are in the order of bigger functions

i.e., $f(x) = O(g(x))$
- $f(x)$: smaller function
- $g(x)$: bigger function

→ Bigger functions are in the omega of smaller functions.

$f(x) = \Omega(g(x))$
- $f(x)$: bigger function
- $g(x)$: smaller function

Eg: if $f(x) = O(g(x))$ then $g(x) = \Omega(f(x))$

Eg: Consider $f(n) = n^2$ & $g(n) = n \log n$

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

&

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

Small / Little Notations :-

The bounds provided by big Notations (O, Ω) may or may not be asymptotically tight.

Eg: $f(n) = n = O(n)$ is a tight upper bound

where as $O(n^2)$ is a loose upper bound

Sly $f(n) = n = \Omega(n)$ is a tight lower bound

$f(n) = n = \Omega(\log n)$ is a loose lower bound.

However the bound provided by small notations are always loose.

(iv) Small-oh (o) : (Proper upper bound)

$f(x)$ is $o(g(x))$

iff $f(x) < c \cdot g(x)$, for all $c > 0$ & $x > k$

Eg: $f(n) = n^2 = O(n^2)$

$\neq o(n^2)$

$f(n) = n^2 = o(n^3)$ $\quad \because n^2 < c \cdot n^3$, $\forall c$, $n > 1$

$= o(n^4)$

$\vdots$

(v) Small-omega (ω) : (Proper lower bound)

$f(x)$ is $\omega(g(x))$

- iff $c \cdot g(x) < f(x)$ $\quad \forall c > 0$ & $x > k$

$f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^2), \Omega(n \log n), \Omega(n), \ldots$

$f(n) = \omega(n \log n), \omega(n) \ldots$

Analogy b/w ASNs & real numbers:

→ Let f & g be functions ; a & b be real numbers.

$f(n)$ is $O(g(n)) \iff a \leq b$

$f(n)$ is $\Omega(g(n)) \iff a \geq b$

$f(n)$ is $\Theta(g(n)) \iff a = b$

$f(n)$ is $o(g(n)) \iff a < b$

$f(n)$ is $\omega(g(n)) \iff a > b$

Note:

→ $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$

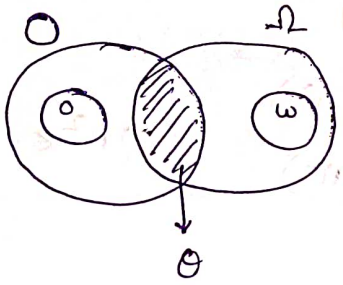
reverse need not to be true.

$\Rightarrow o \subseteq O$

→ $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$

reverse need not to be tree

$\Rightarrow \omega \subseteq \Omega$



Properties of Asymptotic Notations:

i) Discrete Properties:

	Big-Oh	Big-Omega	Thetha	Small-oh	Small-omega
Reflexive	✓	✓	✓	X	X
Symmetric	X	X	✓	X	X
Transitive	✓	✓	✓	✓	✓
Transpose Symmetry	$f(n)$ is $O(g(n))$ iff $g(n)$ is $\Omega(f(n))$		—	$f(n)$ is $o(g(n))$ if $g(n)$ is $\omega(f(n))$	

Reflexive:

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

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

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

$f(n) \neq o(f(n))$

$f(n) \neq \omega(f(n))$

Symmetric:

If $f(n) = O(g(n))$ then it not neccessary that $g(n) = O(f(n))$

∴ Big-oh is not symmetric

Silly Ω is not symmetric

Transitivity:

$f(n) = O(g(n))$ & $g(n) = O(h(x)) \Rightarrow f(n) = O(h(x))$

$\therefore$ ~~transpose~~ transitive

Sily for Ω & theata, o, ω

General Properties:

→ if $f(n) = O(g(n))$ then

$a \cdot f(n) = O(g(n))$

where a is a constant.

→ If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then

(i) $f(n) + d(n) = O(\max(g(n), e(n)))$

(ii) $f(n) * d(n) = O(g(n) \cdot e(n))$

→ ⊛ $\log n^c = O(\log n)$, where c is constant

*→ $(\log n)^x = O(n^y)$ $(x>0, y>0)$

Proof:

$(\log n)^x$ ⊛ n^y

·log on both size

$\log((\log n)^x)$ $\log n^y$

$x \log(\log n)$ $y \log n$

* $x \log(\log n) < y \log n$ $(\because x, y$ are constants$)$

$\therefore$ ~~of $(\log n)^x$~~

$(\log n)^x = O(n^y)$

→ n^2 is $O(a^n)$, $a>1$

$(x$ & a are constants$)$

Dominance Relation:

Constants $\underset{A}{<}$ logarithmic $\underset{A}{<}$ polynomial $\underset{A}{<}$ Exponential

$\underset{A}{<}$ ... Asymptotically lesser.

Eg: Which of the below is asymptotically larger?

$f = \sqrt{n}$ $g = \log n$

Sol:

applying log

$\log(f) = \log \sqrt{n} = \frac{1}{2} \log n$

$\log(g) = \log(\log n)$

$\Rightarrow \log(f) > \log(g)$

$\Rightarrow$ f is asymptotically larger than g

$\Rightarrow g = O(f)$

Tricotomy Property:

tricotomy property of real numbers:

For any two real numbers a & b one of the below 3 conditions must hold.

(i) $a > b$

(ii) $a < b$

(iii) $a = b$

However, for any two functions it is not needed that they satisfy tricotomy property

Eg: $f(n) = n$; $g(n) = n^{1+\sin n}$

f & g here doesn't fall under any case

$(f \underset{A}{<} g,\ f \underset{A}{>} g,\ f \underset{A}{=} g)$

(Q1) Consider the equality $\sum_{i=1}^{n} i^3 = x$ & the following choices for x

a) $\theta(n^4)$ b) $\theta(n^5)$ c) $O(n^5)$ d) $\Omega(n^3)$

Sol:

$$\sum_{i=1}^{n} i^3 = x = \frac{n^2(n+1)^2}{4} = \frac{n^4+2n^3+n^2}{4}$$

$$= O(n^4)$$
(or)
$$\Omega(n^4)$$
(or)
$$\theta(n^4)$$

∴ opt (a) & (b) & (c)

(Q2) Let $w(n)$ & $A(n)$ represent worst case and average case time of an algorithm of input size 'n'. which is always true?

a) $A(n) = O(w(n))$ b) $A(n) = \Omega(w(n))$

c) $A(n) = \theta(w(n))$ d) $A(n) = o(w(n))$

Sol:

$w(n)$ could be some time equal to $A(n)$

∴ opt (d) is not possible.

∴ opt (a)

Eg: Check the correctness of below relations

(i) $100n\log n = O(n\log n)$ --- True

(ii) $2^{n+1} = O(2^n)$ ---- True.

(iii) $2^{2n} = O(2^n)$ ---- False

$\because 2^{2n} = (2^2)^n = 4^n > 2^n$ ∴ False

Note!

→ $\log_b a = \frac{1}{\log_a b}$

→ $\log_b a = \frac{\log_x a}{\log_x b}$

(iv) $(n+k)^m \neq \theta(n^m)$ ----- false.

$(k,m) > 0$

(v) $\sqrt{\log n} = O(\log \log n)$ ----- ~~True~~ False

$\sqrt{\log n}$ $\log \log n$

$\frac{1}{2}(\log \log n)$ $\log(\log \log n)$

$\Rightarrow \sqrt{\log n} > \log(\log n)$

~~$\therefore \sqrt{\log n} = O(\log \log n)$~~

(vi) $\log n = \Omega(1/n)$ ----- True

** (vii) $2^{n^2} = O(n!)$ --------- False

if $2^{n^2} = O(n!)$ then

$2^{n^2} = O(n^n)$ must be true too

$\Rightarrow 2^{n^2} < n^n$

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

$n^2 < n \log n$

which is false

(viii) $n^2 = O(2^{2\log n})$ ----- True

n^2	$2^{2\log n}$
$\log n^2$	$\log 2^{2\log n}$
$2\log n$	$2\log n(\log 2)$

(or)

$2^{2\log_2 n} = 2^{\log_2 n^2} = n^2(\log_2 2) = n^2$

$\therefore$ true

(ix) $a^n \neq$

29/09/20

Asymptotic

(11/02) (i) $f($

(ii)

ix) $a^n \neq O(n^x)$ $a>1, x>0$ ----- True

9/09/20

Asymptotic Comparisions

102

ii) $f(n) = n^2 \log n$, $g(n) = n \log^{10} n$ ~~$\log^2 n = \log(\log n)$~~

$\log^2 n = (\log n)^2$

$\log[f(n)] = 2\log n + \log(\log n)$

$\log[g(n)] = \log n + \log^{11} n$

~~$\log(f(n)) \geq \log(g(n))$~~

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

(or)

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

take common factors aside

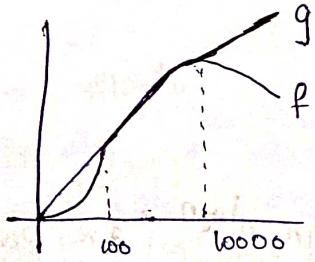
$f(n) = (n\log n)(n)$ $g(n) = (n\log n)(\log n)^9$

So we compare

$f(n) = n$ $g(n) = (\log n)^9$

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

iii) $f(n) = \begin{cases} n^3, & 0 < n \leq 10000 \\ n, & n > 10000 \end{cases}$ $g(n) = \begin{cases} n, & 0 \leq n \leq 100 \\ n^3, & n > 100 \end{cases}$



$\therefore f(n) = O(g(n))$ $\forall n \geq 100$

also $f(n) = o(g(n))$ $\forall n \geq 10000$

H1/03 $10n\log n < 0.0001 n^2$

$10 \cdot \log n < 10^{-4} n$

$\log n < 10^{-5} n$

let $n = 10^x$

$\log_{10} 10^x < 10^{-5} 10^x$

$x < 10^{x-5}$

put $x=1 \Rightarrow 1 < 10^{-4}$ ✗

$x=2 \Rightarrow 2 < 10^{-3}$ ✗

$\Rightarrow x=5 \Rightarrow 5 < 10^0$ ✗

$\Rightarrow x=6 \Rightarrow 6 < 10^1$ ✓

∴ Min value of $x=6$

H#/04 (i) n^2; $n\log n$; $n\sqrt{n}$; e^n; n; 2^n; $1/n$

Sol:

$n\sqrt{n} \neq n\log n$

$\sqrt{n} \gtrsim \log n$

$\Rightarrow n\sqrt{n} \gtrsim n\log n$

$\frac{1}{n} = O(1)$

∴ $\frac{1}{n} < n < n\log n < n\sqrt{n} < n^2 < 2^n < e^n$

(ii) 2^n; $n^{3/2}$; $n\log n$; $n^{\log n}$

Sol:

2^n is exponential

$n^{3/2}$, $n\log n$, $n^{\log n}$ are polynomial

$n^{3/2} > n\log n$

$n \log n$ & $n^{\log n}$

log on both sides

$\log(n \log n)$	$\log(n^{\log n})$	$\log(n^{3/2})$
$\log n + \log(\log n)$	$(\log n)(\log n)$	$\frac{3}{2}\log n$
$O(\log n)$	$O((\log n)^2)$	

$\therefore n^{\log n} \geq_A n \log n$ & $n^{\log n} \geq_A n^{3/2}$

$\therefore$ ~~$n \log n < n^{\log n}$~~

$n \log n < n^{3/2} < n^{\log n} < 2^n$

(iii) $n^{1/3}$; e^n; $n^{7/4}$; $n \log^9 n$; 1.001^n

exponential

$n^{7/4}$	$n \log^9 n$
$\log(n^{7/4})$	$\log(n \log^9 n)$
$\frac{7}{4}\log n$	$\log n + 9 \log(\log n)$
$O(\log n)$	$O(\log n)$

Now we cancel out common terms & compare

$n^{3/4}$	$n \log^9 n$
$n^{3/4}$	$(\log n)^9$
$\log(n^{3/4})$	$\log(\log^9 n)$
$\frac{3}{4}\log n$	$9 \log(\log n)$

$\Rightarrow n^{7/4} \geq_A n \log^9 n$

$\Rightarrow n^{1/3} < n\log^9 n < n^{7/4} < 1.001^n < e^n$

(Hi/06)

g(n) ——————
f(n) ——————

$f(n) = O(g(n))$
$g(n) \neq O(f(n))$
$\Rightarrow$ g(n) is loose upper bound of f(n)

$g(n) = O(h(n))$
$h(n) = O(g(n))$
$\Rightarrow$ g(n) is tight upper bound of h(n) and vice versa
i.e., $g = \Theta(h)$

$\therefore f < (g = h)$

$\therefore f(n) = O(h(n))$ ✓ (transivite property)

$f(n) + h(n) = O(g+h)$ ✓

$h(n) \neq O(f(n))$ ✓

$f(n) \cdot g(n) \neq O(g(n) h(n))$ ×

Framework for analysis of non-recursive & recursive algorithm

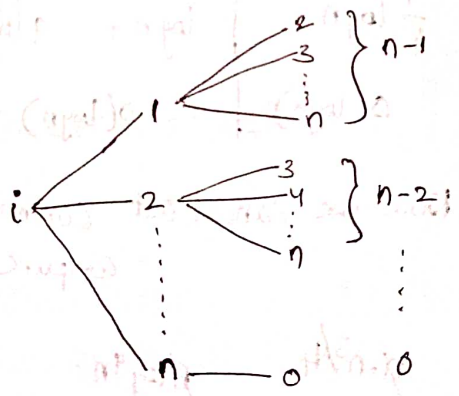
↓ general problems without design strategies.

(Hi/07) Naive/Brute fore

```
Algo Leader1(A,n)
{
    for i=1 to n
    {
        for j=i+1 to n
        {
            if(A[i]<A[j])
                break;
        }
        if(j=n+1)
            print(A[i]);
    }
}
```

TC: $O(n^2)$

no of comparisions



$T(n) = 0 + 1 + 2 + \cdots + n-1$

$= \frac{n(n-1)}{2}$

$\Rightarrow T(n) = O(n^2)$

• Optimized algorithm:

```
Algo LEADER (A,n)
{
    L ← A[n];
    for i ← n-1 downto 1
    {
        if (A[i] > L)
        {
            print(A[i]);
            L ← A[i];
        }
    }
}
```

TC: O(n)

∴ Time complexity of efficient algorithm = O(n)

(H1/08) Let no of levels = k

∴ height = k

$1+2+3+\cdots+k$ = no of nodes = n

$$\frac{k(k+1)}{2} = n$$

$$k^2+k-2n=0$$

$$k = \frac{-1 \pm \sqrt{1+8n}}{2}$$

$$\Rightarrow k = O(\sqrt{n})$$

(H1/09) n MultiQueue operations

$$\Rightarrow O(1)+O(1)+\cdots+O(1) \quad (n \text{ times})$$

$$\therefore O(n)$$

Assuming Queue is not initially empty

(Hi/10) Out of linear & circular queue circular is more efficient. and in circular insertion & deletion can be done in O(1) time.

∴ opt @

Calculate TC for below algorithm

(Hi/11)

```
Algorithm what(n)
{
    if(n==1) return;
    else
    {
        what(n-1);
        B(n)
    }
}
```

a) Assume TC of B(n) = O(1):

TC of recursive algorithm is computed by deriving a recurrence relation.

Let T(n) represent TC of what(n)

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + b, & n>1 \end{cases}$$

a: time for comparision; b: B(n)

$$T(n) = \begin{cases} c, & n=1 \\ d + T(n-1), & n>1 \end{cases}$$

$$T(n) = T(n-1) + d \quad \text{——} ①$$

$$T(n-1) = T(n-2) + d \quad \text{——} ②$$

② in ① ⇒ $T(n) = T(n-2) + 2d$

$= T(n-3) + 3d$

$\vdots$

$T(n) = T(n-k) + kd$

To terminate the condition

$n-k=1$

$\Rightarrow k=n-1$

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

$\boxed{T(n) = c + dn - d}$

$\therefore T(n) = O(n)$
$T(n) = \Omega(n)$ $\Big\}\ \Theta(n)$

b) Assume TC of $B(n) = O(n)$

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + n, & n>1 \end{cases}$$

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

$= a + [a + T(n-2) + n] + n$

$= 2a + T(n-2) + (n + n-1)$

$\vdots$

$= ka + T(n-k) + (n + (n-1) + (n-2) + \cdots + (n-(k-1)))$

$n - k = 1 \Rightarrow k = n-1$

$T(n) = (n-1)a + T(1) + (1 + 2 + 3 + 4 + \cdots + n)$

$= (n-1)a + c + \frac{n(n+1)}{2} - 1$

$= \cancel{n^2 - n + an - a + c}$

$T(n) = \cancel{n^2 + (a+1)n + (c-a)}$

$\therefore T(n) = O(n^2)$

c) Assume TC of $B(n) = O(1/n)$

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + \frac{1}{n}, & n>1 \end{cases}$$

$T(n) = a + T(n-1) + \frac{1}{n}$

$= a + \left(a + T(n-2) + \frac{1}{n-1}\right) + \frac{1}{n}$

$$= 2a + T(n-2) + \frac{1}{n} + \frac{1}{n-1}$$

$$= 2a + \left(T(n-3) + a + \frac{1}{n-2}\right) + \frac{1}{n} + \frac{1}{n-1}$$

$$= 3a + T(n-3) + \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2}\right)$$

$$\vdots$$

$$= ka + T(n-k) + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{n-(k-1)}$$

$n-k=1 \Rightarrow k = n-1$

$$= (n-1)a + T(1) + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2}$$

$$= (n-1)a + c + \underbrace{1 + \frac{1}{2} + \cdots \frac{1}{n}}_{\log n - 1}$$

$$T(n) = (n-1)a + c + \log n - 1$$

$$\Rightarrow T(n) = O(n)$$

Note:

→ If $T(n) = a + T(n-1) + \frac{1}{n}$ then $T(n) = O(n)$

→ If $T(n) = T(n-1) + \frac{1}{n}$ then $T(n) = O(\log n)$

} So constant 'a' also matters.

(H/12)

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n > 2 \end{cases}$$

$$T(n) = a + T(n^{1/2})$$

$$= a + a + T(n^{1/4})$$

$$= 2a + T(n^{1/4})$$

$$= 2a + T(n^{1/2(2)})$$

$$= T(n^{1/2k}) + ka$$

$$= T(2) + \left(\frac{1}{2}\log n\right)a$$

$\Rightarrow \therefore T(n) = O(\log n)$

$n^{1/2k} = 2$

$\frac{1}{2k} = \log_n 2$

$2k = \frac{1}{\log_n 2} \Rightarrow 2k = \log_2 n$

$k = \frac{1}{2}\log n$

01/10/20

(#/12)

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n>2 \end{cases}$$

$$T(n) = a + T(n^{1/2}) \quad \text{--- (1)}$$

$$T(n) = T(n^{1/4}) + 2a$$

$$T(n) = T[(n^{1/4})^{1/2}] + 2a$$

$$T(n) = T(n^{1/8}) + 3a$$

$$\vdots$$

$$T(n) = T(n^{1/2^k}) + ka$$

$$T(n) = T(2) + a \cdot \log_2 \log_2 n$$

$$T(n) = c + a \cdot \log_2 \log_2 n$$

$$\Rightarrow T(n) = O(\log_2 \log_2 n)$$

$$n^{1/2^k} = 2$$

$$\frac{1}{2^k} = \log_n 2$$

$$2^k = \log_2 n$$

$$\Rightarrow k = \log_2 (\log_2 n)$$

Eg: Find TC for below algorithm.

```
Algo A(n)
{
    if(n==2) return
    else
        return A(√n) + n;
}
```

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n>2 \end{cases}$$

$$\Rightarrow T(n) = O(\log_2 \log_2 n)$$

H/14

$$T(n) = \begin{cases} c, & n=2 \\ a + 2T(\sqrt{n}), & n>2 \end{cases}$$

$$T(n) = 2T(n^{1/2}) + a$$

$$= 2\left[2 \cdot T(n^{1/2^2}) + a\right] + a$$

$$= 2^2 T(n^{1/2^2}) + 2a + a$$

$$= 2^3 T(n^{1/2^3}) + 2^2 a + 2a + a$$

$$\vdots$$

$$= 2^k T(n^{1/2^k}) + (a + 2a + \cdots + 2^{k-1}a)$$

$$n^{1/2^k} = 2$$

$$\log n^{1/2^k} = \log_2 2$$

$$\frac{1}{2^k} \log_2 n = 1$$

$$2^k = \log_2 n$$

$$k = \log\log n$$

~~$= 2^{\log(\log n)} T(2) + a(\log\log n)$~~

~~$T(n) = 2^{\log_2 n} + a(\log_2 \log_2 n)$~~

~~$\Rightarrow T(n) = O(\log n)$~~

$$= 2^k T(n^{1/2^k}) + a(2^k - 1)$$

$$= 2^{\log\log n} T(2) + a(2^{\log\log n} - 1)$$

$$= c \cdot \log n + a \cdot \log n - a$$

$$\therefore T(n) = O(\log n)$$

H/13

$$T(n) \begin{cases} c, & n=1 \\ 2T(n-1) + a, & n>1 \end{cases}$$

$$T(n) = 2T(n-1) + a$$

$$= 2\left[2 \cdot T(n-2) + a\right] + a$$

$$= 2^2 T(n-2) + 2a + a$$

$$= 2^2\left[2 \cdot T(n-3) + a\right] + 2a + a$$

H/15 (i) As

$= 2^3 \cdot T(n-3) + 2^2 a + 2a + a$

$\vdots$

$= 2^k T(n-k) + (2^{k-1} + 2^{k-2} + \cdots + 2 + 1)a$

$= 2^k T(n-k) + (2^k - 1)a$ $\quad n-k = 1 \Rightarrow k = n-1$

$= 2^{n-1} T(1) + (2^{n-1} - 1)a$

$= c \cdot 2^{n-1} + a \cdot 2^{n-1} - a$

$= (a+c)2^{n-1} - a$

$\therefore T(n) = O(2^n)$

(H/15) (i) Assume $B(n)$ is ~~O(n)~~ $O(1)$

$\Rightarrow T(n) = \begin{cases} c, & n = 1 \\ 2T\left(\frac{n}{2}\right) + a + 1, & n > 1 \end{cases}$

Comparision $B(n)$

$T(n) = 2T\left(\frac{n}{2}\right) + a$ $\quad (\because a+1$ is const$)$

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

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

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + a\right] + 2a + a$

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

$\vdots$

$\Rightarrow = 2^k T\left(\frac{n}{2^k}\right) + a\left(2^{k-1} + 2^{k-2} + \cdots + 2 + 1\right)$

$= 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)a$ $\quad \frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$

$= 2^{\log_2 n} T(1) + (2^{\log_2 n} - 1)a$

$= nc + an - a$

$\Rightarrow T(n) = O(n)$

Assume B(n) is O(n)

(ii) $$T(n) = \begin{cases} c, & n = 1 \\ 2T(n/2) + a + n, & n > 1 \end{cases}$$

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

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

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

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + a\right] + 2n + 2a + a$$

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

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n + (2^2 + 2 + 2^0)a$$

$$\Rightarrow \quad \vdots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn + (2^{k-1} + 2^{k-2} + \cdots + 2 + 1)a$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn + (2^k - 1)a$$

$$= 2^{\log_2 n} T(1) + (\log n)\left(\frac{n}{n}\right) + (2^{\log n} - 1)a$$

$$\left| \frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n \right.$$

$$= \log n\, c + a \log n$$

$$= \Phi c \log n + n \log n + an - a$$

$$\Rightarrow T(n) = O(n \log n)$$

(4/16)

$$T(n) \begin{cases} 2 & , n=2 \\ \sqrt{n}\,T(\sqrt{n})+n & , n>2 \end{cases}$$

$$T(n) = n^{1/2} T(n^{1/2}) + n$$

$$= n^{1/2}\left[n^{1/4} T(n^{1/4}) + n^{1/2}\right] + n$$

$$= n^{\frac{1}{2}+\frac{1}{4}} T(n^{1/4}) + n+n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}} T(n^{1/4}) + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}}\left[n^{\frac{1}{8}} T(n^{\frac{1}{8}}) + n^{1/4}\right] + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}} T(n^{1/2^3}) + n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^2}} + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}} T(n^{1/2^3}) + 3n$$

$$\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^k} = \frac{\frac{1}{2}\left(1-\frac{1}{2^k}\right)}{1/2} =$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^k}} T(n^{1/2^k}) + kn$$

$$n^{1/2^k} = 2$$

$$\frac{1}{2^k} = \log_n 2$$

$$2^k = \log_2 n$$

$$k = \log\log n$$

$$= n^{1-1/2^k} T(n^{1/2^k}) + kn$$

$$= n \cdot n^{-1/2^k} T(n^{1/2^k}) + kn$$

$$= n \cdot \frac{1}{n^{1/2^k}} T(n^{1/2^k}) + kn$$

$$= n\frac{1}{2} T(2) + n \log\log n$$

$$= n + n\log\log n$$

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

17/17 **Best case:**

Best case is when less no of steps are executed

∴ 2 recursive calls per one call will be called

$$\Rightarrow T(n) = 2T(n/2) + a \qquad n/2^k = 1$$

$$\Rightarrow \vdots \qquad 2^k = n$$

$$= 2^k T(n/2^k) + (2^k - 1)a \qquad k = \log n$$

$$= n\,T(1) + (n-1)a$$

$$\Rightarrow T(n) = O(n)$$

Worst case:

Worst case is when more no of recursive calls are executed per each call

$$T(n) = 8T(n/2) + a$$

$$= 8^2 T(n/2^2) + 8a + a$$

$$= 8^3 T(n/2^3) + 8^2 a + 8a + a$$

$$\Rightarrow \vdots$$

$$\Rightarrow 8^k T(n/2^k) + a(8^{k-1} + 8^{k-2} + \cdots + 8a + a)$$

$$= 8^{\log_2 n} T(1) + a\left(\frac{8^k - 1}{8 - 1}\right) \qquad \frac{n}{2^k} = 1, \quad k = \log n$$

$$= 8^{\log_2 n} T(1) + \frac{a}{7}\left(8^{\log_2 n} - 1\right) \qquad 8^{\log_2 n} = 2^{3\log_2 n} = n^3$$

$$= n^3(1) + \frac{a}{7}(n^3 - 1)$$

$$\therefore T(n) = O(n^3)$$

Note:

From previous problem, we can say

if $T(n) = aT(n/2) + c$ then

$$T(n) = O(n^{\log_2 a}) \quad \left.\right\} a \neq 1$$

Silly if $T(n) = aT(n/b) + c$ then

$$T(n) = O(n^{\log_b a}) \quad \left.\right\}$$

if $a=1$, then $T(n) = O(\log_b n)$ (in both cases)

(H/W) 's' is a string of length n

$$T(n) = \begin{cases} c, & n=0 \\ 2T(n-1)+a, & n>1 \end{cases}$$

$$T(n) = 2T(n-1) + a$$

$$= 2[2T(n-2)+a] + a$$

$$= 2^2 T(n-2) + (2^2-1)a$$

$$\vdots$$

$$= 2^k T(n-k) + (2^k-1)a \qquad n-k=0 \Rightarrow k=n$$

$$= 2^{n-0} T(1) + (2^{n-0}-1)a$$

$$= 2^n \cdot c + 2^n \cdot a - a$$

$$\Rightarrow T(n) = O(2^n)$$

Note:

1) $T(n) = T(n-1) + c$ ——— $O(n)$
2) $T(n) = T(n-1) + n$ ——— $O(n^2)$
3) $T(n) = T(n-1) + \frac{1}{n} + a$ ——— $O(n)$
4) $T(n) = T(n-1) + \frac{1}{n}$ ——— $O(\log n)$

5) $T(n)$ ...
6) $T(n) = 2T($
7) $T(n) = T(\sqrt{}$
8) $T(n) = 2T($
9) $T(n) = 2T$
10) $T(n) = 2T$
11) $T(n) = T($

Analysing runnin

→ for i ← 1 to
c = c+
⇒

→ for i ← 1 to
for j ←
c

→ for i ← 1
for j ←
for

5) $T(n) = 2T(n-1)+c$ ------ $O(2^n)$

6) $T(n) = 2T(n-i)+n$ ------ $O(2^n)$ --- solve it

7) $T(n) = T(\sqrt{n})+c$ ------ $O(\log\log n)$

8) $T(n) = 2T(\sqrt{n})+c$ ------ $O(\log n)$

9) $T(n) = 2T(n/2)+c$ ------ $O(n)$

10) $T(n) = 2T(n/2)+n$ ------ $O(n\log n)$

11) $T(n) = T(n/2)+c$ ------ $O(\log n)$

Analysing running time of program segments with loops:

→ for i ← 1 to n
  c = c+1; ---- repeats n times.

⇒ T.C : $O(n)$

→ for i ← 1 to n ---------- n times.
  for j ← 1 to n/2 ----- n/2 time
    c = c+1; ---------- n · n/2 time

T.C = $(n)(\frac{n}{2}) = \frac{n^2}{2} = O(n^2)$

If initial value of c is 0, then value of c in the end will be $\frac{n^2}{2}$

→ for i ← 1 to n ---------- n times.
  for j ← 1 to n/4 ---------- n/4 times
    for k ← 1 to n ---------- 1 time (i.e., n · n/4 times)
      break; ---------- n · n/4 time

$\therefore TC : n \cdot \frac{n}{4} \Rightarrow TC : O(n^2)$

→
```
for (i=1; i<=n; i++)  ---------- n
    for (j=1; j<=n; j++)  ---------- n
        for (k=n/2; k<=n; k=k*n/2)  ---------- 2
            c=c+1;
```

TC: $n*n*2 = O(n^2)$

value of c in the end = $2n^2$

→
```
i=1;
while (i<=n)
{
    i=i*2;
}
```

Assume that above loop is repeated k times

$\Rightarrow 2^{k+1} > n$

$k+1 > \log n$

$\Rightarrow k > \log n - 1$

$\therefore$ TC: $O(\log n)$

we repeat the loop until below condition is violated

$2^k \leq n$

$k \leq \log_2 n$

$\Rightarrow$ Time... $O(k) = O(\log n)$

→
```
i=n;
while(i>0)
{
    i=i/2;
}
```
TC: $O(\log n)$

→
```
for (i=1; i<=n; i++)  ---------- n
    for(j=1; j<n; j=2*j);  ---------- log n
        c=c+1;
```
TC: $O(n \log n)$

→ $m = 2^n$

```
for(i=1; i<=n; i++)  ------ n
    for(j=1; j<=m; j=2*j)  ------ log2 m = n
        c=c+1;
```

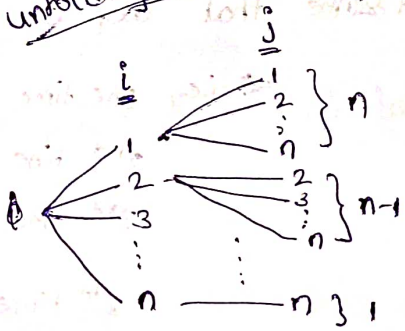
TC : $O(n^2)$

unfolding has to done if loop variables depend on each other

→ for $i \leftarrow 1$ to n ------ n

for $j \leftarrow i$ to n ----- $n-i$

$c = c+1;$



$TC = 1+2+3+\cdots+n$

$= \frac{n(n+1)}{2} = O(n^2)$

value of $c = \frac{n(n+1)}{2}$

→

```
i=n;
while (i>0) ------ log2 n
{
    j=1;
    while (j<=n) ------ log2 n
    {
        j=2*j; ------ ∴ (log2 n)(log2 n)
    }
    i=i/2;
}
```

j=2*j has max frequency

i.e., it is executed $(\log_2 n)^2$ times

$\therefore TC: \quad O(\log^2 n)$

→
```
k=1; i=1;
while (k<=n)
{
    i++;
    k= k+i;
}
```

Assume that loop is repeated x times

i.e., one time for k=1

2nd time for k=3

3rd time for k=6

⋮

x^{th} time for $k = 1+2+\cdots+x$

$\Rightarrow 1+2+3+\cdots+x \leq n$

$\frac{x^2+x}{2} \leq n$

~~$x^2+x-2n \geq 0$~~

~~$x^2+x \geq n$~~ $x^2+x \leq 2n$

$\therefore x$ is $O(\sqrt{n})$

(pyq)

→
```
int fun(int n)
{
    int i,j,P,q=0;
    for(i=1; i<=n; ++i)
    {
        P=0;
        for(j=n; j>1; j=j/2)
            ++P;
        for(k=1; k<P; k=k*2)
            ++q;
    }
    return(q);
}
```

For the function given, to find order of value returned by q.

$q = O(?)$

j loop runs $\log_2 n$ times

$\Rightarrow p = \log_2 n$

k loop runs $\log_2 p$ times

$\Rightarrow$ i.e., $\log\log n$ times

$\Rightarrow$ for every outer loops iteration q valued is increamented by $\log\log n$.

Outer loop runs 'n' times.

$\therefore$ order of value q is $n\log\log n$

→ Find order of value returned.

```
int fun(int n)
{
    int i, j, p, q=0;
    p=0;
    for(i=1; i<=n; i++)
    {
        for(j=n; j>1; j=j/2)
            ++p;
        for(k=1; k<p; k=k*2)
            ++q;
    }
    return q;
}
```

TC of the program on the left side is $O(n\log n)$ ↳ Solve it

For every 'j' loop execution p is increamensed by $\log n$

i=1	i=2	i=3	----	i=n
$p=\log n$	$p=2\log n$	$p=3\log n$	----	$p=n\log n$
$q=\log p$	$q=\log p$	$q=\log p$	----	$q=\log p$
$=\log\log n$	$=\log(2\log n)$	$=\log(3\log n)$	----	$=\log(n\log n)$

$\Rightarrow q = \log(\log n) + \log(2\log n) + \log(3\log n) + \cdots + \log(n\log n)$

$= \log(\log n) + \log 2 + \log^{\log n} + \log 3 + \log\log n + \cdots + \log n + \log\log n$

$= n\log(\log n) + \log n!$ $\Rightarrow$ Order $q = O(\log n!) = O(n\log n)$

→ Find TC of below code

```
for (i=1; i<=n; i++) ---------- n
{
    j=1; ---------- 1
    while (j<=n) ---------- log n
        j=2*j;
    for (k=1; k<=n; k++) ---- n
        c=c+1;
}
```

Time = $n\,(1+\log n+n)$

$= n+n\log n+n^2$

$\Rightarrow$ TC: $O(n^2)$

→ ~~$n=2^k$~~ $n=2^k$

```
for (i=1; i<=n; i++)
{
    j=2;
    while (j<=n)
    {
        j=j*j;
        print("*");
    }
}
```

for one iteration of outer loop

$j=2^{2^0}$	2^{2^1}	$(2^2)^2=2^{2\cdot2}=2^{2^2}$	$(2^4)^2=2^{2\cdot4}=2^{2^3}$	$(2^8)^2=2^{2^4}$	----	$j=2^{2^k}=n$
*	*	*	*	*		*

while loop runs k+1 times

$\Rightarrow$ k+1 '*'s are printed

$\therefore$ total no of stars printed $= n(k+1)$

$= n(\log\log n+1)$

$n=2^{2^k}$

$2^k=\log n$

$k=\log\log n$

→ A: Array [1....n] of binary

```
f(m) = O(m)
count : integer;
count = 1;
for i=1 to n
{
    if ( A[i]==1) count ++;
    else
    {
        f(count);
        count = 1; // line 1
    }
}
```

(i) What is best case & worst case TC for above program?

Sol:

Case I:

all bits are 1

TC: O(n)

Case II: all bits are 0

TC: O(n)

Case III: A[1 to n-1] = 1 & A[n] = 0

TC: (n-1) + O(n) = O(n)

Case IV: ~~first half~~ A[1 to n/2] = 1 A[n/2+1 to n] = 1

~~time~~: i = 1 to n/2 i = n/2+1 i = n/2+2 to n

time: n/2 $\frac{n}{2}+1$ O(n/2-1)

=> TC: O(n)

Case V:

Alternative 1's & 0's

TC: O(n)

* Here we can observe that irrespective of the sequence the TC is $O(n)$.

$\therefore$ Best case : $O(n)$

Worst case : $O(n)$

(ii) First Deleting line 1, determine the best & worst case TC

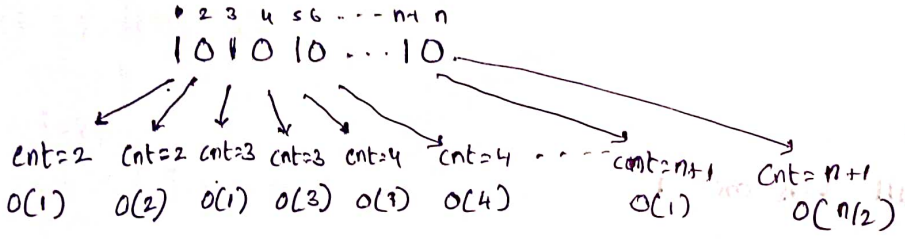
Sol:

If all inputs are 1 then

$O(n)$ --- Best case.

Worst Case:

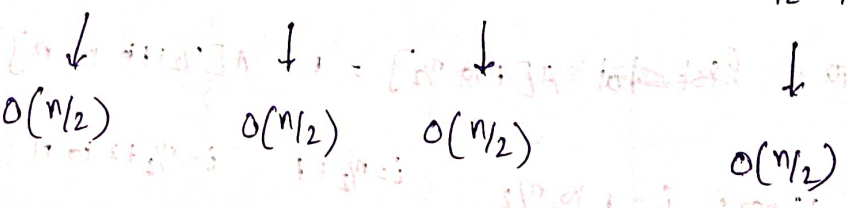
i) Consider case with alternate 1's & 0's



$\Rightarrow O(1)\frac{n}{2} + O[2+3+4+\cdots+n/2]$

$\Rightarrow O(n^2)$ ------ worst case

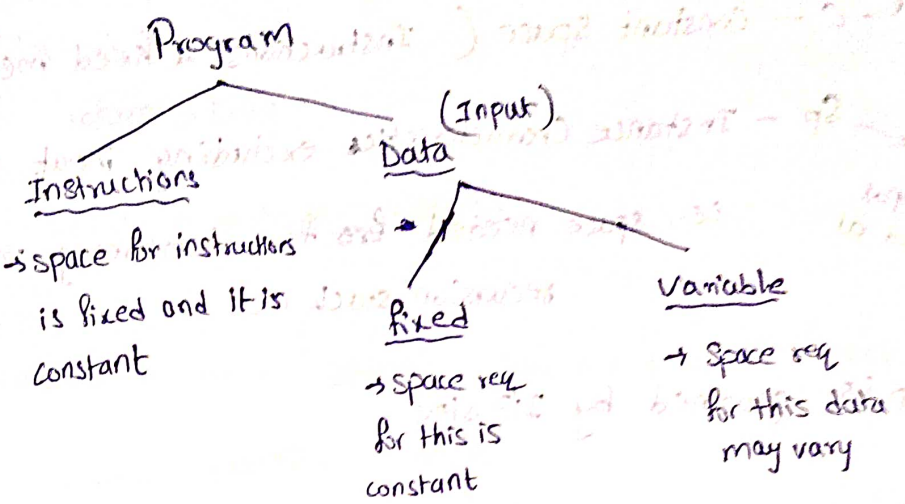
ii) Consider case with first half are 1's and next half are 0's



$\therefore$ time $= O(n/2) + \frac{n}{2} \cdot O(n/2)$

$= O(n^2)$ --- worst case

Space Complexity



→ space complexity is not for program instructions or not for data.

Defn:

The space used by an algorithm is defined as the number of memory words needed to carry out the computational steps required to solve an instance of the problem, excluding the space allocated to hold the input. In other words, it is only work space required by the algorithm.

→ All definitions of order of growth & asymptotic bounds pertaining to time complexity equally applies to space complexity.

→ The work space (space complexity) cannot exceed the running time of an algorithm, since writing into each memory cell requires atleast a constant amount of time.

i.e., if $T(n)$ = time complexity &

$S(n)$ = ~~sap~~ space complexity then

$$S(n) = O(T(n))$$

Space Complexity (P) = C + S_p

C — Constant Space (Instructions & fixed memory)
determined prior to the execution

S_p — Instance Characteristics excluding input space.
depends on variable input and determined at runtime
i.e., space needed for temporary variable, recursion stack etc.

→ S_p is governed by I/P size

Eg: void swap (int a, int b)
{
int t;
t = a;
a = b;
b = t;
}

input
additional variable

TC : O(1)

→ Now 't' contributes to space complexity

size (t) : workspace : time complexity

size (t) : 2 bytes = O(1)

∴ S = C + S_p
= 2 + 0
= O(1)

(I)

Eg: Algorithm Sum (A, n)
A[1...n], n: int
{
int i, sum = 0;
for i ← 1 to n
sum = sum + A[i];
}

TC : O(n)

A & n are inputs
∴ A & n doesn't contribute is Space Complexity

variables i, sum contribute to space complexity

S = C + S_p = 4 bytes + 0
= O(1)

Eg: Algo Rsum(A,n)

(II)

```
{
    if(n==1)
        return A[1];
    else
        return (Rsum(A,n-1)+A[n]);
}
```

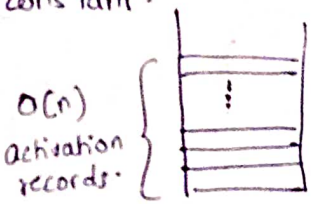
Since recursion is used we need consider size of stack used.

stack constains activation records.

Activation record has formal parameters, local variables, return address etc.

∴ size of activation ~~vector~~ record is constant.

Here we need O(n) activation records.



↓
workspace
of this algorithm

∴ amount of space used in stack

$= O(n) * C = O(n)$

∴ Space Complexity = $C + S_p$

$= 0 + O(n)$

$= O(n)$

Time complexity = $O(n)$

Observation:

Both algo I & algo II does the same job of computing sum of array elements.

Algo I < TC: O(n), SC: O(1)

Algo II < TC: O(n), SC: O(n)

∴ Algo I is space efficient.

Algo II is space inefficient.

General rule:

An algorithm is said to be space efficient, if its space requirement is $O(1)$ or atmost $O(\log n)$ in case of recursive algorithm.

Eg:

```
Algo Test (A,n)
A[1..n, 1..n] , n:int ← input
{
    integer B[1...n], i;
    for i←1 to n
        B[i] = A[i,i]
}
```

TC: $O(n)$

B, i are considered workspace.

$S = C + S_p$

$= \downarrow$ size(i) + size(B)

$= k + O(n)$

$\therefore SC = O(n)$

Eg:

```
Algo A(n)
{
    if (n=1) return;
    else
    {
        A[n/2];
    }
}
```

TC: $O(\log n)$

Space complexity is calculated by amount of space used by stack.

⇒ no of recursive calls = $\log n$

⇒ * space = $(\log n)$ (size of activation record)

$S = C + S_p = 0 + \log n * c = O(\log n)$

Eg:

```
recur(n)
{
    if(
    els
    {
    }
}
```

space

recursive

Eg: A(n)

```
{
    if(n=2) return;
    else
        return (A(√n));
}
```

Space complexity in this case is order of no of rec. calls.

let S(n) denote no of recursive calls.

$$\Rightarrow S(n) = 1 + S(\sqrt{n})$$
$$= 1 + S(n^{1/2})$$
$$= 1 + 1 + S(n^{1/2^2})$$
$$= 2 + S(n^{1/2^2})$$
$$\vdots$$
$$= k + S(n^{1/2^k})$$

$n^{\frac{1}{2^k}} \leq 2$

If k is no of recursive calls.

then $n^{1/2^k} = 2$

$\Rightarrow 2^k = \log_2 n$

$k = \log\log n$

$\therefore$ Space Complexity $= O(\log\log n)$

Eg: recur(n)

```
{
    if(n=1) return;
    else
    {
        recur(n/2);
        recur(n/2);
        B(n);
    }
}
```

let S(n) denote no of recursive calls.

f space

~~S(n) = order of no of recursive calls~~

$= 1 + 2S(n/2)$

$= 2[2 \cdot S(n/2^2) + 1] + 1$

$= 2^2 S(n/2^2) + 2^2 - 1$

$\vdots$

$= 2^k S(n/2^k) + 2^k - 1$

~~If~~ ~~we are no of recursive calls then~~

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

$\Rightarrow k = \log_2 n$

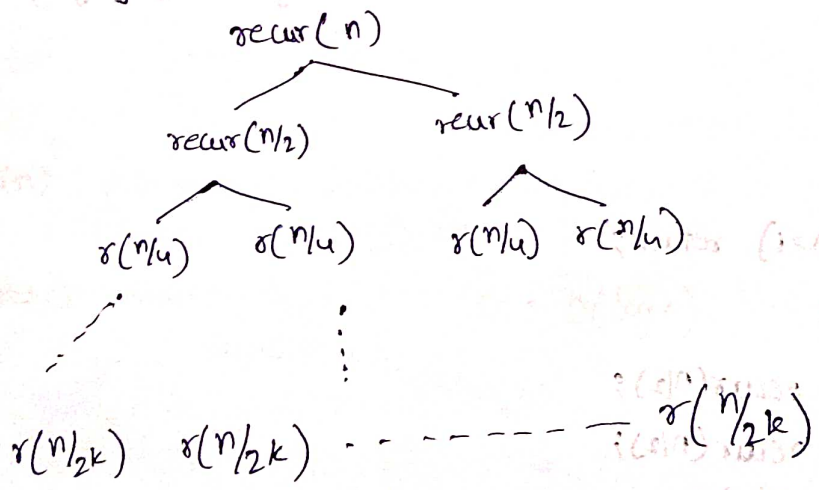
$S(n) = 2^{\log_2 n} S(1) + 2^k - 1$

$= n + n - 1$

$= 2n - 1$

$\therefore$ no of recursive calls $= 2n - 1$

However observing the recursion stack, the depth of recursion stack is only k..



$\therefore$ Space complexity = depth of recursion stack

$= O(k) = O(\log n)$

~~Design Strategies~~

~~Divide & Conquer etc.~~

→ Find value of c, & time complexity for below code segment.

```
for i ← 1 to n-1
    for j ← i+1 to n
        for k ← 1 to j
            c = c+1;
```

TC: $O(n^3)$

$$c = \frac{(n-1)(n)(n+1)}{3}$$

→ Find value of c & TC for below code segment

```
for (i=1; i<=n; i++)
    for (j=1; j<=i*i; j++)
        if ((j%i)==0)
            for (k=1; k<=j; k++)
                c=c+1;
```

TC: $O(n^4)$

$$c = \frac{n(n+1)(3n^2+7n+2)}{24}$$

04/10/20

Divide & Conquer

→ This strategy is used when the problem (its input) is large / complex.

→ So we divide it into subproblems until the subproblem is small.

→ A subproblem is said to be small, if it can be solved in one / two fundamental (basic) operations.

→ Combine (conquer) the results of smaller problems to get the result of the original problem.

This step of combining is optional. It is done only if it is needed.

Eg: quicksort uses only dividing but doesn't perform combining.

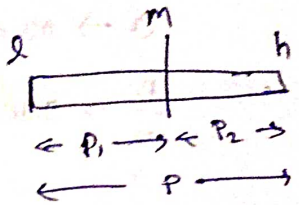
→ Division can be done into 2 or more problems and each subproblem can be of different sizes.

Control Abstraction: → A procedure whose flow of control is clear but the inner details are abstracted.

```
Algorithm DandC(P)
{
    if (SMALL(P)) then
        return(S(P)); // S(P) is soln of smaller problem.
    else
    {
        Divide P into smaller problems P1, P2, ... Pk (k>=1) // Mandatory
        Apply DandC to each subproblems
        return(COMBINE(DandC(P1), ... DandC(Pk))); // optional
    }
}
```

→ The below ~~pro~~ procedure is a general procedure for dividing problem into 2 subproblems.

```
Procedure DandC(P,l,h)
{
    if (SMALL(l,h))
        return G(P,l,h);
    else
    {
        m ← Divide(l,h);
        Soln1 ← DandC(P,l,m);
        Soln2 ← DandC(P,m+1,h);
        return COMBINE(soln1, soln2);
    }
}
```

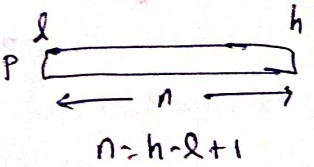


→ The TC of D&C problems is expressed in the form of DandC recurrence equations.

size of the problem (i/p)

Let T(n) represent time complexity of DandC(n)

Consider we divide the problem into 2 subproblems



$$T(n) \begin{cases} f(n) & , n \text{ is small} \\ 2T(n/2)+g(n) & , n \text{ is large} \end{cases}$$

$f(n)$ ⟶ time for SMALL & function's (in most of the cases this is O(1))

$g(n)$ ⟶ time for SMALL, dividing & combining

The general form of Dand C Recurrence: (for symmetric Dand C) i.e., equal size subproblems.

$$T(n) \begin{cases} f(n) & , n \text{ is small} \\ a \cdot T(n/b) + g(n) & , n \text{ is large} \end{cases}$$

$a \rightarrow$ no of subproblems $\therefore a \geq 1$

$\frac{n}{b} \rightarrow$ size of each subproblem $\therefore b > 1$ ($\because$ problem size has to reduce when it is divided)

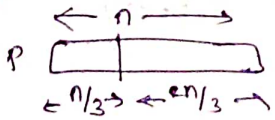
$g(n) \rightarrow$ time for SMALL, divide, combine

$\therefore g(n) > 0$

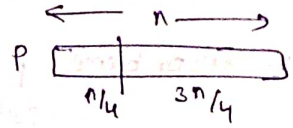
$\therefore a \geq 1;\ b > 1;\ g(n) > 0$

Variation of Dand C reccurence: (asymmetric)

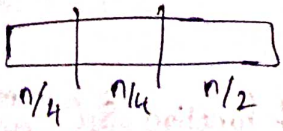
↳ unequal sized subproblems



$\Rightarrow T(n) = T(n/3) + T(2n/3) + g(n)$



$\Rightarrow T(n) = T(n/4) + T(3n/4) + g(n)$



$\Rightarrow T(n) = 2 \cdot T(n/4) + T(n/2) + g(n)$

~~Sum of sizes of subproblems~~

~~= size of original problem.~~

→ sometimes its possible that sum of size of subproblems not equal to size of original ~~program~~ problem.

Eg: $T(n) = T(n/2) + T(n/4) + g(n)$

Eg: $T(n) = T(n/2) + g(n)$

general form: $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + \cdots + g(n)$

where $0 < \alpha, \beta, \gamma, \cdots < 1$

Note:

$T(n) = 8T(n/2) + g(n)$

The above eq is valid even ~~thor~~ though size of subproblem is $n/2$ and no of subproblems are 8.

Reason: It could the case the a subproblem is solved multiple times

(or)

It is possible when the input data ~~struc~~ structure is non-linear

for example, for input being $n \times n$ matrix & dividing ~~in to~~ into 4 subproblems where each is $\frac{n}{2} \times \frac{n}{2}$ matrix.

i) Max-Min:

The problem is finding minimum and maximum element in an array of size 'n' elements.

Naive approach (Non-DC):

```
Procedure Non-DC (A, n, fmax, fmin)
{
    1. fmax ← fmin ← A[1];
    2. for i ← 2 to n
       {
          if (A[i] > fmax)
                fmax ← A[i]
          if (A[i] < fmin) // line 1
                fmin ← A[i];
       }
}
```

TC: O(n)

→ Here the metric based on which we measure the time is comparision. (Also we consider comparisions involving array element but we ignore comparision of index in for loop)

Here total no of comparisions = 2(n-1)

↳ (best, avg, worst cases)

→ Here by replacing line 1 with "else", for certain elements we can reduce no of comparisions.

best case ⟶ n-1 (increasing order)

worst case ⟶ 2(n-1) (decreasing order)

Avg. case → on an ~~avg~~ average assum that the first comparision fails for 1/2 of given elements.

It is given that comparision fails for 1/2 elements

⇒ for $n/2$ elements comparision fails

⇒ $n/2$ comparisions out of n-1 comparisions fails

In this case no of comparisions req $= n/2 * 2 = n$

⇒ for $n-1-n/2$ comparision are success

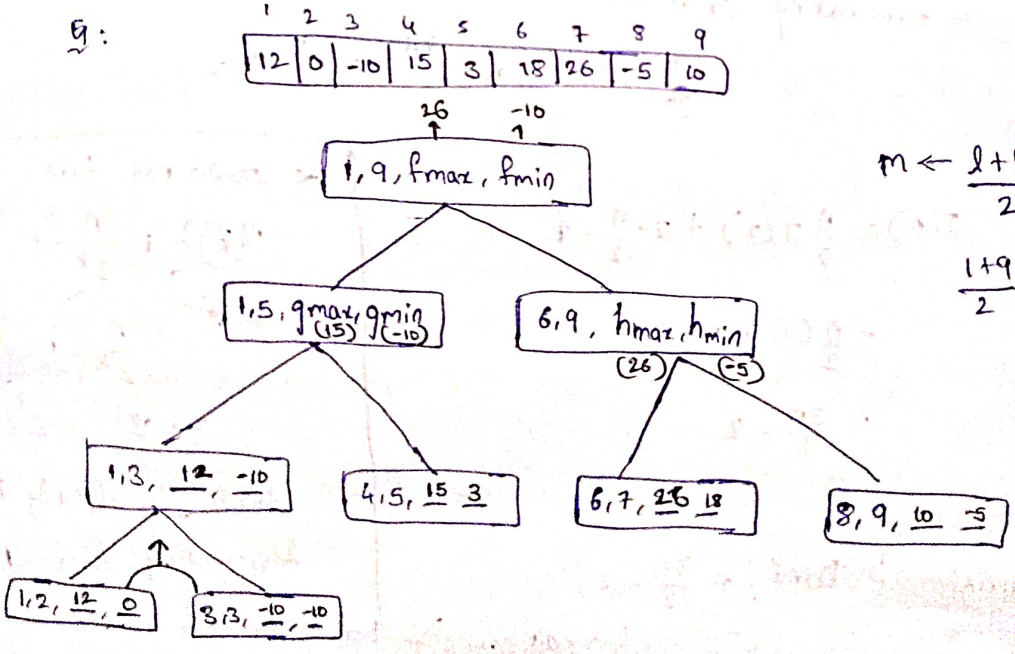
⇒ total comparisions → $n + n - 1 - n/2$

$$\frac{3n}{2} - 1$$

D and C approach:

If n=1 or n=2 then we say the problem is small. since in that case no of comparisions will be either 0 (or) 1 , respectively.

Eg:



$m \leftarrow \frac{l+h}{2}$

$\frac{1+9}{2} = 5$

∴ Every level of combine requires 2 comparisions

Let $T(n)$ represent the no of comparisions made in DandC Maximum(n)

$$T(n) = \begin{cases} 0 & , n=1 \\ 1 & , n=2 \\ 2T(n/2)+2 & , n>2 \end{cases}$$

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

$$= 2[2T(n/4)+2]+2$$

$$= 2^2 T(n/4) + 2^2 + 2$$

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

$$\vdots$$

$$= 2^k T(n/2^k) + 2^k + 2^{k-1} + \cdots + 2$$

$$= 2^k T(n/2^k) + 2\frac{(2^k - 1)}{2-1}$$

$$= 2^k T(n/2^k) + 2 \cdot 2^k - 2$$

~~$= nT(1) + 2n - 2$~~

terminating condition

$$\frac{n}{2^k} = 2 \Rightarrow n = 2^{k+1} \Rightarrow 2^k = n/2$$

$$T(n) = \frac{n}{2} T(2) + 2 \cdot \frac{n}{2} - 2$$

$$= \frac{n}{2}(1) + n - 2$$

$$= \frac{3n}{2} - 2$$

→ no of comparision in DandC = $\frac{3n}{2} - 2$ ↳ (All cases)

→ Time complexity : $O(n)$

we should not take $T(n/2^k) = 1$ if $\frac{n}{2^k} = 1$

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

$\Rightarrow \frac{n}{2^{k-1}} = 2$

which was already a terminating condition

In most of the cases it terminates with $n=2$.

Performance

I/P class
Inc order
Dec order
random order
TC
space-C

Space Complex

Non-DC :

DC :

Merge Sort

A: 3 | 10 | 2

Merging Proce

→ The below lists

Problem

them

Performance Comparision

I/p class	Non-DC	DC	
Inc order	$(n-1)$	$\frac{3n}{2}-2$	} naive approach is better
Dec order	$2(n-1)$	$\frac{3n}{2}-2$	} D and C is better here
random order	$\frac{3n}{2}-1$	$\frac{3n}{2}-2$	
TC	$O(n)$	$O(n)$	
Space.C	$O(1)$	$O(\log n)$	

Space Complexity

Non-DC: $O(1)$ --- memory for i

DC:

Here we need consider the space for stack.

The stack grows upto a depth of $\log n$

$\therefore O(\log n)$

Merge Sort

	1	2	3	4	5	6	7	8	9	10
A:	310	285	179	652	351	423	861	254	450	520

Merging Process (conquer)

→ The below is about 2-way merging i.e., we merge 2 sorted lists.

Problem stmt: Given two sorted lists L_1 & L_2, it is req to merge them into single sorted list.

Lists

$L_1: \langle 4, 5, 8, 12, 20 \rangle$ $L_2: \langle 3, 7, 11, 18, 25, 30 \rangle$

Assume $size(L_1) \leq size(L_2)$

Let combined list be L.

Let i, j be indices pointing to the start of the list.

$L_1: \langle 4, 5, 8, 12, 20 \rangle$, $L_2: \langle 3, 7, 11, 18, 25, 30 \rangle$

i j

$L_1[i] > L_2[j]$

$\therefore L[1] = L_2[j]$; j++

$L_1: \langle 4, 5, 8, 12, 20 \rangle$ $L_2: \langle 3, 7, 11, 18, 25, 30 \rangle$

i j

$L: \langle 3, \ldots \rangle$

$L_1[i] < L_2[j]$

$\therefore L[2] = L_1[i]$; i++

$L_1: \langle 4, 5, 8, 12, 20 \rangle$ $L_2: \langle 3, 7, 11, 18, 25, 30 \rangle$

i j

$L: \langle 3, 4, \ldots \rangle$

Continuing this process, we obtain

$L: \langle 3, 4, 5, 7, 8, 11 \rangle$

one of the two array will be completely traversed.

Now append the elements of array that is not completely traversed to the end of L.

Finally we obtain

$L: \langle 3, 4, 5, 7, 8, 11, 12, 20, 25, 30 \rangle$

~~Let n be size of L_1 & m be size of L_2~~

Analysis of Merge process:

Finding no of comparisions:

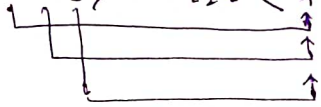
→ Here we consider comparision only b/w array elements

let n be size(L_1) & m be size of(L_2)

Case I: (best case)

Every element of L_1 < starting element of L_2

Eg: L_1: ⟨1,2,3⟩ L_2: ⟨4,5,6,7,8⟩



∴ no of comparision req = ~~size~~ 3 = size of L_1

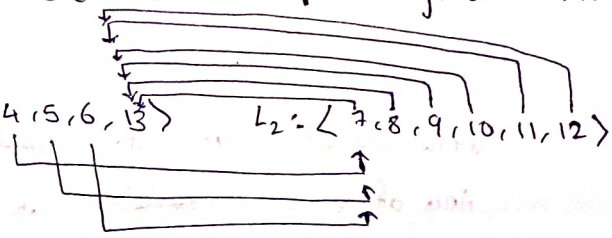
Case II: (worst case)

~~every L_1 except last element of~~

n-1 of list L_1 are less than 1st element of L_2

and nth element of L_1 is greater than all elements of L_2

L_1: ⟨4,5,6,13⟩ L_2: ⟨7,8,9,10,11,12⟩



∴ no of comparisions = (n-1) + m

= n+m-1

∴ ~~Time complexity~~

no of comparisions in best case = min(n, m)

no of comparisions in worst case = n+m-1

∴ TC: O(n+m)

Now we see how mergesort works

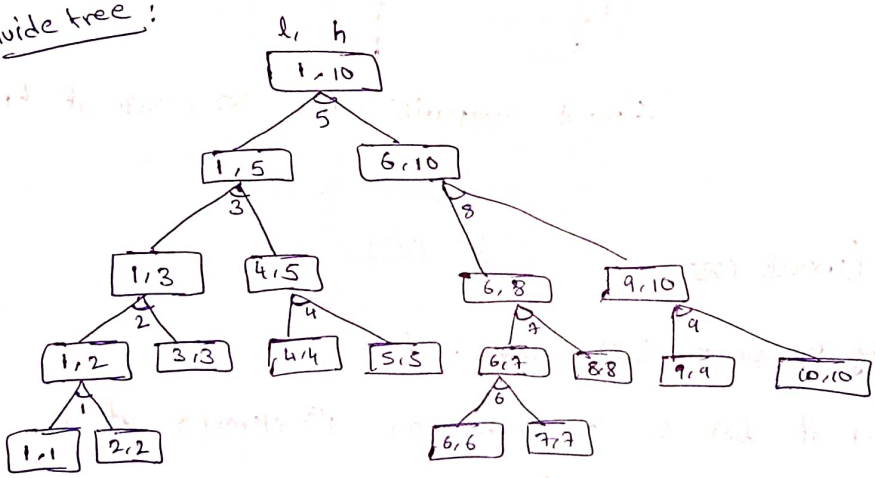
	1	2	3	4	5	6	7	8	9	10
A:	310	285	179	652	351	423	861	254	450	520

Merge sort uses a temporary array which is of the same size of A. Let this temporary array be B.

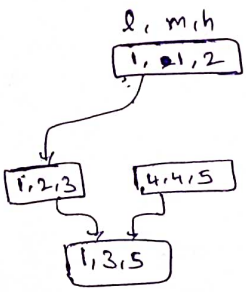
→ we say the problem is small when there is 1 element in the list.

Division: mid ← $\frac{l+h}{2}$

Divide tree:



Conquer tree:



while combining we store ~~store~~ the elements in B. And after every combine, we copy those combined elements from B to A.

B[1]=285 B[2]=310

Now copy them into A

A: <285, 310, 179, 652, ...>

Combine(1,2,3):

~~B[1]=179~~

<285, 310> <179>

B: <179, 285, 310, ...>

A: <179, 285, 310, ...>

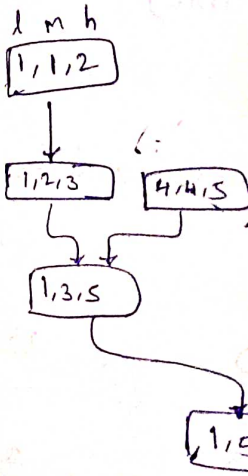
Combine(1,3,5):

<179, 285, ...

B: <179, 285, ...

A: <179, 285, ...

Combine tree:



Performance Analysis:

Let T(n) represent

$$T(n) = \begin{cases} c \\ 2 \end{cases}$$

I can also have $T(n) = 2T(n/2) + O(n)$

Combine (4,4,5):

B: ⟨17⟩

⟨652⟩ ⟨351⟩ ⇒ ⟨351,652⟩

B: ⟨179, 285, 310, 351, 652, ...⟩

A: ⟨179, 285, 310, 351, 652, ...⟩

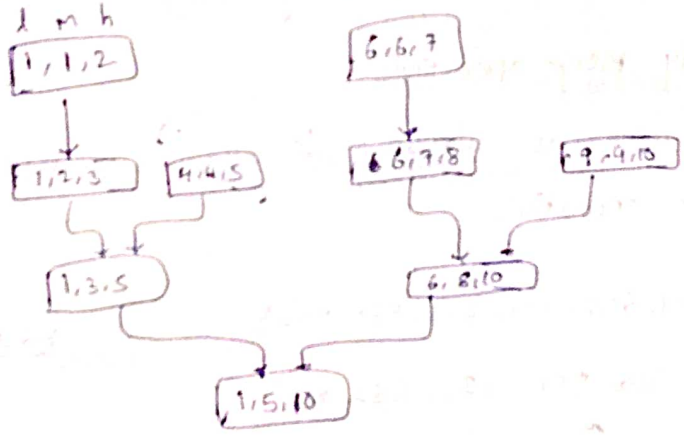
Combine (1,3,5):

⟨179, 285, 310⟩ ⟨351, 652⟩

B: ⟨179, 285, 310, 351, 652, ...⟩

A: ⟨179, 285, 310, 351, 652, ...⟩

Combine tree:



Performance Analysis:

Let T(n) represent time complexity of DandC-MS(n)

$$T(n) = \begin{cases} c, & n=1 \\ 2T(n/2) + bn, & n>1 \quad (b>0) \end{cases}$$

for the process of merging, checking if the problem is small etc.

bn in best case → $n/2$

worst case → $n/2 + n/2 - 1 = n-1$

We can also have $T(n) = 2T(n/2) + O(n)$

∴ time complexity of merge sort

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

space Complexity:

$S = C + Sp$

C is some constants and

Sp → sum of stack size + additional array B

→ $O(\log n) + O(n)$

∴ $S = O(n)$

∴ MergeSort is not space efficient.

MergeSort is not inplace sorting.

2-way / bottom-up MergeSort (variation of merge sort)

→ This sorting uses only merge operations.

→ Here we consider array is already divided and hence we perform only merge operations

A: ⟨310, 285, 179, 652, 351, 423, 861, 254⟩

no of merge operations

Pass3 → [179, 254, 285, 310, 351, 423, 652, 861] → $\frac{8}{2^3} = \frac{2}{2} = 1$

[179, 285, 310, 652] [254, 351, 423, 861]

Pass2 → → $\frac{8}{2^2} = \frac{4}{2} = 2$

Pass1 → [285, 310] [179, 652] [351, 423] [254, 861] → $8/2 = 4$

A: [310] [285] [179] [652] [351] [423] [861] [254]

Time Complexity : $O(n \log n)$

↳ : $\log n$ passes & every pass has comparisions less than n.

Time

Apos

Aprio

n = 64

6 min

let n be no

then

let

=)

k=

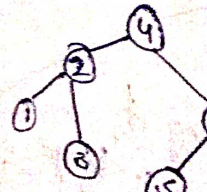
∴ n

Binary Search:

Binary search requires t

Consider a list of size

1 2 3



(H/35) Time complexity of mergesort = nlogn

n=64 → Apostenori time = 30s

→ Apriori time = $64\log_2 64 = 64 \times 6$ units

$\Rightarrow$ 64x6 units ⟶ 30s

1 units ⟶ $\frac{30}{64 \times 6}$ s

~~$\Rightarrow$ 360s ⟶ 64~~

$\Rightarrow$ 1s ⟶ $\frac{64 \times 4}{30}$

6 min $\Rightarrow$ 360 s ⟶ $\frac{64 \times 4 \times 360}{36} = 4608$

let n be no of elements sorted in 360s

then $n\log n = 4608$

let $n = 2^k$

$\Rightarrow 2^k \cdot k = 4608$

k=9 satisfies the equation

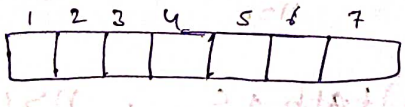
$\therefore n = 2^k = 512$ elements.

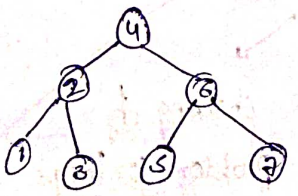
> no of comparisions in 2 way merge sort = $n\log n - n + 1$

Binary Search:

→ Binary search requires that the list is sorted

Consider a list of size 7



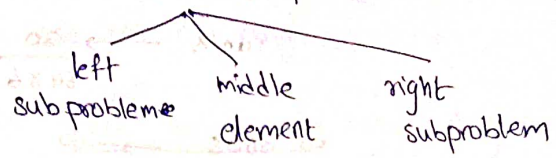


Search path ~~in this~~ using binary search corresponds to one of the path from root to leaf in the above tree.

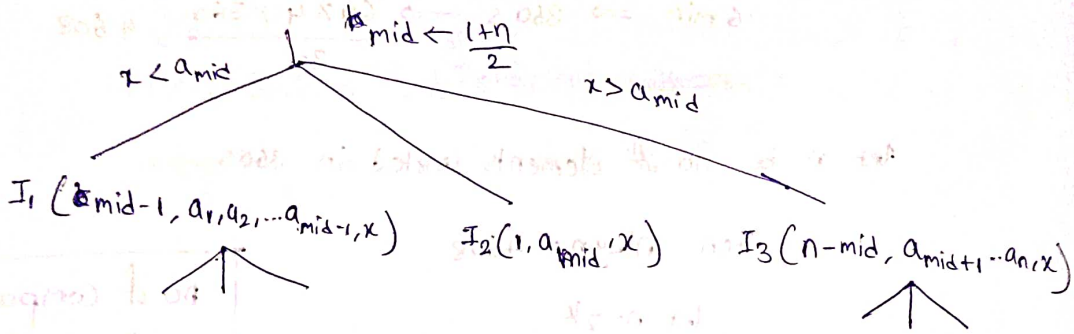
Dand C - BS(n, x)

→ we say a problem is small when there is only one element

→ In binary search, the ~~binary~~ when problem is large (n>1), the problem is divide into 3 subproblems



$I : \langle n, a_1, a_2, a_3 \ldots, a_n, x \rangle$



Out of these 3 subproblems, we solve only 2

i.e., I_2 and/or $(I_1$ (or) $I_3)$

→ Binary search ~~don~~ doesn't perform combine operation.

Performance Analysis:

time Complexity: Let T(n) represent time complexity of Dand C-BS(n)

$$T(n) = \begin{cases} c, & n=1 \\ T(n/2) + a + b + c, & n>1 \end{cases}$$

- $T(n/2)$: solving I_2
- a : checking if problem is large
- b : time for divide
- c : constant

(or)

$$T(n) = T(n/2) + k, \quad n>1, k>0$$

$\therefore T(n) = O(\log n)$

→ Best case TC of binary search = O(1)

* Space Complexity:

$S = C + Sp$

$= c_1 +$ size of stack used

$= c_1 + \log n$

$S = O(\log n)$

However for non-recursive implementation of binary search space Complexity, $S = O(1)$

Linear Search:

→ Best Case $TC = O(1)$

→ worst case & avg case $TC = O(n)$

→ The avg no of key comparisions involved in a successful sequential search with n-elements is $\frac{n+1}{2}$

i.e., $\frac{1+2+3+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$

Matrix Multiplication [square matrices]

→ $C = A \times B$

naive approach:

```
for i ← 1 to n
    for j ← 1 to n
    {
        C[i,j] = 0;
        for k ← 1 to n
    }       C[i,j] += A[i,k] * B[k,j];
```

(a) Calculate the total no of minimum & maximum no of element comparisions involved in 2-way / bottom up merge-sort with $n=2^k$ elements.

Sol:

Minimum Comparisions:

$$\left(\frac{2^k}{2}\times 1\right)+\left(\frac{2^{k-1}}{2}\times 2\right)+\left(\frac{2^{k-2}}{2}+2^2\right)+\cdots+\left(\frac{2^{k-i}}{2}\times 2^i\right)$$

($\frac{2^k}{2}$: no of lists; 1: min no of comparisions; $\frac{2^k}{2}$: no of merge operations)

At the termination, no of merge operations will be one.

$\Rightarrow$ no of lists $=2$

i.e., $2^{k-i}=2$

$\Rightarrow k-i=1$

$\Rightarrow i=k-1$

$$\Rightarrow \frac{2^k+2^k+\cdots+2^k \text{ (k times)}}{2} = \frac{k\cdot 2^k}{2} = k\cdot 2^{k-1}$$

$\therefore$ Min no of comparisions $= \frac{n\log n}{2}$

Maximum Comparisions:

$$\left(\frac{2^k}{2}\times 1\right)+\left[\frac{2^{k-1}}{2}\times(2^2-1)\right]+\left[\frac{2^{k-2}}{2}+(2^3-1)\right]+\cdots+\left[\frac{2^{k-i}}{2}\times(2^{i+1}-1)\right]$$

$$\Rightarrow\left[\frac{2^k}{2}\times 2^1-1\right]+\left[\frac{2^{k-1}}{2}\times(2^2-1)\right]+\cdots+\left[\frac{2^1}{2}\times(2^k-1)\right]$$

$2^{k-i}=2$

$k-i=1$

$i=k-1$

$\therefore$ Max no of

Consider

$$A=\left[\begin{array}{cc|c} 4 & 5 & 6 \\ 8 & 9 & 1 \\ \hline 3 & 4 & 1 \\ 6 & 7 & 2 \end{array}\right]$$

(A_{11}, A_{12}, A_{21}, A_{22})

Here

→ Here we say prob

T(n

8 multiplic

where each mu

is 1)

$$\frac{1}{2}\left[\underbrace{(2^{k+1}+2^{k+1}+\cdots+2^{k+1})}_{k \text{ times}} - (2^k + 2^{k-1} + \cdots + 2)\right]$$

$$= \frac{1}{2}\left[k \cdot 2^{k+1} - 2(2^k - 1)\right] = 2^k(k-1) + 1$$

$$= n\log n - n + 1$$

$\therefore$ Max no of comparision $= n\log n - n + 1$

05/10/20

Consider

$$A = \begin{bmatrix} 4 & 5 & 6 & 7 \\ 8 & 9 & 1 & 2 \\ 3 & 9 & 1 & 5 \\ 6 & 7 & 2 & 3 \end{bmatrix} \quad (A_{11}, A_{12}, A_{21}, A_{22}) \qquad B = \begin{bmatrix} 9 & 6 & 5 & 2 \\ 4 & 1 & 3 & 9 \\ 8 & 6 & 3 & 2 \\ 4 & 1 & 3 & 5 \end{bmatrix} \quad (B_{11}, B_{12}, B_{21}, B_{22})$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Here

$C_{11} = A_{11}B_{11} + A_{12} \cdot B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$

$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

$A, B, C \rightarrow n \times n$

$A_{ij}, B_{ij}, C_{ij} \rightarrow \frac{n}{2} \times \frac{n}{2}$

→ Here we say problem is small when the matrix is 2×2 or 1×1

$$T(n) = \begin{cases} c & , n <= 2 \quad (c > 0) \\ 8T(n/2) + bn^2 & , n > 2 \end{cases}$$

8 multiplications where each multiplication is blw two $n/2 \times n/2$ matrices

time for addition

$$T(n) = 8T(n/2) + bn^2$$

$$= 8\left[8T(n/2^2) + b\left(\frac{n}{2}\right)^2\right] + bn^2$$

$$= 8^2 T(n/2^2) + 8\frac{bn^2}{4} + bn^2$$

$$= \cancel{8^2 T(n/2^2)} + \cancel{8^2} + bn^2$$

$$\vdots$$

$$= \cancel{8^k T(n/2^k)} + \cancel{8^k - 1} bn^2$$

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow 8^k = n^3$$

$$= \cancel{n^3 T(1)} + n^3 - 1$$

$$= 8^2\left[8T(n/2^3) + b(n/2^2)^2\right] + 2bn^2 + bn^2$$

$$= 8^3 T(n/2^3) + 2^2 bn^2 + 2bn^2 + bn^2$$

$$= 8^3 T(n/2^3) + (2^3 - 1)bn^2$$

$$\vdots$$

$$= 8^k T(n/2^k) + (2^k - 1)bn^2$$

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow 8^k = n^3$$

$$= n^3 T(1) + (n-1)bn^2$$

$$= cn^3 + bn^3 - bn^2$$

$$\therefore T(n) = O(n^3)$$

Space Complexity $= O(\log n)$

∴ So this approach of matrix multiplication didn't reduce the time complexity.

Strassen's

→ The idea

reducing no

subtraction

let A, B, C

Aij, Bij

P, Q, R, S, T

P = (

Q = (

R = (

S = (

T = (

U = (

V = (

C_{11} = P

C_{12} =

C_{21} =

C_{22} =

Here to compute

Strassen's Matrix Multiplication

→ The idea behind strassen's matrix multiplication is reducing no of multiplication by increasing addition & subtraction operations.

let $A, B, C \longrightarrow n \times n$

$A_{ij}, B_{ij}, C_{ij} \longrightarrow \frac{n}{2} \times \frac{n}{2}$

$P, Q, R, S, T, U, V \longrightarrow \frac{n}{2} \times \frac{n}{2}$

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) \cdot B_{11}$$

$$R = (A_{11}) \cdot (B_{12} - B_{22})$$

$$S = (A_{22}) \cdot (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) \cdot (B_{22})$$

$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Here to compute $C_{11}, C_{12}, C_{21}, C_{22}$ we perform

7 multiplications & 18 add/sub

$$T(n) = \begin{cases} c, & n<=2 \ (c>0) \\ 7T(n/2) + bn^2, & n>2 \end{cases}$$

$T(n) = 7T(n/2) + bn^2$

$= 7\left(7T(n/2^2) + b(n/2)^2\right) + bn^2$ —

$= 7^2 T(n/2^2) + 7b(n/2)^2 + bn^2$

$= 7^2\left[7T(n/2^3) + b(n/2^2)^2\right] + 7b(n/2)^2 + bn^2$

$= 7^3 T(n/2^3) + 7^2(n/2^2)^2 + 7^1 b(n/2)^2 + bn^2$

$\vdots$

$= 7^k T(n/2^k) + 7^{k-1}(n/2^{k-1})^2 + 7^{k-2} b(n/2^{k-2})^2 + 7^0 b(n/2^0)^2$

$= 7^k T(n/2^k) + n^2\left(\left(\frac{7}{2^2}\right)^{k-1} + \left(\frac{7}{2^2}\right)^{k-2} + \cdots + 1\right)$

$= 7^k T(n/2^k) + n^2\left[\frac{(7/4)^k - 1}{\frac{7}{4} - 1}\right]$

$n/2^k = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

$\Rightarrow 7^k = 7^{\log_2 n}$

$= 7^{\log_2 n} T(1) + \frac{4}{3} n^2\left[(7/4)^{\log_2 n} - 1\right]$

$= n^{\log_2 7} T(1) + \frac{4}{3} n^2\left[n^{\log_2 7 - \log_2 4} - 1\right]$

$= n^{\log_2 7} \cdot c + \frac{4}{3} n^2\left[n^{\log_2 7 - 2} - 1\right]$

$= c \cdot n^{\log_2 7} + \frac{4}{3} n^{\log_2 7} - 1$

$\therefore T(n) = O(n^{\log_2 7})$

$\Rightarrow T(n) = O(n^{2.81})$

Space Complexity, $S = O(n^2)$

↳ space for matrices P, Q, R, S, T, U, V

Quick Sort [Partition - Exchange Sort]

→ Quick sort is based on the principle of partitioning (pivoting)

	1	2	3	4	5	6	7	8	9
A:	65	70	75	80	85	60	55	50	45

Partitioning:

(i) Select a pivot (partitioning element)

(ii) Pivot the partitioning element at its correct place (fix)

(iii) All elements that are less than pivot has to be on left of the pivot.

(iv) All elements that are greater than pivot has to on right of the pivot.

```
int PARTITION (A, l, h)
{
    i ← l;  j ← h; // h is n+1
    v ← A[l]; // v contains partitioning element.
    loop
    {
        loop i ← i+1 until (A[i] ≥ v);
        loop j ← j-1 until (A[j] ≤ v);
```

```
if (i<j)
    SWAP(A[i], A[j]);
else
    break;
}until(false);
A[1] = A[j]; A[j] = v; return j;
} until(false);
```

↳ position of pivot

(1) (h)

1	2	3	4	5	6	7	8	9	10
65	70	75	80	85	60	55	50	45	∞
↑ i									↑ j

↳ Having infinity here helps when pivot is the largest element. Sometimes, if we don't use ∞ the program may go to infinite loop

$v \leftarrow A[1]$

$v \leftarrow 65$

$i \leftarrow i+1 \Rightarrow i=2$

$A[i] \geq v \Rightarrow A[2] \geq 65$

↓

true ∴ exit loop

$j \leftarrow j-1 \Rightarrow j=9$

$A[j] \leq v \Rightarrow A[9] \leq 65$

↓

true ∴ exit loop

if $(i<j)$ .. true

swap(A[i], A[j])

∴ swap(A[2], A[9])

1	2	3	4	5	6	7	8	9
65	45	75	80	85	60	55	50	70
	↑ i						↑ j	

$i \leftarrow i+1 \Rightarrow i=3$

$A[3] \geq 65$ ✓

$j \leftarrow j-1 \Rightarrow j=8$

$A[8] \leq 65$ ✓

if $(i<j)$ .. true

∴ SWAP(A[8], A[8])

next iteration

A: [65

A: [65 45 50

next iteration:

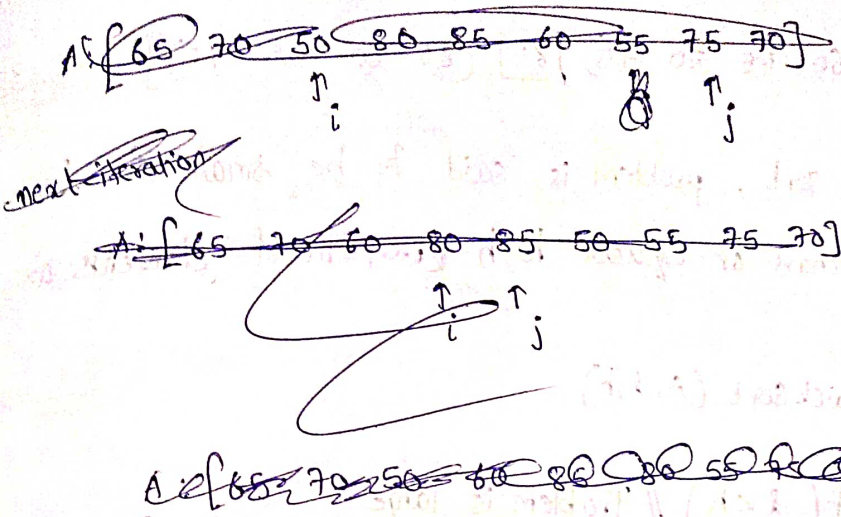
A: [65 45

next iteration:

A: [65

Now we need

This



- pivot

> Having infinity here helps when pivot is the largest element. Sometimes, if we don't use ∞ the program may go to infinite loop

A: [65 45 50 80 85 60 55 75 70]
i j

next iteration:

A: [65 45 50 55 85 60 80 75 70]
i j

next iteration:

A: [65 45 50 55 60 85 80 75 70]
i j

i 5 j 6

$i \leftarrow i+1 \Rightarrow i$ 6

$A[i] \geq 65$ ✓

$j \leftarrow j-1$ · j 5

$A[j] \leq 65$

$60 \leq 65$ ✓

if $(i<j)$ ... false

Now we need to put partitioning element in its position.

This position is present in j.

A: [(60 45 50 55) 65 (85 80 75 70)]

→ In quick sort, problem is said to be small if low is greater than or equal to h (i.e., no of elements are 1)

```
void QuickSort (A, l, h)
{
    if( l < h) // Problem is large
    {
        m ← PARTITION (A, l, h);
        QuickSort (A, l, m-1);
        QuickSort (A, m+1, h);
    }
}
```

Analysis:

TC of PARTITION:

→ In both loops which are inside loop-until loop, we perform a total 'n+1' comparision blw array elements and v (pivot).

∴ TC of partition = O(n)

TC of quicksort:

A: <a1, a2, a3, ... an>

Partitioning : O(n)

<(...) a1 (...)> <(...) a1> <a1 (...)>

(I) (II) (III)

Case I:

$$T(n) = \begin{cases} c, & n=1 \\ 2T(n/2)+O(n), & n>1 \end{cases}$$

$$\Rightarrow T(n) = O(n \log n)$$

Case II & III:

$$T(n) = \begin{cases} c, & n=1 \\ T(n-1)+n, & n>1 \end{cases}$$

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

Note:

→ avg case TC of Quick sort = $O(n \log n)$

∴ for QS,

best case TC = avg case TC = $O(n \log n)$

~~worst c~~ worst case TC = $O(n^2)$

→ worst case of TC happens when elements are already sorted (i.e., either increasing or decreasing)

Space Complexity:

best case space complexity → $O(\log n)$

worst case space complexity → $O(n)$

↳ sorted list

H/76

Partitioning has to start choosing pivot element from one of the end elements of array.

So we swap central element with any other end elements of array & finally the worst case TC is $O(n^2)$

H/77

time to find median $O(n)$

time to partition w.r.to median $O(n)$

$\therefore$ total time for partitioning $O(n)$

Here every time pivot goes to middle of list and hence forms best case.

$\therefore O(n \log n)$

H/78

$$T(n) = O(n) + O(n) + T(n/4) + T(3n/4)$$

$$\Rightarrow T(n) = O(n \log n)$$

Note:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + O(n) \quad , \quad 0 < \alpha < 1$$

$$\Rightarrow T(n) = O(n \log n)$$

H/79

$$T(n) \leq T(n/5) + T(4n/5) + n$$

H/85

H/86

H/89 Both are

Master Theor

$T(n) =$

Case I:

If $f(n$

Case II:

If $f(n)$

a)

b)

c)

(H/85) both are worst case

the question talks abt realistic time. (∵ the word program is given)

$\therefore t_1 < t_2$

(H/86) $t_1 \rightarrow$ worst case

$t_2 \rightarrow$ best case

$\therefore t_1 > t_2$

(H/87) Both are worst case & they are symmetric

$\therefore c_1 = c_2$

Master Theorem for solving Dand C Recurrences:

$$T(n) = \begin{cases} aT(n/b) + f(n), & n > d \quad a \geq 1;\ b > 1,\ f(n) \text{ is +ve} \\ c, & n \leq d \end{cases}$$

Case I:

If $f(n)$ is $O(n^{\log_b a - \epsilon})$, $\epsilon > 0$ then

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

Case II:

If $f(n)$ is $\theta(n^{\log_b a} \cdot \log^k n)$ for some 'k'

a) $k \geq 0$ then $T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$

b) $k = -1$ then $T(n) = \theta(n^{\log_b a} \cdot \log\log n)$

c) $k < -1$ then $T(n) = \theta(n^{\log_b a})$

Case III :

If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and

$a \cdot f(n/b) \leq \delta \cdot f(n)$ whenever $\delta < 1$ then

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

Examples on Master theorem :

→ $T(n) = 4T(n/2) + n$

$a = 4$; $b = 2$; $f(n) = n$;

$\log_2 4 = 2$

Case (i) : $n = O(n^{2-\epsilon})$

for $\epsilon > 0$

say $\epsilon = 0.5$

the equation satisfies

$\therefore \Rightarrow T(n) = O(n^{\log_b a})$

$= O(n^2)$

→ $T(n) = 2T(n/2) + n\log n$

$a = 2$ $b = 2$ $f(n) = n\log n$

$\log_a b = 1$

Case (i) : $n\log n = O(n^{\log_b a - \epsilon})$

$n\log n = O(n^{1-\epsilon})$

$\because$ for $\epsilon > 0$

the above equation is false

Case(ii) :

$$n \log n = \theta(n \log^k n)$$

for $k=1$, the above equation is satisfied.

also $k \geq 0$

$$\therefore T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$$

$$= \theta(n^1 \cdot \log^2 n)$$

$$= \theta(n \cdot \log n \cdot \log n)$$

→ $T(n) = T(n/3) + n$

$a=1$, $b=3$; $f(n)=n$

$$\log_b a = \log_3 1 = 0$$

Case I : $n = O(n^{0-\epsilon})$

false

Case II : ~~$n = \theta(\log n \log^k n)$~~

$$n = \theta(n^{\log_b a} \cdot \log^k n)$$

$$n = \theta(n^0 \cdot \log^k n)$$

$$n = \theta((\log n)^k)$$ ∝

Case III : $n = \Omega(n^{\log_b a + \epsilon})$

$$n = \Omega(n^{\epsilon}), \quad \epsilon > 0$$

$\forall$ $0 < \epsilon \leq 1$

above eqn satisfies

Now $af(n/b) \leq \delta f(n)$, $\delta < 1$

$1 \cdot n/3 \leq \delta \cdot n$

$\frac{n}{3} \leq \delta n$ ✓

for ~~$\delta \geq \frac{1}{3}$~~ δ

for $\frac{1}{3} \leq \delta < 1$ the above inequality holds

$\Rightarrow T(n) = \theta(f(n))$

$\Rightarrow T(n) = \theta(n)$

→ $T(n) = 9T(n/3) + n^{2.5}$

$a = 9 \quad b = 3 \quad f(n) = n^{2.5}$

$\log_b a = 2$

Case I:

$f(n) = n^{2.5} = \theta(n^{\log_b a - \epsilon})$

$n^{2.5} = \theta(n^{2-\epsilon})$ ✗

Case II:

$f(n) = n^{2.5} = \theta(n^{\log_b a} \log^k n)$

$n^{2.5} = \theta(n^2 \log^k n)$

for no k the above eqn will be satisfied $\left(\because \log^k n = O(n^a) \text{ as } a > 0\right)$

Case III:

$n^{2.5} = \Omega(n^{\log_b a + \epsilon})$

$= \Omega(n^{2+\epsilon})$ ✓

now consider

$$a \cdot f(n/b) \leq \delta \cdot f(n)$$

$$9 f(n/3) \leq \delta\, n^{2.5}$$

$$9 \cdot (n/3)^{2.5} \leq \delta\, n^{2.5}$$

$$\frac{n^{2.5}}{\sqrt{3}} \leq \delta\, n^{2.5}$$

$$\Rightarrow \frac{1}{\sqrt{3}} \leq \delta$$

$$\Rightarrow \delta \geq \frac{1}{1.732}$$

$$\delta \geq 0.\_$$

∴ The above eqn satisfies for $\delta < 1$

$$\therefore T(n) = \theta(f(n))$$

$$= \theta(n^{2.5})$$

→ MaxMin :

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

Case (i) : $2 = O(n^{\log_2 2 - \epsilon})$.

$$2 = O(n^{1-\epsilon})$$

true for $\epsilon = 1$

$$\therefore T(n) = O(n^{\log_a b}) = O(n)$$

→ MergeSort :

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

Case (i) : $bn = O(n^{1-\epsilon})$ ✗

Case (ii) : $bn = \theta(n^1 \log^k n)$

true for $k = 0$

k≥0

$\therefore T(n) = \theta(n^{\log_a b} \cdot \log^{k+1} n)$

$T(n) = \theta(n \log n)$

→ Matrix multiplication:

$T(n) = 8T(n/2) + n^2$

$n^2 = O(n^{3-\epsilon})$ ✓

$\therefore T(n) = \theta(n^{\log_2 8}) = \theta(n^3)$

• Strassen's Matrix mul:

$T(n) = 7T(n/2) + bn^2$

$bn^2 = O(n^{2.81-\epsilon})$ ✓

$\therefore T(n) = \theta(n^{\log_2 7}) = \theta(n^{2.81})$

→ Binary Search:

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

(i) $c = O(n^{\log_2 1 - \epsilon})$

$c = O(n^{-\epsilon})$ ✗

(ii) $c = \theta(n^0 \log^k n)$

for $k = 0$ eqn is satisfied

now $k \geq 0$

$\Rightarrow T(n) = \theta(n^{\log_b a} \log^{k+1} n)$

$= \theta(n^0 \log n)$

$\therefore T(n) = \theta(\log n)$

* *

$T(n) = 2 \cdot T(\sqrt{n}) + \log n$

Here we cannot directly apply master theorem.

so we perform appropriate transformations.

Let $n = 2^k$

$\Rightarrow T(2^k) = 2 \cdot T(2^{k/2}) + k$

Let $T(2^k) \rightarrow S(k)$

$S(k) = 2 \cdot S(k/2) + k$

while transform only T has to be changed to S. while remaining has to be kept same.

$\Rightarrow S(k) = \Theta(k \log k)$

$\Rightarrow T(n) = \Theta(\log n \cdot \log\log n)$

* *

$T(n) = 2T(n/2) + \frac{n}{\log n}$

Case (i) : $\frac{n}{\log n} = O(n^{1-\epsilon})$ ✗

Case (ii) $\frac{n}{\log n} = \Theta(n \cdot \log^k n)$

$\Rightarrow k = -1$

$\therefore T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$

$\Rightarrow T(n) = \Theta(n \cdot \log\log n)$

$T(n) = T(n/2) + T(n/4) + n$ → Recursion tree method gives $O(n)$

$\because T(n/4) < T(n/2)$

$\Rightarrow T(n) < 2T(n/2) + n$

$\Rightarrow T(n) = O(n \log n)$

Long Integer Multiplication:

We represent long integers using 1-D array

→ In general time for addition or multiplication is $O(1)$.

However representing with array ; time complexity for addition or multiplication will not be $\Theta(1)$.

→ TC for add / sub of two long integers = $O(n)$

→ TC for mul of two long integers = $O(n^2)$ (naive approach)

$\because$ for every digit of one number we multiply it with the other number.

D&C approach:

Let u, v be n digit numbers

$m \leftarrow \lfloor \frac{n}{2} \rfloor$

. now we divide u as follows into w & x ($2(n/2)$ integers)

$u = w * 10^m + x$

where $w = u / 10^m$

$x = u \% 10^m$

silly we divide u into 2 integers

$v = y * 10^m + z$

$y = v / 10^m$

$z = v \% 10^m$ w, x, y, z are $n/2$ sized integers

→ now $uv = (w \times 10^m + x)(y \times 10^m + z)$

$= (wy \times 10^{2m}) + (wz + xy) 10^m + xz$

4 multiplications of size $n/2$

$$T(n) = \begin{cases} c & , n = 1 \\ 4T(n/2) + bn & , n > 1 \end{cases}$$

↓ additions

$T(n) = 4T(n/2) + bn$

Case (i): $bn = O(n^{2-\epsilon})$ ✓

$\therefore T(n) = \Theta(n^2)$

∴ time complexity → $O(n^2)$

~~space complexity~~

Anatolii Karatsuba's Optimization: for LTM:

→ The ideology behind the optimization is reducing no of multiplication & operations

$$uv = (wy \times 10^{2m}) + (wz + xy) 10^m + xz$$

Let t be an $n/2$ sized integer

$$\therefore t = (w+x)(y+z)$$

$$t = wy + \underline{wz + xy} + xz$$

$$\Rightarrow wz + xy = t - (wy + xz)$$

from this we define

$$P_1 = wy$$

$$P_2 = xz$$

$$P_3 = (w+x)(y+z)$$

$$\Rightarrow wz + xy = P_3 - (P_1 + P_2)$$

Now we write

$$uv = (P_1 \times 10^{2m}) + (P_3 - (P_1 + P_2)) \times 10^m + P_2$$

This approach requires only 3 multiplication. However no of add/sub operations are increased.

$$\therefore T(n) = 3T(n/2) + O(n), \quad n > 1$$

$$= C \qquad\qquad n = 1$$

$$\Rightarrow T(n) = (n^{\log_2^3}) = O(n^{1.58})$$

Toom & Cook optimization for LIM:

→ This approach uses multi-way split.

3-way split:

$m \leftarrow \lfloor n/3 \rfloor$

$u = a \times 10^{2m} + b \times 10^{m} + c$

$v = d \times 10^{2m} + e \times 10^{m} + f$

$uv = (a \times 10^{2m} + b \times 10^{m} + c)(d \times 10^{2m} + e \times 10^{m} + f)$

⇒ no of multiplication req $= 3^2 = 9$

each on $n/3$ sized integers

$$\Rightarrow T(n) = \begin{cases} 9T(n/3) + O(n) & n > 1 \\ c & n = 1 \end{cases}$$

$\Rightarrow T(n) = O(n^2)$

using anatolli's optimization we get

$T(n) = 8T(n/3) + O(n)$

In anatolli's optimization value of 'a' is always reduced by 1.

$\Rightarrow T(n) = O(n^{1.89})$

Ideology behind Toom & Cook optimization

Let x be no of subproblems solved

$\Rightarrow T(n) = xT(n/3) + O(n)$

Now target is to have $T(n) < O(n^{1.58})$

Here $T(n) = O(n^{\log_3 x}) < O(n^{1.58})$

$\Rightarrow \log_3 x < 1.58$

$\Rightarrow x = 5$

Now the conclusion is that if we solve only 5 subproblems then we can have TC which is less than $O(n^{1.58})$.

→ Toom & Cook were successful in devising equation in which we require to perform only 5 multiplication operations.

$\Rightarrow T(n) = 5T(n/3) + O(n)$

$\Rightarrow T(n) = O(n^{\log_b a})$

$\Rightarrow T(n) = O(n^{1.46})$

4-way split :

normal D&C approach

$T(n) = 16\,T(n/4) + O(n^2)$

Anatolli's optimization

$T(n) = 15\,T(n/4) + O(n)$

$\Rightarrow T(n) = O(n^{1.9})$

Toom & cook's Optimization:

$\Rightarrow T(n) = xT(n/4) + O(n) = O(n^{\log_4 x}) < O(n^{1.46})$

$\Rightarrow x = 7$

$\Rightarrow T(n) =$

$\Rightarrow T(n) = 7T(n/4) + O(n)$

$\Rightarrow T(n) = O(n^{1.40})$

k-way split:

DC: $T(n) = k^2 T(n/k) + O(n) : O(n^2)$

AK: $T(n) = (k^2-1) T(n/k) + O(n) : O(n^{\log_k (k^2-1)})$

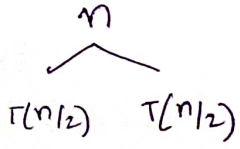
TFC: $T(n) = (2k-1) T(n/k) + O(n) : O(n^{\log_k (2k-1)})$

07/10/20

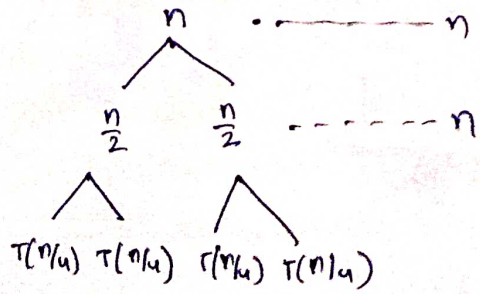
Solving Dand C Recurrences using Recursion tree method:

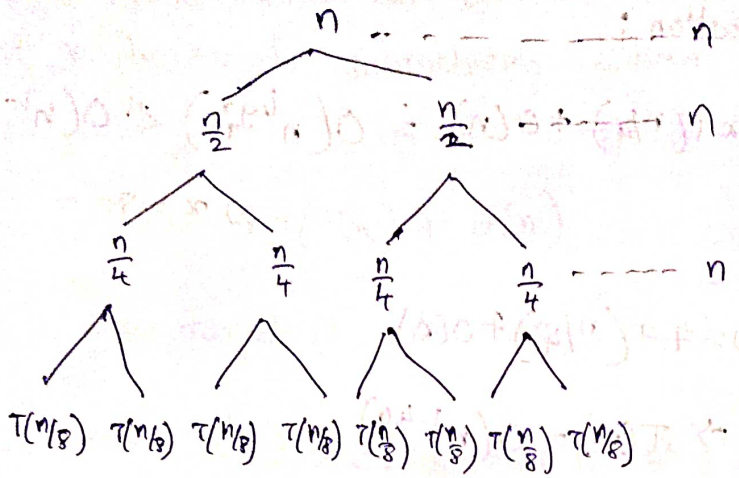
→ Recursion tree can be used to solve ~~any~~ assymetric recurrences.

Eg: $T(n) = 2T(n/2) + n$



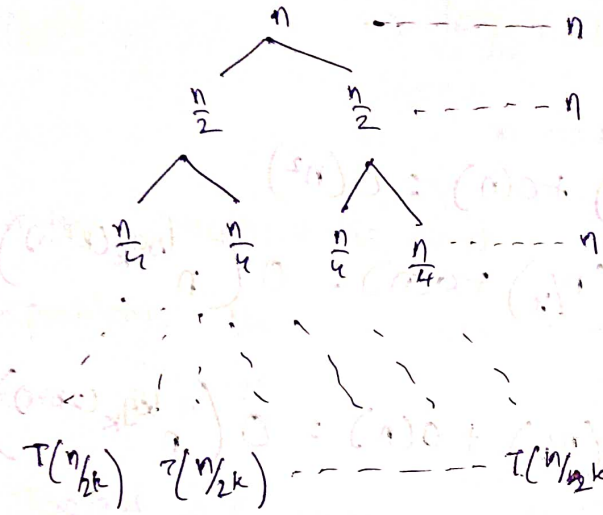
$\Rightarrow T(n/2) = 2T(n/4 + n/2)$





At every level it is n.

So we continue to max possible depth



This last level containing 2^k subproblems.

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$$

no of levels are $\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2} \ldots \frac{n}{2^{k-1}}, \frac{n}{2^k}$

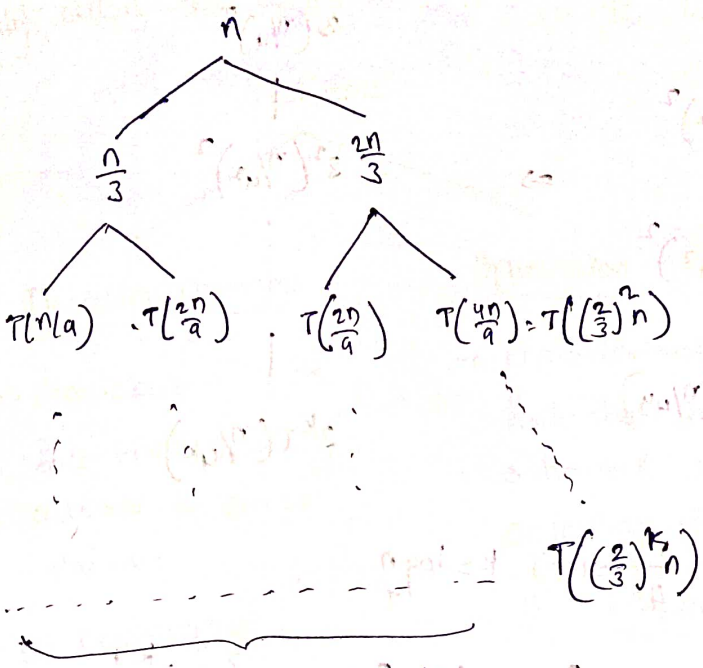
$\therefore$ k+1 level.

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

$$= n(\log n + 1)$$

$$\Rightarrow T(n) = O(n \log n)$$

Eg: $T(n) = T(n/3) + T(2n/3) + n$; $T(1) = 1$



→ This node goes to maximum depth

→ 2^k subproblem are present in this level.

we assume all nodes have reach this depth and get an approximate answer

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

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

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

total of $k+1$ levels.

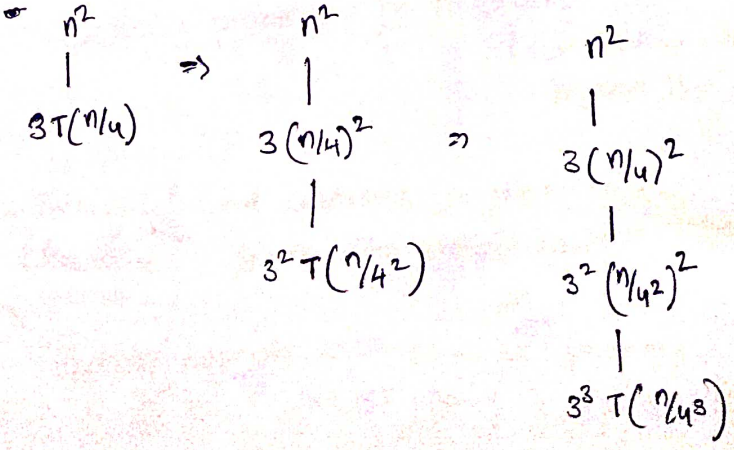
$\therefore T(n) = n(k+1)$

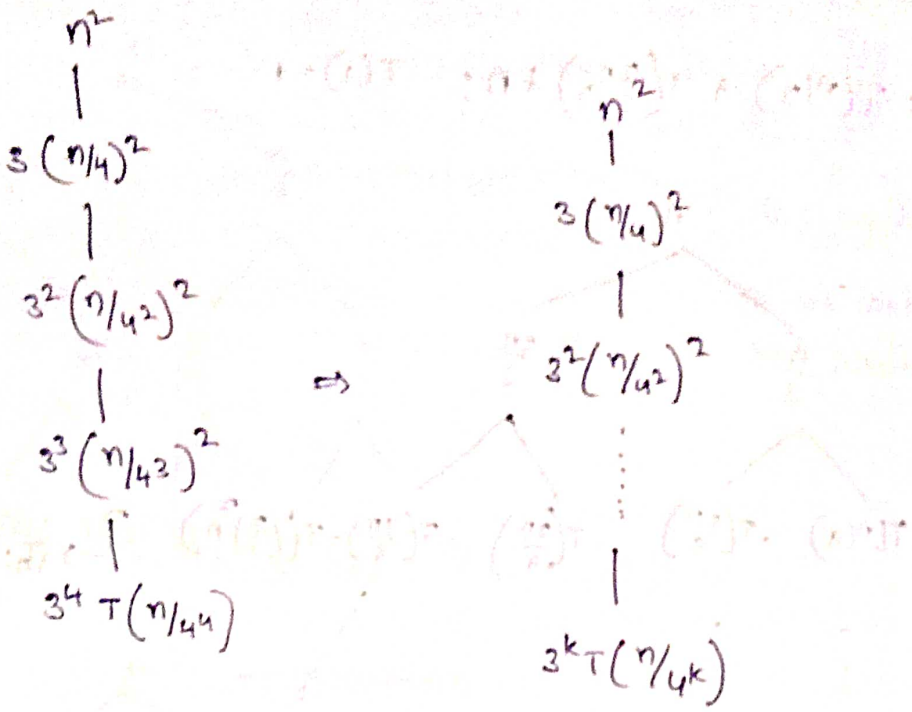
$= n(\log_{3/2} n + 1)$

$\therefore T(n) = O(n \log_{3/2} n) = O(n \log n)$

Eg: $T(n) = 3T(n/4) + n^2$

$T(n/4) = 3T(n/4^2) + (n/4)^2$





$$\frac{n}{4^k} = 1 \Rightarrow k = \log_4 n$$

$$\therefore n^2 + \frac{3}{4^2}n^2 + \left(\frac{3}{4^2}\right)^2 n^2 + \left(\frac{3}{4^2}\right)^3 n^2 + \cdots + \left(\frac{3}{4^2}\right)^{k-1} n^2 + 3^k$$

$$= n^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{k-1}\right) + 3^k$$

$$= n^2\left(\frac{1}{1-3/16}\right) + 3^{\log_4 n}$$

$$= n^2\left(\frac{16}{13}\right) + n^{\log_4 3}$$

$$\Rightarrow T(n) = O(n^2)$$

$\rightarrow$ $T(n) = T(n/2) + T(n/4) + n$

Ans: $O(n)$ 2 → solve

Greedy Method:

→ generally used for optimization problem.

Problems

Decision Problems

→ Characterized by the fact that its result/ outcome is either yes/no.

Eg: Divisibility, prime number etc.

Optimization → by optimization, it never means optimizing time complexity.

→ characterized by the fact that it tries to optimize a given criterian of the problem.

Eg: knapsack, Job sequencing, etc.

Eg: Graph coloring ⟶ optimization problem

Travelling salesman ⟶ optimization problem

shortest Path ⟶ optimization problem

n-Queens ⟶ Decision (whether queens can be placed in non-attacking position or not)

Sorting ⟶ Decision

Banker's algo ⟶ Decision

CPU scheduling ⟶ Optimization

Memory allocation ⟶ Optimization.

Terminology:

i) Problem Definition : Defining problem Eg: n-queens
Here we define what the problem is.

ii) Constraints
- Implicit (local constraints to the problem. (Feasibility Criteria) Eg: no two queens should in attacking positions)
- Explicit (depends on instance of problem. For example solving 4-queens, so array size is 4)

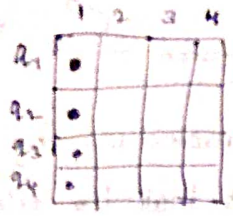
2.

(iii) Solution space:

All possible ways of organizing inputs but satisfying explicit constraints.

ie., It may or may not meet implicit constraint.

Eg:



⇒ This part of soln space

ie., $\langle 1,2,3,4 \rangle$ is part of soln space.

However $\langle 1,3,5,2 \rangle$ is not a part of soln space because 4-queens problems explicit constraints is that queens could be place in locations from 1 to 4.

∴ Having 5 doesn't meet explicit constraint.

(iv) So for n-queens problems size of soln space = n!

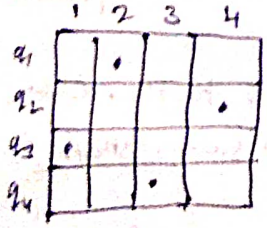
If problem is constructing binary search tree with 'n' nodes, then size of soln space = $\frac{1}{n+1} {}^{2n}C_n$

→ Working with entire soln space is called brute fore method.

(v) Feasible solution:

The solutions in solutions space, that satisfy the implicit constraints of the problem is/are called feasible solutions.

Eg:



4-queens has 2 feasible solns

(v) Objective Function:

Refers to min/max of given criterian of the problem.

Problems may have objective function or may not have.
↳ n-queens problem doesn't have objective function.

Eg: Consider shortest path problem

→ soln space has all paths blw any two vertices

→ feasible soln has only those paths that connect source and destination vertex.

→ Here objective function is to minimize the cost of path.

(vi) Optimal Solution:

It is the feasible solution that satisfies objective function.

→ optimal soln is always unique.

Note:

→ If a problem doesn't have objective function then we find only feasible soln.

→ Decision problems doesn't have objective function, hence for decision problems we find only feasible solutions.

→ Greedy method is an algorithm design strategy, used for solving problems, whose outcomes are seen as a result of making a sequence of decisions.

→ Greedy method makes these decision in a stepwise manner by applying the principle of local optimality (greedy choice property)

→ Principle of local optimality says that whatever the initial state & available options, greedily & select that option that satisfies the objectivity function (this give feasible soln)

Eg: SJF algorithm, Shortest seek time First

i) Knapsack Problem:

→ given a knapsack of capacity: M

→ given n-objects (O_i) & each object is associated with weight (w_i) and profit (P_i)

→ Maximizing the profit such that the total weight that put into knapsack should not exceed its capacity (M).

→ For every object we make decision to find x_i (fraction of O_i put into knapsack)

$$0 \le x_i \le 1$$

for x_i the weight put into knapsack $= w_i x_i$

the profit object $= P_i x_i$

∴ we need to maximize $\sum_{i=1}^{n} P_i \cdot x_i$ such that

$$\sum_{i=1}^{n} w_i \cdot x_i \le M$$

→ x_i can be real value

This type of knapsack problem is called ~~real-knap~~ real knapsack / fractional knapsack / greedy knapsack

→ Size of soln space of real knapsack problem infinite.

→ for 0/1 knapsack the size of soln space is 2^n.

Consider below instance of knapsack problem.

$n=3;\ m=20;\quad \langle P_1, P_2, P_3 \rangle = \langle 25, 24, 15 \rangle$

$\langle w_1, w_2, w_3 \rangle = \langle 18, 15, 10 \rangle$

$\langle x_1, x_2, x_3 \rangle = ?$

> Explicit constraint of knapsack problem is
>
> $\sum_{i=1}^{n} w_i > M$

(i) Greedy about profit:

P_1 has highest profit

$\Rightarrow x_1 = 1$

remaining capacity = $20 - 18 = 2$

P_2 has next highest profit

$\Rightarrow x_2 = \frac{2}{15}$

Here profit $P = P_1x_1 + P_2x_2 + P_3x_3$

$= 25(1) + 24\left(\frac{2}{15}\right) = 28.3$

(ii) Greedy about weight:

w_3 is least.

$\therefore x_3 = 1$

remaining capacity = $20 - 10 = 10$

w_2 has next least weight

$\Rightarrow x_2 = \frac{10}{15} = \frac{2}{3}$

$\therefore$ profit = $P_1x_1 + P_2x_2 + P_3x_3$

$= 0 + \frac{2}{3}(24) + 1(15)$

$= 31$

(iii) Greedy about object that has highest profit per weight ratio:

$\frac{P_1}{\omega_1} = \frac{25}{18} = 1.3 \Rightarrow x_1 = 0$

$\frac{P_2}{\omega_2} = \frac{24}{15} = 1.6 \Rightarrow x_2 = 1$

$\frac{P_3}{\omega_3} = \frac{15}{10} = 1.5 \Rightarrow x_3 = \frac{5}{10}$

Profit $P = P_1x_1 + P_2x_2 + P_3x_3$

$= 0 + 24(1) + 15\left(\frac{5}{10}\right)$

$= 24 + 7.5 = 31.5$

→ 3rd approach is optimal. ✓

→ So, arrange the objects in decreasing order of P/w ratio & keep on including the objects, until a kth object cannot be placed and for this object we take proportionate fraction.

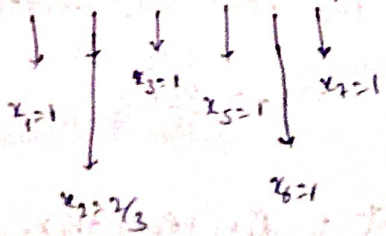
Time complexity after sorting → O(n)

Time complexity including sorting → O(n log n)

Eg: n=7 m=15; $\langle P_1 - P_7 \rangle = \langle 10, 5, 15, 7, 6, 18, 3 \rangle$

$\langle \omega_1 - \omega_7 \rangle = \langle 2, 3, 5, 7, 1, 4, 1 \rangle$

$\left\langle \frac{P_1}{\omega_1} - \frac{P_7}{\omega_7} \right\rangle = \langle 5, 1.66, 3, 1, 6, 4.5, 3 \rangle$



$P = 10 + \frac{2}{3}(5) + 15 + 0 + 6 + 18 + 3$

$= 55.33$