Gunjan ①

Analysis and Design of Algorithm.

Algorithm Design and Analysis.

What is an algorithm?

- An algo. is a step by step set of unambiguous instruct which is used to solve a problem
  ↳ computational problem.
  [which can be implemented as a program in computer.]

→ An algo. is named from the ninth centuary mathematici al-Khowarizmi :- is simply a set of rules used to perform some calculations, either by hand or more usually by machine.

→ Algorithm is any well-defined computational procedure that takes some values, or set of values as input and produce some values or set of values as output.

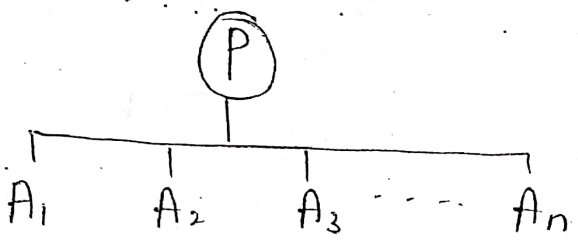
An algorithm is thus a sequence of computational steps that transforms the input into the output.

All the algorithms must satisfy the following criteria:-

1) Input : Zero or more quantities are externally supplied
2) Output : At least one quantity is produced.
3) Definiteness : Each instruction is clear and unambiguous.
4) Finiteness : The algo terminates after a finite no. of steps.
5) Effectiveness : Every instruction must be very basic, so that it can be carried out effectively.

In order to solve any problem in computer science, generally we write programs and before writing programs we should write a blueprint or an informal description of the sol^n which is also called as algorithms.

Let us say we have a problem P & for solving that problem we have many algo. or sol^n.



Now we need to analyze the best algo. for the problem and then write a program accordingly

So, the course Analysis & Design of Algo (ADA) will talk about how to design various algo. for a given problem and how to analyze them.

Analysis of Algorithm

Analysis of Algo. means developing a formula of how fast the algo. works based upon the problem size.

problem size:-

→ No. of input / output in an algo.

→ No. of operations involved in an algo.

The analysis of algo. based on time of computation is called as time Complexity of the algo.

Analysis of Algo.

Predicting the resources that an algorithm requires :-

- time (computational)
- memory space
- bandwidth ; etc.

Reason for predicting computational time :-

When we have multiple possible algo. to solve a partica problem we want to analyze those algo, we want to pick those algo, whose efficiency in terms of time is best and that is the algo. we implement.

Problem in Analysis

Exact No. of resources will depend on the exact program, prog. lang., hardware used, other jobs running on machine, etc.

Solution :- Define a generic machine and imagine running the algo. on it.

- Random Access Machine (RAM)

features :-

1) One processor
2) Execution is sequential
3) Instruction set and their cost in term of time.

Analysis of algo. is based upon :-

1) Worst case complexity
2) Best case complexity
3) Avg case complexity

* The worst case complexity of an algo. is the functi defined by the max. no. of steps taken on any instance of size n.

* The best case complexity of an algo. is the functio defined by the min. no. of steps taken on any instan of size n.

Example: Let's take an array of element.

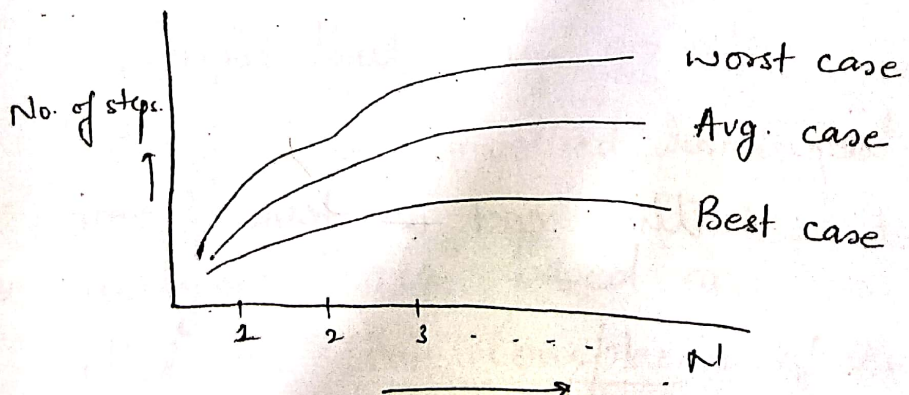
5	10	15	10	25

we want to find a no. 'x' in this array, we want to know whether it is present or not & if present then at which position.

let $x = 5 \longrightarrow$ 1 comparison $\longrightarrow$ best case

$x = 25$ or $35 \longrightarrow$ "x" comparisons $\longrightarrow$ worst case.
or 5.

$x = 15 \longrightarrow$ avg. case.



Order of growth of functions:

Measuring the performance of an algo. relation with the input size "n" is called order of growth.

The order of growth for varying input size n is as given below.

n	Logn	nLogn	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536

Asymptotatic Notations:

- It is used to describe the running time of an algo. This shows the order of growth of function mathematical
- We always analyze the algo. by calculating its time complexity ie. total time taken by the algo. to execute.
- We represents the time complexity with some notation called as Asymptotic Notation.

These are categorized as:-

1) Big-oh ('O')
2) Omega ('Ω')
3) Theta ('Θ')
4) little oh ('o')
5) little omega ('ω')

1. Big Oh Notation

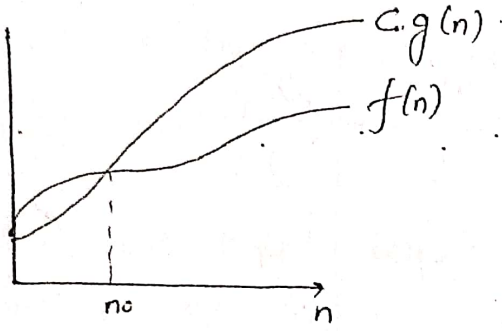
$$f(n) = O(g(n))$$

iff there exist positive constants c and n_0, such that

$$f(n) \leq c * g(n)$$ for all n, $n \geq n_0$, where

$c > 0$

$n_0 \geq 1$



It will give → Worst-case complexity.

eg: $f(n) = 3n + 2$

$g(n) = n$

we can say $f(n) = O(g(n))$

then $f(n) \leq c \cdot g(n)$ for some $c > 0$, $n_0 \geq 1$.

$3n + 2 \leq c \cdot n$

$3n + 2 \leq 4n$

$n = 1 \quad 5 \leq 4 \quad \times$

$n = 2 \quad 8 \leq 8 \quad \checkmark$

$n = 3 \quad 11 \leq 12 \quad \checkmark$

$\Rightarrow$ If $n \geq 2$, & if $C = 4$ then $f(n) \leq c\,g(n)$

$\Rightarrow$ $3n + 2 = O(n)$

Note:

If g(n) for this is satisfing then for every greater value than n which means $g(n) = n^2$ or n^3 or n^n or 2^n will also bound $3n+2$. Keep in mind that big oh is upper bound but always go for least upper bound or tighest bound which is 'n' here.

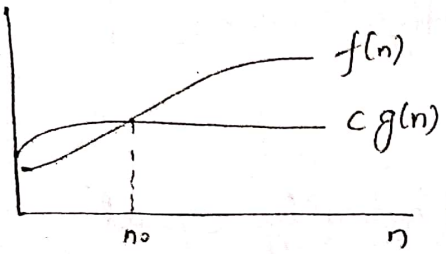
2.) Big omega (Ω)

This provides asymptotic **Lower Bound** and gives the **Best Case** complexity.

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

then there exist positive constants c & n_0, such that

$f(n) \geq c \cdot g(n) \quad \forall \; n \geq n_0$ $\quad$ $c > 0,\ n_0 \geq 1$



Rate of growth of $f(n)$ is lower bounded by the growth rate of $g(n)$.

eg: $f(n) = 3n+2$

$g(n) = n$

now check if it can be bounded.

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

then $f(n) \geq c \cdot g(n)$

$3n+2 \geq c \cdot n$, obviously $3n+2 \geq n$ even if $c = 1$ & $n_0 \geq 1$

$\therefore$ $3n+2 = \Omega(n)$

eg: $f(n) = 3n+2$

$g(n) = n^2$

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

$f(n) \geq c \cdot g(n)$

$3n+2 \geq c \cdot n^2$ for some n_0., can you find such c'

not possible.

* $f(n) = \Omega(n)$ — anything less than 'n' can be lower bounded.

$\Omega(n)$
↓
$\log n$
↓
$\log \log n$

But it is better to take the closest bound or tighest bound

3) Big theta ('Θ')

This will provide - Avg. case behaviour
It is also known as - tight bound.

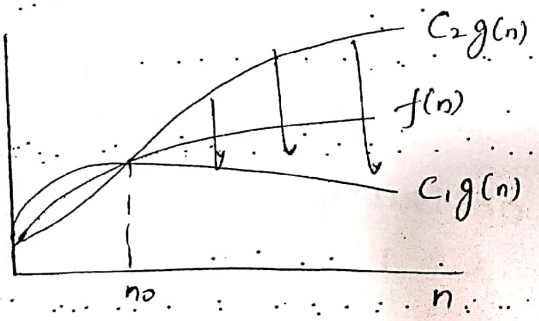
$f(n) = \Theta(g(n))$

$\Rightarrow$ $f(n)$ & $g(n)$ are asymptotically equal.

then there exists 3 +ve constants C_1, C_2 & n_0 such that

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

$n_0 \geq 1$.



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

$C_1(g(n)) \leq f(n) \leq C_2(g(n))$

Lower bound
'Ω'

upper bound.
'O'

"Θ"

eg: $f(n) = 3n+2$

$g(n) = n$

We have seen that

$f(n) \leq c_2 g(n)$

$3n+2 \leq 4(n) \rightarrow c_2 = 4, \quad n_0 \geq 2.$

&

$f(n) \geq c_1 g(n)$

$3n+2 \geq n \quad n_0 \geq 1, \; c = 1.$

$\therefore$ $3n+2$ is bounded both by upper & lower by n.

$\Rightarrow \boxed{3n+2 = \Theta(n)}$

We can say these both are asymptotically equal.

4) little 'o'.

$f(n) = o(g(n))$

$\rightarrow$ $g(n)$ is strictly larger than $f(n)$

$\boxed{f(n) < c(g(n))} \quad \forall \quad n \geq n_0$

$$\boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0}$$

eg: $2^n = o(3^n)$

$$\lim_{n \to \infty} \frac{2^n}{3^n} = \lim_{n \to \infty} \left(\frac{2}{3}\right)^n = 0.$$

$n = o(n)$ ✗

$n = o(n^2)$ ✓

$n = o(n^3)$ ✓

$\log n = o(n)$ ✓

$\sqrt{n} = o(n)$ ✓

$\log \log n = o(\log n)$ ✓

5) Little 'ω'

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

$\Rightarrow$ $g(n)$ is strictly smaller than $f(n)$

$$\boxed{f(n) > c\,g(n)} \quad \forall \; n \geq n_0$$

$$\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty}$$

eg $2^{2n} = \omega(2^n)$

$$\lim_{n\to\infty} \left(\frac{2^{2n}}{2^n}\right) = \lim_{n\to\infty} 2^n = \infty$$

$$\left[2^{2n} - 2^n = 2^n\right]$$

$\rightarrow$ $n = \omega(\log \log n)$

~~$\log n = \omega(\log$~~

$\rightarrow$ $\frac{n^2}{2} = \omega(n)$

* Golden Rule

The cost will always depend on the dominating term ignore lower degree term & its associated constant.

* If an algo has time complexity of linear form than it is better than the algo having quadratic time complex

$T(n)$ — linear, quadratic, cubic.

Insertion Sort

Gunjan

Sorting Problem :-

Input : A sequence of n numbers $\langle a_1, a_2, a_3 \ldots a_n \rangle$

Output : A permutation $\langle a_1', a_2', a_3', \ldots a_n' \rangle$ of input sequence that $a_1' \leq a_2' \leq a_3' \ldots \leq a_n'$.

The no. that we wish to sort are also called as **keys**.

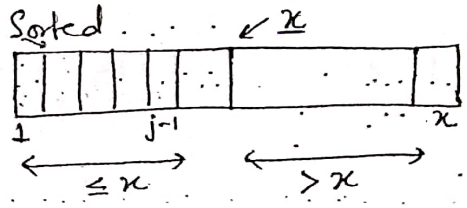
An **instance** of a problem consists of the input needed to compute a solution to the problem.

Insertion Sort

It is an efficient algo. for sorting a small no. of elements.

Works :- the way many people sort a hand of playing cards

We start with an empty hand ie left hand and cards face down on the table. When we remove one card at a time from the table, insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of cards already in hand from right to left:



J-1 elements have been picked up & inserted into an output array, which is maintaining the partially sorted array

∴ first j-1 elements have been inserted & then sorted.

→ In the j^th step, we will pick up the j^th elements & let's call it as 'x', next we have to find out the location where x needs to go.

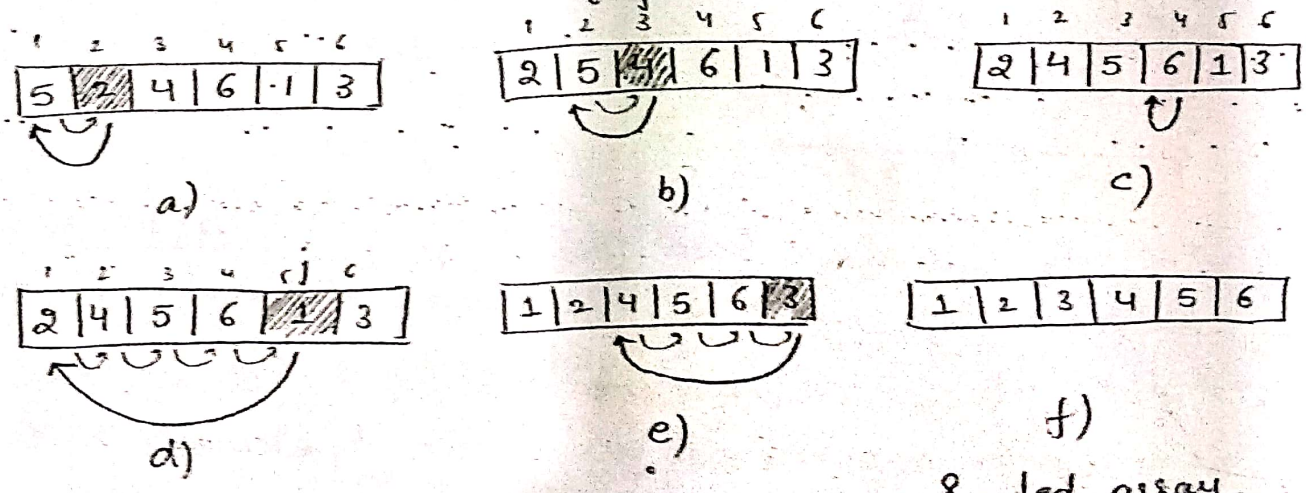
How?

We know that the sequence is already sorted, so in general left of x will be less & right will be greater. So we start scanning the array from right end A[j-1] and we will start shifting the elements one place to right as we scan from left.

INSERTION-SORT (A)

		check if
1. for j = 2 to A.length	— C_1	n ($n-1+1$)
2. Key = A[j]	— C_2	$(n-1)$
3. // Insert A[j] into the sorted sequence A[1....j-1]	— 0	
4. i = j-1	— C_4	$(n-1)$
5. while i > 0 and A[i] > Key	— C_5	$\sum_{j=2}^{n} t_j$ (t_j-1)
6. A[i+1] = A[i]	— C_6	$\sum_{j=2}^{n} (t_j-1)$
7. i = i-1	— C_7	$\sum_{j=2}^{n} (t_j-1)$
8. A[i+1] = Key	— C_8	$(n-1)$

eg A = <5, 2, 4, 6, 1, 3>



In each iteration, the black rectangle holds the key taken from A[j], which is compared with the values in rest places to its left in the test of line 5.

Shadded arrows show array value moved one position to the right in line 6.

Black arrays indicate where the ~~key~~ moves to in line 8.

f → final sorted array.

Calculating time Complexity

The running time of an algo. is the sum of running times for each statement executed.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case - already sorted array. $\therefore t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Linear function : $(an+b)$-form

Worst case $(t_j = j)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) - (c_2 + c_4 + c_5 + c_8)$$

(an^2+bn+c)-form

$$= O(n^2)$$

Merge Sort

It closely follows the divide-and-conquer paradigm.

Divide :- Divide the n-element sequence to be sorted into 2 subsequence of $n/2$ elements each.

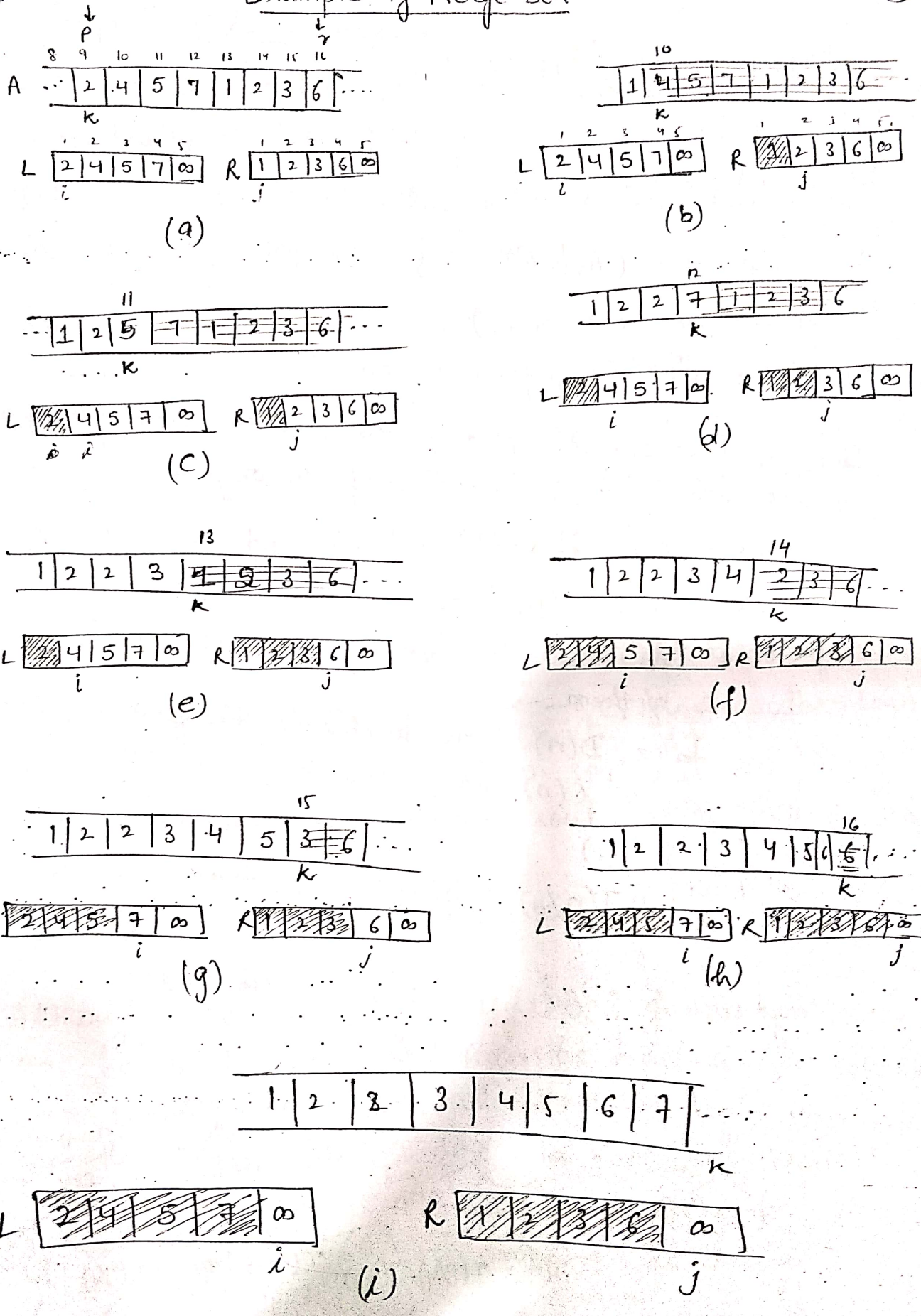
Conquer :- Sort the 2 subsequences recursively using merge s...

Combine :- Merge the 2 sorted subsequences to produce the sorted answer.

MERGE (A, p, q, r)

1. $n_1 = q - p + 1$ ⟶ compute length n_1 of $A[p \ldots q]$
2. $n_2 = r - q$ ⟶ compute length n_2 of $A[q+1 \ldots r]$
3. Let $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ be new arrays.
   (Left array, Right array)
4. for $i = 1$ to n_1
5. $L[i] = A[p+i-1]$] copies subarray $A[p..q]$ in $L[1..n_1]$
6. for $j = 1$ to n_2
7. $R[j] = A[q+j]$] copies subarray $A[q+1 \ldots r]$ into $R[1..n_2$
8. $L[n_1+1] = \infty$
9. $R[n_2+1] = \infty$] put the sentinels at end of array L...
10. $i = 1$
11. $j = 1$
12. for $k = p$ to r
13. if $L[i] \leq R[j]$
14. $A[k] = L[i]$
15. $i = i+1$
16. else $A[k] = R[j]$
17. $j = j+1$

Example of Merge Sort



(A)

MERGE - SORT (A, p, r)

1) if $p < r$
2) $q = \lfloor (p+r)/2 \rfloor$
3) MERGE - SORT (A, p, q)
4) MERGE - SORT (A, q+1, r)
5) MERGE (A, p, q, r)

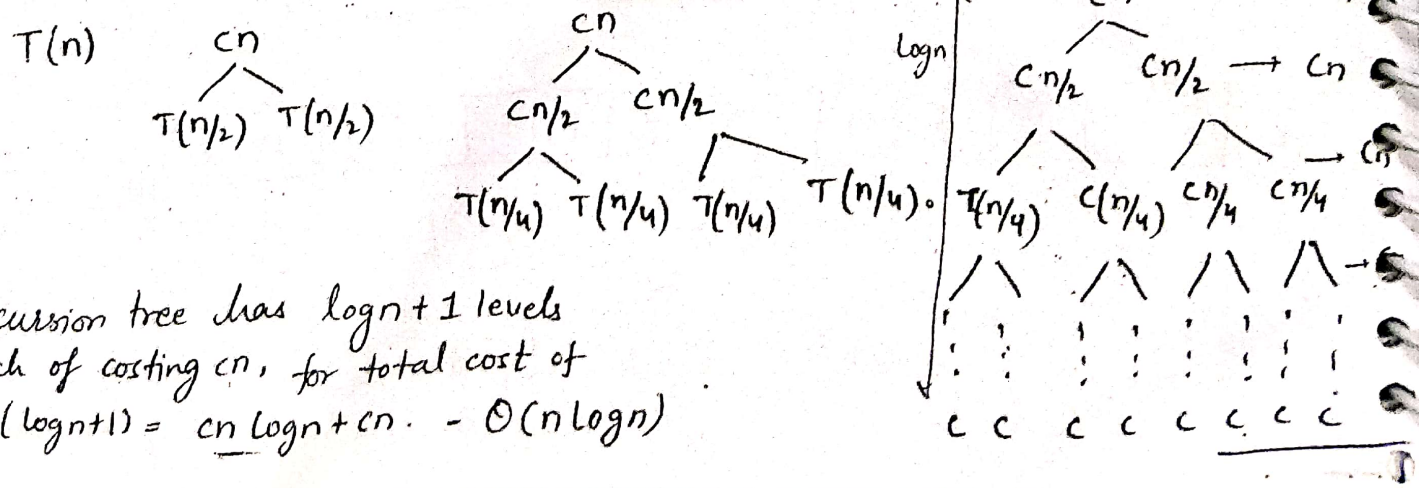
Suppose that our division of the problem yields a subproblem each of which is 1/b the size of the original.
(for merge sort $a = b = 2$)

It takes time $T(n/b)$ to solve one subproblem of size n/b, so it takes time $aT(n/b)$ to solve a of them.

If we take $D(n)$ time to divide the problem into subproblems & $C(n)$ time to combine the soln.

$$T(n) = \begin{cases} \Theta(1) & ; \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & ; \text{otherwise} \end{cases}$$

$$\text{Merge sort} = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & n > 1, \end{cases}$$



recursion tree has $\log n + 1$ levels each of costing cn, for total cost of $cn(\log n + 1) = cn \log n + cn$. $- \Theta(n \log n)$

Algorithms as a Technology

Different algo. used to solve the same problem often differ dramatically in their efficiency.

Let us consider 2 algorithms insertion sort and merge sort for sorting a sequence of elements.

Insertion sort takes time → $C_1 n^2$ to sort n items where C_1 is a constant that does not depends on 'n'.

Merge sort takes time roughly equal to $C_2 n \log n$, where C_2 is another constant & that also does not depend on n.

Insertion sort typically has a smaller constant factor than merge sort, so that $\boxed{C_1 < C_2}$

for Insertion sort we can write → $C_1 n^2$ as $C_1 n \cdot n$

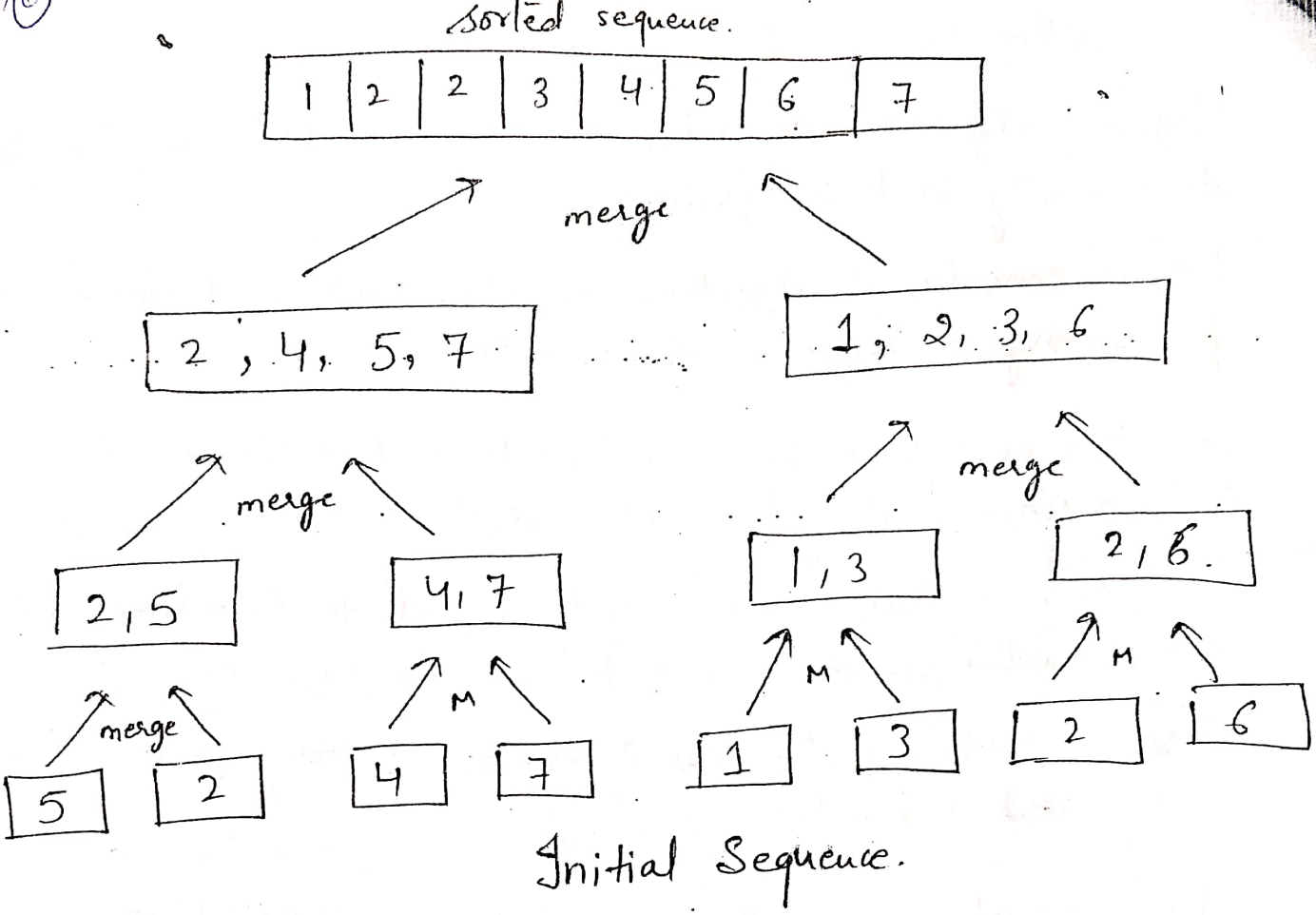
for merge sort we can write → $C_2 n \log n$ as $C_2 n \cdot \log n$.

Then we see that where insertion sort has a factor of 'n' in its running time, merge sort has a factor of $\log n$ which is much smaller.

eg. when $n = 1000$, $\log n \simeq 10$

Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\log n$ vs n will more than compensate for diff. in constant factors.

eg. faster computer (A) — running Insertion sort
slower computer (B) — " merge sort.
Each must store 10 million nos.


Sorted sequence.
1 2 2 3 4 5 6 7
merge
2, 4, 5, 7
1, 2, 3, 6
merge
merge
2, 5
4, 7
1, 3
2, 6
merge
M
M
M
5
2
4
7
1
3
2
6
Initial Sequence.

Merge Sort (imp points)

→ In merge sort, recursion ends or "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The Key operation of the merge sort algo. is the merging of 2 sorted sequences in the "combine" step. We ~~attain~~ merge by calling an auxiliary procedure MERGE (A, p, q, r) where A is an array and p, q & r are indices into the array such that $p \leq q < r$.

* The procedure assumes that subarrays A[p..q] and A[q+1...r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p...r].

Our MERGE procedure takes time $\boxed{O(n)}$, where $n = r - p + 1$ is the total no. of elements being merged.

⇒ 2 piles of playing cards, face up in sorted sequence. By merging these 2 piles we need 1 sorted pile face down. We repeat this step until one input pile is empty, at which time we just take the remaining input pile & place it face down onto the output pile. Computationally, each basic step takes constant time since we are comparing just 2 top cards. Since we perform at most 'n' basic steps, merging takes $O(n)$ time.

Analysis

$$T(n) = \begin{cases} \Theta(1) & ; \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & ; \text{otherwise} \end{cases}$$

$D(n) \rightarrow$ time to divide the problem into subproblems

$C(n) \rightarrow$ time to combine the solutions to subproblems into the soln to original problem

(11)

Suppose computer A → executes 10 billion instructions per sec & computer B → executes 10 million instruction per sec. so that A is 1000 times faster than B in raw computing power.

Now suppose
Computer A → efficient compiler, good programmer.
Computer B → avg. programmer implements merge sort, using high-level lang with an inefficient compiler with resulting code taking $50n\log n$ instruct

To sort 10 million no., computer A takes

$$\frac{2 \times (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/sec}} = 20{,}000 \text{ sec.} \simeq 5.5 \text{ hrs.}$$

While computer B takes

$$\frac{50 \cdot 10^7 \log 10^7 \text{ instructions}}{10^7 \text{ instructions/sec}} \simeq 1163 \text{ sec} \simeq 20 \text{ min.}$$

By using an algo whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A.

$$\log n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n < n!$$

We add the costs across each level of the tree.

Top level → cn

next level → $c\left(\frac{n}{2}\right) + c\left(\frac{n}{2}\right) = 2\frac{cn}{2} = cn.$

next level → $c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) = 4\frac{cn}{4} = cn.$ & so on.

In general

~~level i^{th} level below the top has total cost $2^i c\left(\frac{n}{2^i}\right) = cn.$~~

level i^{th} level below the top has ~~cost~~ 2^i nodes, each contributing a cost of $c\left(\frac{n}{2^i}\right)$

total cost = $2^i \times \frac{cn}{2^i} = cn.$

The bottom level has 'n' nodes, each contributing a cost for a total cost of cn.

Total no. of levels in recursion tree = $\log_2 n + 1$.

$n \rightarrow$ no. of leaves, corresponding to input size.

Best case, occurs when $n=1$, in which case the tree has only 1 level.

Since $\log 1 = 0$, we have that $\log n + 1$ gives correct no. of levels.

Computation

Total level = $\log n + 1$.

each costing = cn

$\therefore$ total cost = $cn(\log n + 1)$

$= cn \log n + cn$

ignoring low-order term & constant c gives the desired result of $= \Theta(n \log n)$

For merge sort (Analysis)

Although the pseudocode for MERGE-SORT works correctly when the no. of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2.
Each divide step then yields 2 subsequences of size exactly $n/2$.

⇒ Merge sort on just 1 element → constant time
when $n > 1$ elements, we break down the running time as follows :-

Divide → divide step just computes the middle of subarray which takes const. time. $D(n) = \Theta(1)$

Conquer → we recursively solve 2 subproblems, each of size $n/2$, which contributes $2T(n/2)$ time.

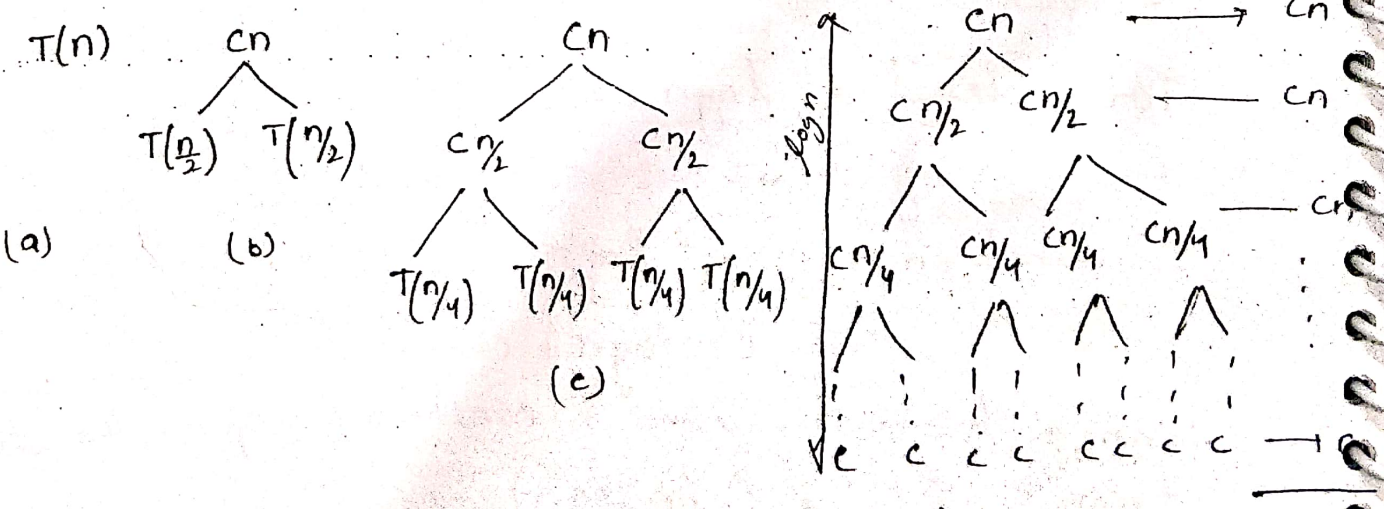
Combine → Merge procedure on n-element : $C(n) = \Theta(n)$

$$\therefore \quad T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Let

$$T(n) = \begin{cases} c & ; \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & ; \text{if } n > 1. \end{cases}$$

$c \rightarrow$ constant



QUICKSORT

Gunjan

Quick is based upon divide and conquer paradigm.

Divide : Partition the array $A[p \ldots r]$ into 2 subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that each element in $A[p \ldots q-1]$ is less than $A[q]$ and each element in $A[q+1 \ldots r]$ is greater than $A[q]$.

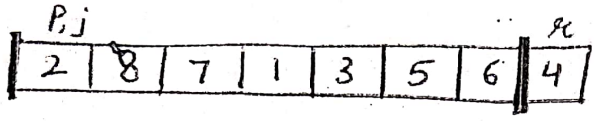
Conquer : Sort the 2 subarrays $A[p \ldots q-1]$ & $A[q+1 \ldots r]$ by recursive calls to quicksort

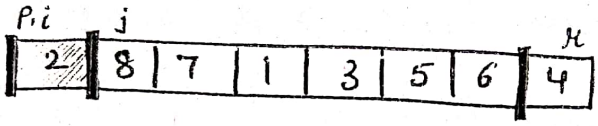
Combine : Since the subarrays are sorted ∴ No need to combine the entire array, Now $[p \ldots r]$ is sorted

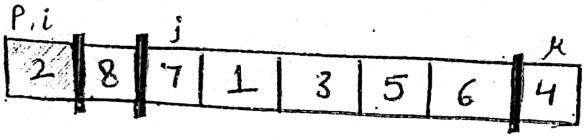
```
QUICKSORT (A, p, r)
1. if p < r
2.     q = PARTITION (A, p, r)
3.     QUICKSORT (A, p, q-1)
4.     QUICKSORT (A, q+1, r)
```

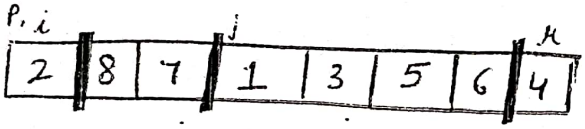
```
PARTITION (A, p, r)
1. x = A[r]  ⟶ pivot element
2. i = p-1
3. for j = p to r-1
4.     if A[j] ≤ x
5.         i = i+1
6.         exchange A[i] with A[j]
7. exchange A[i+1] with A[r]
8. return i+1
```

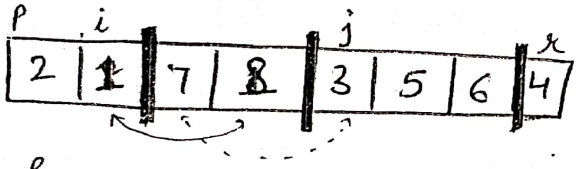
] the pivot element is swapped so that it lies between the two partitions.

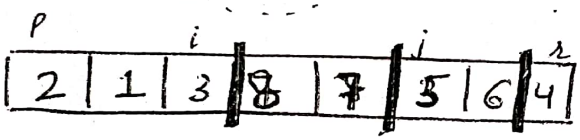
a) 

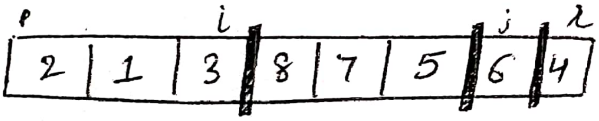
b) 

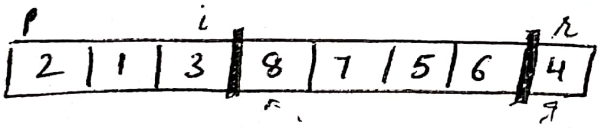
c) 

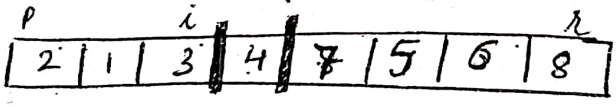
d) 

e) 

f) 

g) 

h) 

i) 

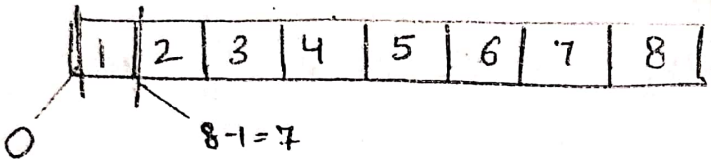
Performance Analysis of Quicksort :-

The running time of quicksort depends on whether the partitioning is balanced or unbalanced.

1) Worst case Partitioning :-

The worst case behaviour for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements.

for eg: when the list is already sorted.



Let us assume that this unbalanced partitioning arises in each recursive call.

The partitioning costs $\Theta(n)$ time.

Since the recursive call on an array of size 0 just returns $T(0) = \Theta(1)$

recurrence for the running time is

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

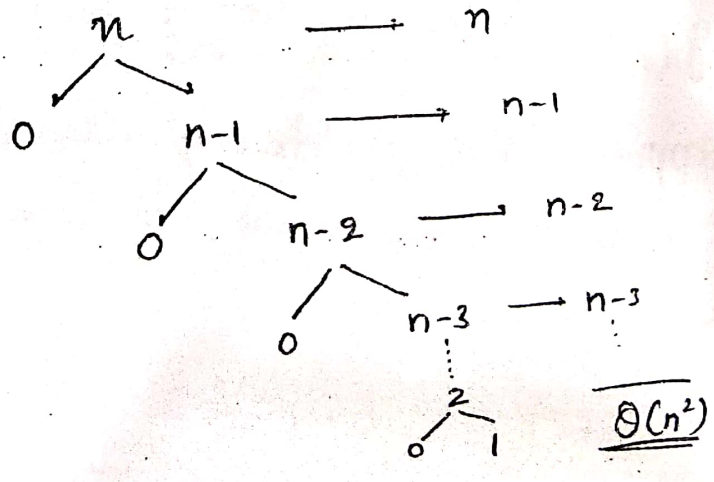
Solved by substitution method

$$T(n) = \Theta(n^2)$$

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running is $\Theta(n^2)$.

∴ the worst-case running time of quicksort is not better than that of insertion sort.

Array already sorted time	Insertion sort $\Theta(n)$	Quick sort $\Theta(n^2)$



$\Theta(n^2)$

2) Best case Partitioning:

PARTITION produces 2 subproblems, each of size no more than $n/2$. Since one is of size $\frac{n}{2}$ and one of size $\left\lceil \frac{n}{2} \right\rceil - 1$.

⇒ In this case, quicksort runs much faster.

Recurrence for running time is -

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

Now it satisfies case 2 of Master Method

$a = 2$; $b = 2$ $\quad n^{\log_2 2} = n$ & $f(n) = n$

Complexity is $O(n \log n)$

Balanced Partitioning

The average case running time of quicksort is much closer to the best case than to the worst case.

In best case, we have $n/2$ & $n/2$ partitioning.

In worst case we have $n-1$, 0 partitioning

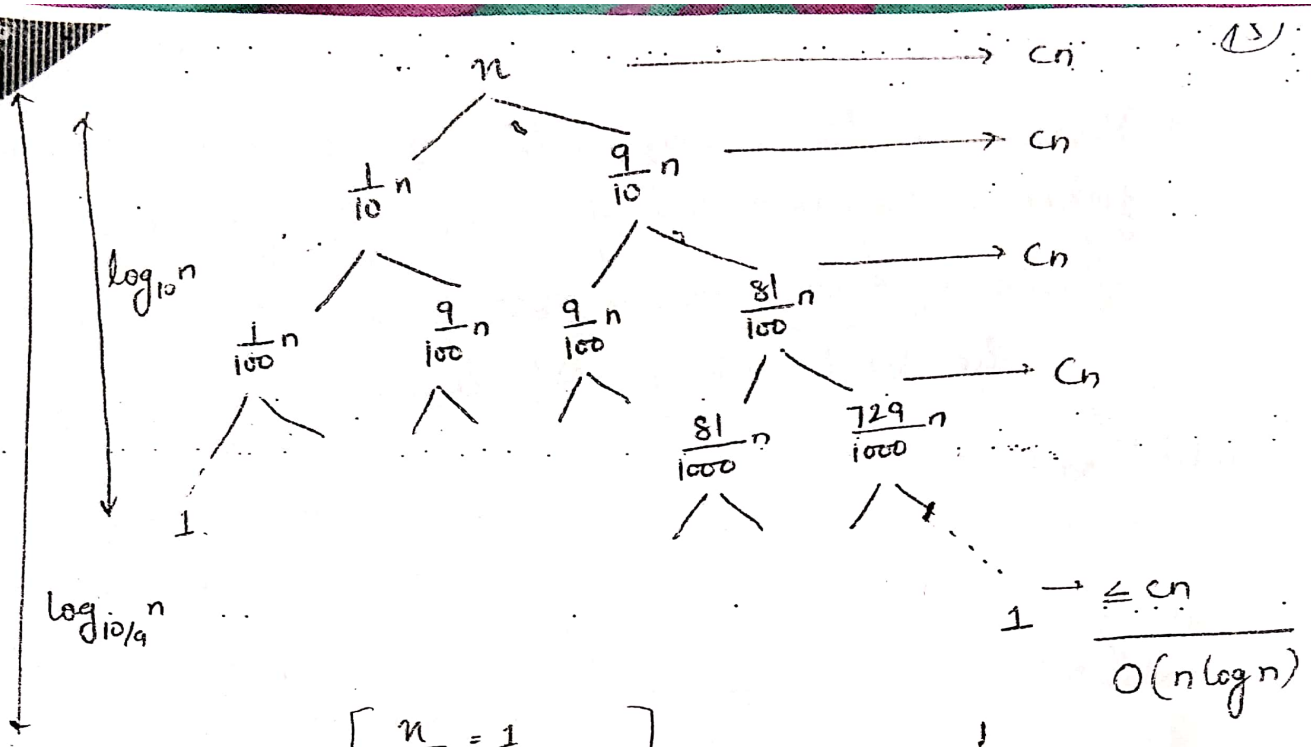
In average case we have partitioning somewhere b/w best & worst case.

Best case → good split

worst case → bad split

Avg. case → combination of good & bad split.

Suppose, the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced.



$$\left[\begin{array}{l} \frac{n}{\left(\frac{10}{9}\right)^i} = 1 \\ n = \left(\frac{10}{9}\right)^i \\ i = \log_{10/9} n \end{array}\right]$$

We then obtain the recurrence

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

In recursion tree, every level of tree has cost cn, until a boundary condition is reached at depth

$$\log_{10} n = \Theta(\log n)$$

levels have cost at most cn.

Recursion terminates at depth $\log_{10/9} n = O(\log n)$ with cost cn.

∴ the total cost of quicksort is $\Theta(n \log n)$.

In avg case we take random no. of splits. Even if we take 99 to 1 split, this also yields $\Theta(n \log n)$ running time.

Recurrence Relation

A Recurrence Relation is an inequality / equality which is expressed in terms of itself with smaller input.

Significance

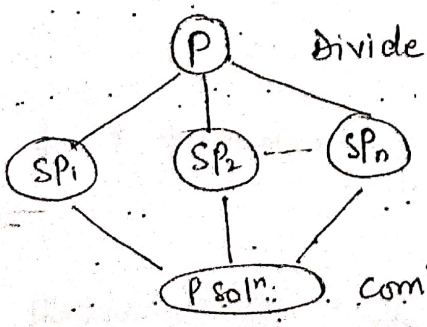
Algorithm design technique "Divide Conquer Combine Technique" is widly used to solve queries.

Recurrence relation can find the time Complexity of the Algo. by using techniques based on "Divide conquer Combine".

We solve a problem recursively, applying 3 steps at each level of recursion in 'Divide & conquer problem techniques"

1) Divide 2) Conquer 3) Combine.

A bigger problem is divided into smaller onces and then technique is applied on those subparts ie. concquer by recursively solve the problems & then combine back.



$T(n) = T(n/3) + T(n/4) + n$

↓ original problem

Divided into 2 subproblems.

⇒ Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out and we have gotten down to base case.

Methods for solving recurrences :-

i) (ie. for obtaining asymptotic "Θ" or "O" bounds on the solution")

1) Substitution Method :- we guess a bound and then use mathematical induction to prove our guess correct.

2) Iteration Method :- also known as backward substitution.

3) Recursion-tree Method :- converts the recurrence into a tree whose nodes represents the costs incurred at various levels of the recursion.

4) Master method :- provide bounds for recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and $f(n)$ is a given function.

* The recurrence of the form of Master method characterizes a divide & conquer algo. that creates a subproblems, each of which is $1/b$ the size of original problem & in which the divide & combine steps together take $f(n)$ time.

(1) Substitution Method

The substitution Method for solving recurrences comprises of 2 steps :-

1. Guess the form of the solution.
2. Use mathematical induction to find the constants & show that the soln works.

eg: $$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases} \quad \text{——— (1)}$$

Sol^n Guess $\rightarrow O(n^2)$

∴ If we wish to prove $T(n) = O(n^2)$,
then it is sufficient to show $T(n) \leq c n^2$ for $c > 0$ & $\forall\ n > n_0$

1st we will assume

$$T(\alpha) \leq c(\alpha)^2 \quad \text{for} \quad 1 \leq \alpha \leq n-1$$

Then we use this induction hypothesis to show that $T(n) \leq cn^2$. By substituting the induction hypothesis in eqn (1).

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\leq c(n-1)^2 + n \\ &\leq c(n^2 + 1 - 2n) + n \\ &\leq cn^2 + c - 2n + n \\ &\leq cn^2 - 2n + c + n \end{aligned}$$

$$\therefore \quad \boxed{T(n) \leq cn^2}$$

This holds for all $n \geq 1$ & $c = 1$. $[n_0 \geq 1]$

from eqn 1. we have $T(1) = 1$

$T(1) = c(1)^2$

when $c = 1$

$T(1) = 1$. will also satisfy this base condn.

Hence $\boxed{T(n) = O(n^2)}$

2) $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Sol^n Guess $\Rightarrow T(n) = O(n \log n)$

ie $T(n) \leq cn \log n \quad \forall \quad n \geq n_0$.

Substitute in eq^n ①.

$$T(n) \leq 2c\left\lfloor\frac{n}{2}\right\rfloor \log\left\lfloor\frac{n}{2}\right\rfloor + n$$

$$\leq cn \log\left\lfloor\frac{n}{2}\right\rfloor + n$$

$n \rightarrow$ no. of elements, which is very large.

$\therefore \frac{n}{2}$ is also very large.

Hence $\frac{n}{2} \cong \left\lfloor\frac{n}{2}\right\rfloor \simeq \left\lceil\frac{n}{2}\right\rceil$

Now. $T(n) \leq cn \log \frac{n}{2} + n$ → $\log_2 2 = 1$

$$= cn \log n - cn \log 2 + n$$

$$= cn \log n + n - cn$$

$$T(n) \leq cn \log n \quad , \quad c \geq 1$$

By induction hypothesis we require to show that our soln. holds for boundary condition ie.

$T(1) \leq c(1) \log 1 = 0$ ($\log_2 1 = 0$)

Thus for any value of c, this will not hold, so by asymptotic definition we need to prove.

$T(n) \leq cn \log n \quad \forall \; n \geq n_0$

$T(2) \leq c(2)(\log 2)$

$T(3) \leq c(3)(\log 3)$

as 'n' can't be 1, this relation holds for $c \geq 2$

Thus, $\boxed{T(n) = O(n \log n) \text{ is true.}}$

Trick

* How to make a good guess in substitution method.

for the given recurrence relation ie.

$$T(n) = aT(n/b) + f(n)$$

Case 1: If $a = b = 2$, then

$T(n) = O(f(n) \log n)$ is a good guess.

Case 2: If $a = 1$, b = any value, then

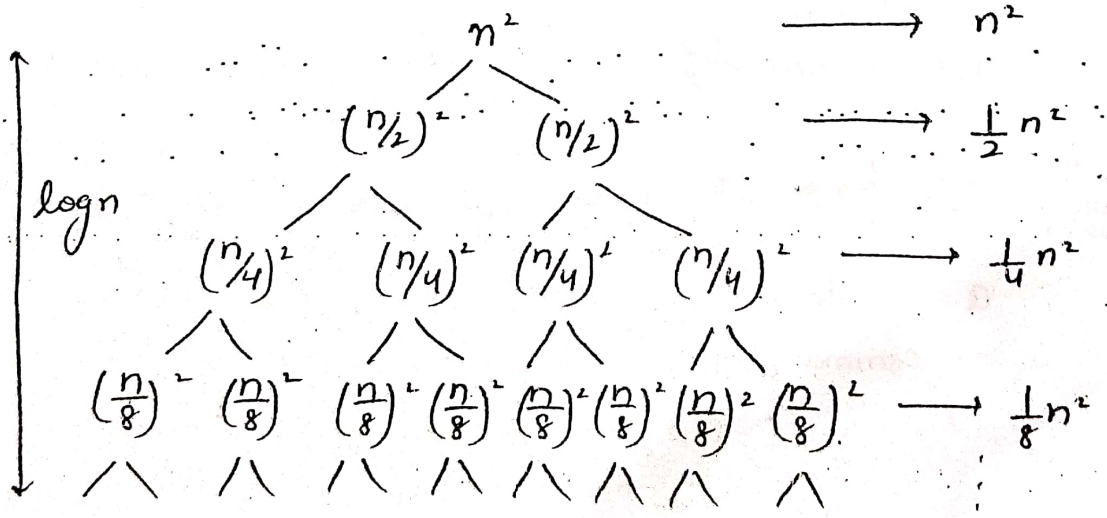
$T(n) = O(\log n)$

Case 3: If $a \neq 1$, $b > a$, then

$T(n) = O[f(n).\ n]$ is a good guess.

Recurrence Tree method

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



$O\ n^2$

Cost of eqn

Summing the values at each level, we get

$$= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots$$

$$= n^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$$

$$= n^2 \sum_{i=0}^{\log n} \frac{1}{2^i} \leq n^2 \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= n^2 \frac{\frac{1}{2^0}}{1 - \frac{1}{2}} = 2n^2 = \Theta(n^2)$$

Useful formulae :

1) $\sum_{i=1}^{n} \frac{1}{a^i}$ if common ratio is less than 1, then convert it in infinite GP series ie. $\sum_{i=1}^{\infty} \frac{1}{a^i}$

2) $\sum_{i=0}^{\infty} \frac{1}{a^i}$ for infinite G.P series, formula is $\frac{a}{1-r}$

a = 1st term
r = common diff.

3) $\sum_{i=0}^{\infty} a^i$, for finite GP series, formula is $\frac{a(r^n - 1)}{r - 1}$

a = 1st term
r = common diff

② Iterative Method

This technique is also known as backward substitution. In this method, we simply continue to substitute back until we see a pattern of some sort. We then deduce a formula from the pattern.

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

original problem

1 subproblem of size $(n-1)$

$\boxed{T(1) = 1}$ base case

↳ Time to solve the problem of unit size is 1.

eg: $T(n) = T(n-1) + 1$

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

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

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

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

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

$\vdots$

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

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

$= 1 + n - 1$

$= n$

$$\boxed{T(n) = n = \Theta(n)}$$

eg2: $T(n) = T(n-1) + n$, base case $= T(n) = 1$. $n=1$.

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

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

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

(+1, +1, +1)

$\vdots$

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

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

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

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

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

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

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

Imp: $T(n) = T(n-1) + 1 = O(n)$

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

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

$\vdots \qquad \vdots$

$$\boxed{T(n) = T(n-1) + n^k = O(n^{k+1}) \quad k \geq 0}$$

eg: $T(n) = \begin{cases} T(n/2) + c & ; n > 1 \\ 1 & n = 1. \end{cases}$ — (1)

$n = 1 \rightarrow$ base case

$T(n) = T(n/2) + c$ put $n = 1/2$

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

$T(n/2) = T(n/4) + 2c$ put this value in (1)

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

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

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

$T(n/4) = T(n/8) + 2c + c$

$= T(n/8) + 3c$

$\vdots$ n times

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

$= T(1) + kc$

$= 1 + \log_2 n$

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

$n = 2^k$

$k = \log_2 n$

$$\boxed{T(n) = O(\log_2 n)}$$

eg: $T(n) = \begin{cases} 2T(n/2) + n & ; n > 1 \\ 1 & ; n = 1. \end{cases}$

$T(n) = 2T(n/2) + n$ ——— (1)

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

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

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

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

$= 8T(n/8) + 3n$

⋮

k times

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

$\left[\begin{array}{l} 2^k = n \\ k = \log_2 n \end{array}\right]$

$= nT(1) + kn$

$= n + kn$

$= n + \log_2 n \cdot n$

$= n + n\log_2 n$

$\boxed{T(n) = O(n \log n)}$

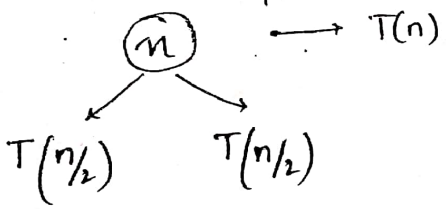
3) Recursion - Tree Method

Steps:

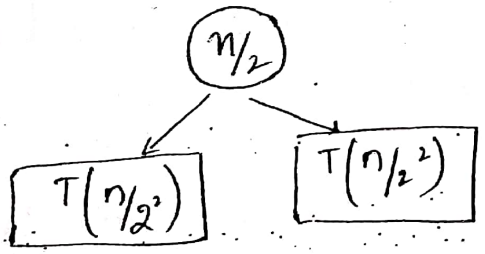
1. Construct the Recurrence Tree
2. find the cost at each level
3. find the overall cost
   → represents the time complexity.

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

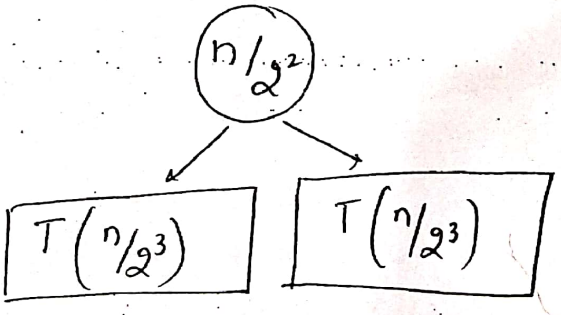
The term which is free from 'T' will be the root of tree
Bcoz we have 2 subproblems of size $(n/2)$ ∴ 2 sub-branches



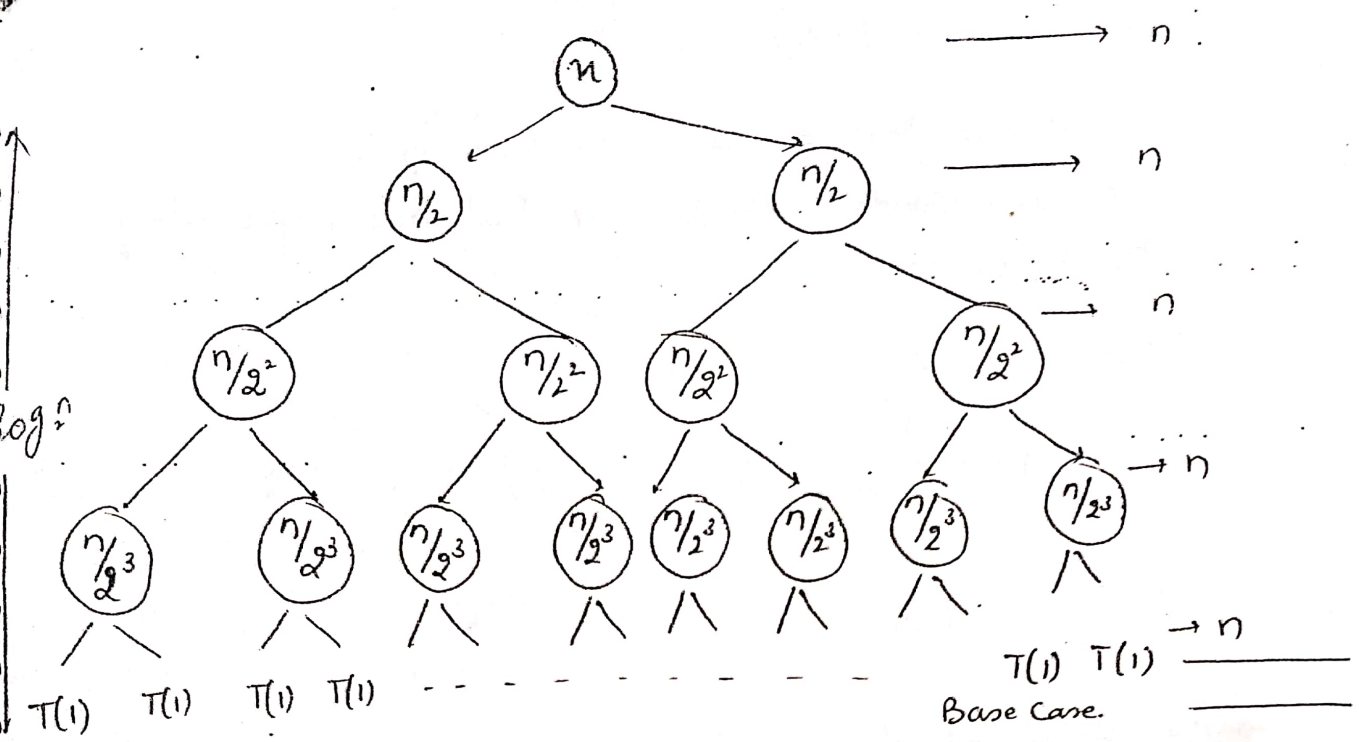
eqn: $T(n/2) = 2T(n/2^2) + n/2$



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



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



Overall cost

$= n + n + n \cdots\cdots n$ (No. of level times)

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \frac{n}{2^3} \rightarrow \cdots \left(\frac{n}{2^k} = 1\right)$

$n = 2^k$

$\boxed{k = \log_2 n}$ ⟶ height of tree.

Cost = n times the height of tree

$= n \log_2 n$

$\boxed{T(n) = O(n \log_2 n)}$

$= n(\log_2 n + 1)$

$= n \log_2 n + 1$

$= O(n \log_2 n)$

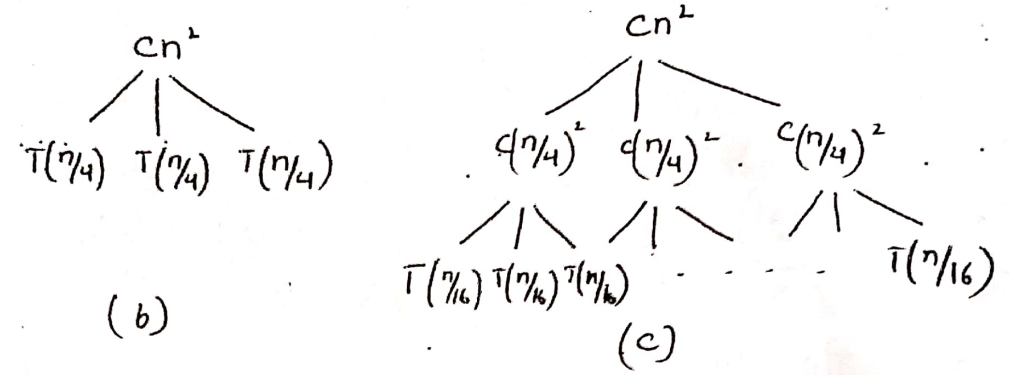
Ht. of binary tree $= \log_2 n$

Level of binary tree $= \log_2 n + 1$.

eg: $T(n) = \underset{\text{No. of subproblems}}{3T(n/4)} + \underbrace{cn^2}_{\text{cost.}}$

⇒ We assume that n is exact power of 4.

$T(n)$ is expanded into an equivalent tree representing the recursion.



(a) (b) (c)

⇒ cn^2 term at the root represents the cost at the top level of recursion.

⇒ Three subtrees of the root represents the costs incurred by the subproblems of size $n/4$.

⇒ Part (c) shows this process carried one step further by expanding each node with cost $T(n/4)$ from (b)

⇒ The cost for each of three children of the root is $c(n/4)^2$

⇒ The subproblem size for a node at depth i is $n/4^i$

⇒ Boundary condition is when $\frac{n}{4^i} = 1$.

⇒ $\frac{n}{4^i} = 1$, $n = 4^i$ ⇒ $i = \log_4 n$.

Tree has total $\log_4 n + 1$ levels.

⇒ Now we determine the cost at each level

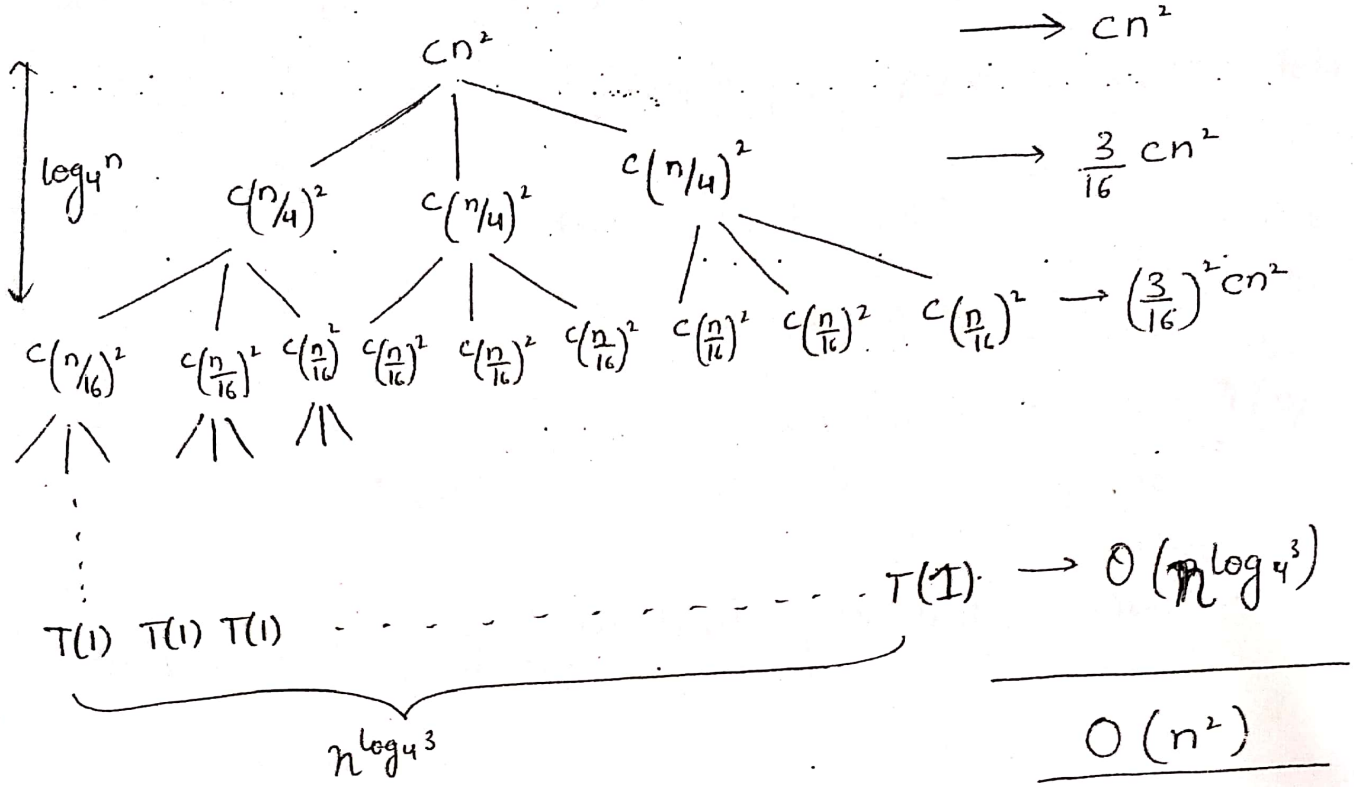
⇒ Each level has 3 times more nodes than the level above

⇒ No. of nodes at depth i is 3^i

∴ Last level, at depth i [$i = \log_4 n$] has 3^i nodes

means → $3^{\log_4 n}$ nodes. $3^{\log_4 n} = n^{\log_4 3}$

$= n^{\log_4 3}$ nodes



Last level has $n^{\log_4 3}$ nodes, each contributing cost T(1) for a total cost of $n^{\log_4 3} T(1)$ which is $\Theta(n^{\log_4 3})$.

→ Adding the cost at each level for the entire tree

$$T(n) = cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n + 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

$\sum_{i=0}^{\infty} \frac{1}{q^i} = \frac{1}{1-q}$

(The cost of root dominates the total cost of tree)

Now, we can use substitution method to verify that our guess was correct.

$T(n) = O(n^2)$ — upper bound

$T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$

we want to show that $T(n) \leq dn^2$ for some const. $d > 0$.

$$T(n) \leq 3T(\lfloor \tfrac{n}{4} \rfloor) + cn^2$$

$$\leq 3d \lfloor \tfrac{n}{4} \rfloor^2 + cn^2$$

$$\leq 3d \left(\tfrac{n}{4}\right)^2 + cn^2$$

$$= \frac{3}{16} dn^2 + cn^2$$

$$\leq dn^2$$

where last step holds as long as $d \geq \left(\frac{16}{13}\right)c$.

4.) Master Method:

It provides a "cook book" method for solving recurrences of the form:

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

where $a \geq 1$ & $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The above recurrence describes the running time of an algo. that divides a problem of size 'n' into 'a' subproblems each of size n/b, where a & b are positive constants. The 'a' subproblems are solved recursively, each in time $T(n/b)$. The cost of dividing the problem & combining the results of the subproblems is described by the function $f(n)$.

The master method depends on the master theorem.

Master Theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and $T(n)$ be defined on the non-negative integers by recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret $\frac{n}{b}$ to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$.

Then $T(n)$ has the following asymptotic bounds:-

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some const $\epsilon > 0$,
   then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some const. $\epsilon > 0$, if $af(n/b) \leq cf(n)$ for const. $c < 1$ & all sufficiently large n,
   $T(n) = \Theta(f(n))$.

eg 1: $T(n) = 9T\left(\frac{n}{3}\right) + n$

Sol^n: $a = 9$, $b = 3$, $f(n) = n$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = O(n^{\log_3 9 - 1}) = O(n)$$

hence we apply case 1, $\therefore$ sol^n is $\Theta(n^{\log_b a})$

$$T(n) = \Theta(n^{\log_b a})$$
$$= \Theta(n^{\log_3 9})$$
$$= \Theta(n^2).$$

eg 2: $T(n) = T\left(\frac{2n}{3}\right) + 1.$

$a = 1$, $b = \frac{3}{2}$, $f(n) = 1.$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1.$$

which is equal to $f(n)$. $\therefore$ apply case 2.

$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$
$$= \Theta(1 \log n)$$
$$= \Theta(\log n).$$

eg 3: $T(n) = 3T\left(\frac{n}{4}\right) + \cancel{n \log n}\ n^2$

$a = 3$, $b = 4$, $f(n) = \cancel{n \log n}\ n^2$

$\Omega(n^{\log_b a + \varepsilon})$
$= (n^1)$

$f(n) = \Theta(n^2)$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$$

$f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ where $\varepsilon = 0.2$, case 3 applies

for larger n, we have $af(n/b) = 3(n/4)\log(n/4) \leq \left(\frac{3}{4}\right) n \log n = cf(n)$

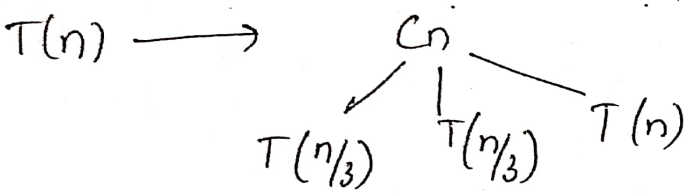
$c = 3/4.$

$\therefore T(n) = \Theta(f(n)) = \Theta(n \log n).$

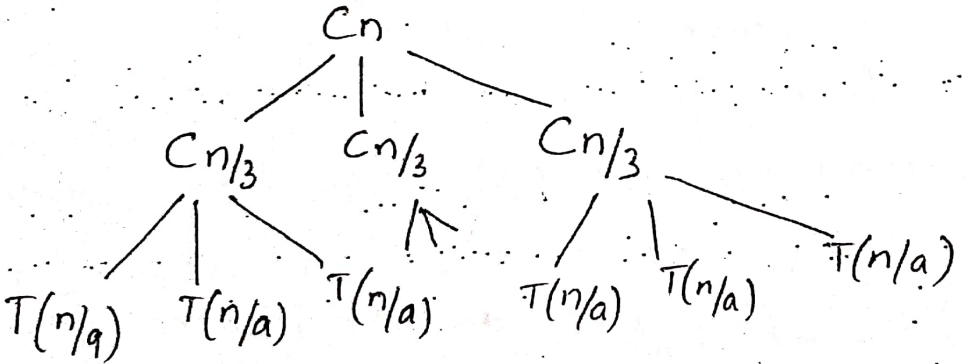
Recursive Tree Method.

$T(n) = 3T(n/3) + Cn$ (C is some constant)

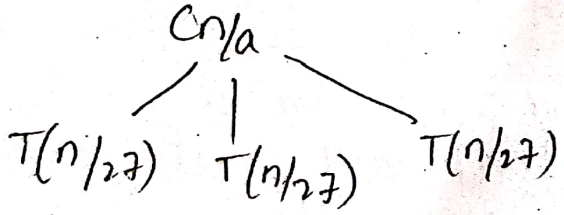
We start building the recursion tree from a single node and that single node is $T(n)$. What we will do in each step is we will replace $T(n)$ or $T(n)$ divided by some no. b, we will replace these T's by their corresponding values as given by right hand side of recurrence relation diagramatically. So $T(n)$ is replaced by Cn with 3 children $T(n/3)$.



Note that value of $T(n) = Cn + T(n/3) + T(n/3) + T(n/3)$ we can do the same by further expanding the tree by one more level. So what is $T(n/3)$ ↓.



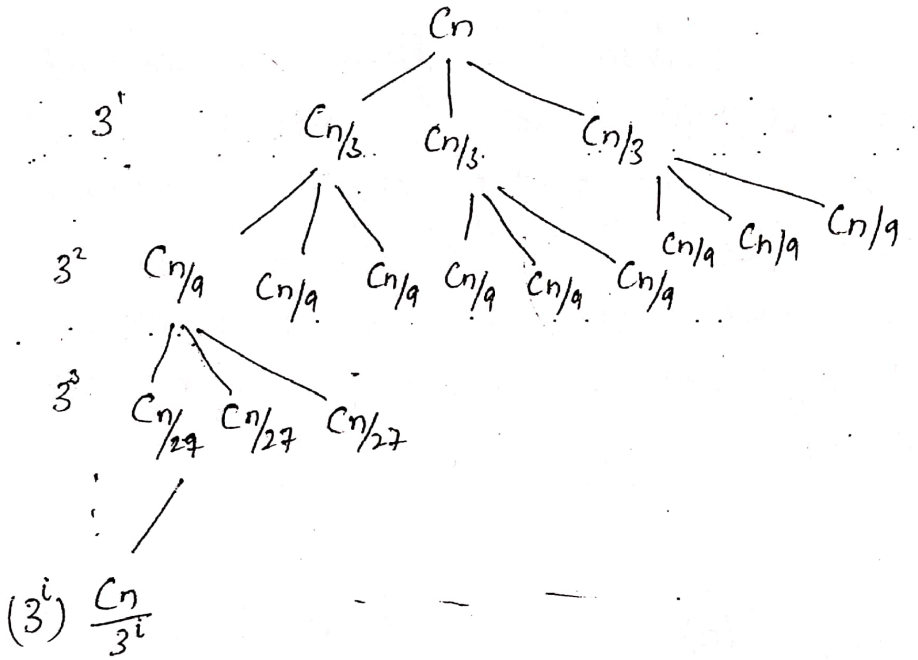
In the next iteration of recursion tree method, each of the $T(n/a)$ will be replace by a sub-tree.



and by that we will create another level in recursion tree

So we keep on expanding, until we hit a sub-problem of size 1.

level 0 # nodes = 1



level Seems Cn

$3 \cdot \left(\frac{Cn}{3}\right) = Cn$

$9\left(\frac{3n}{9}\right) = Cn$

$(3)^3\left(\frac{Cn}{3^3}\right) = Cn$

$3^i\left(\frac{Cn}{3^i}\right) = Cn$

$3^{n-1}\left(\frac{Cn}{3^{n-1}}\right) = Cn$

---

h T(1) T(1) T(1) T(1) - - - - -

↓ no. of nodes 3^h $3^h \times T(1)$

at i^{th} level, we will have sub-problem of size since at level 0; we have problem size c times n, at level 1, we have c times n/3, at level 2, we have C times n/3², level 3, c time $n/(3)^3$, so at i $\left[\frac{Cn}{3^i}\right]$ value of node & how many nodes at each level.

If call bottom most level as h (ht of tree)

T(1) = constant

no of nodes at $h = 3^h$.

→ How do evaluate value of T(n), we do so by summing up the cost at each node at all levels in tree. sum up in level by level.

so what is h in terms of n

size of sub-problem is reducing by a factor 3 until size of sub-problem reduced by 1.

$$\frac{n}{3^h} = 1 \Rightarrow n = 3^h$$
$$h = \log_3 n$$

So, we can write an expression for $T(n)$

$$T(n) = 3^h \times T(1) + Cn + Cn + Cn \cdots n-1$$

$$T(n) = 3^h \times T(1) + \sum_{i=0}^{h-1} Cn$$

$$T(n) = 3^{\log_3 n} T(1) + \underbrace{\sum_{i=0}^{h-1} Cn}_{Cn \times n \rightarrow Cn \log_3 n}$$

$$T(n) = nT(1) + Cn \log_3 n \longrightarrow \text{dominant term}$$

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

Note

$$T(n) = 3^h \times T(1) + \sum_{i=0}^{h-1} Cn$$

appears as pattern as $T(n)$ is written as sum of 2 expressions 1st → no. of nodes at level h + a series having h term & added thing is level sum.

We can get a G.P term

$$T(n/3) = 3T(n/9) + Cn$$
$$T(n/3) = 3T(n/3) + Cn$$

STRASSEN'S MATRIX MULTIPLICATION

Gunjan

The time complexity of calculating the matrix product $C = AB$ where A, B and C are $n \times n$ matrices, using traditional matrix multiplication, is $O(n^3)$.

Strassen's Matrix multiplication algo. is able to perform this calculation in time $O(n^{2.81})$.

Traditional Matrix Multiplication

C_{ij} = dot product of ith row in A with j^{th} column in B.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Algo: Multi (A, B, C)

1. for $i = 1$ to n ⟶ n times
2. for $j = 1$ to n ⟶ n times
3. C[i][j] = 0
4. For k = 1 to n ⟶ n times
5. c[i][j] = C[i][j] + A[i][k] x B[k][j]

Complexity = $O(n^3)$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

If A & B are 2 matrices.

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$
$C_{12} = A_{11} B_{12} + A_{12} B_{22}$
$C_{21} = A_{21} B_{11} + A_{22} B_{21}$
$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

= 8 multiplication along with 4 addition operation.

The divide-and-conquer strategy suggests another way to compute the product of $n \times n$ matrices.

for simplicity we assume that n is a power of 2, $n = 2^k$.

In case n is not a power of 2, then enough rows & columns of zeros can be added to both A and B so that the resulting dimensions are a power of two.

Imagine that A and B are each partitioned into 4 square matrices, each submatrix having dimensions $n/2 \times n/2$.

Then the product AB can be computed.

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

If $n = 2$, then C is computed using a multiplication operation for elements of A & B.

for $n > 2$, the elements of C can be computed using matrix multiplication & addition operation applied to matrices of size $n/2 \times n/2$.

AB → 8 multipl^n of $\frac{n}{2} \times \frac{n}{2}$ matrices & 4 additions of $\frac{n}{2} \times \frac{n}{2}$.

Since 2 $\frac{n}{2} \times \frac{n}{2}$ matrices can be added in time cn^2 for some consth. c, overall computing time $T(n)$ will be.

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(\frac{n}{2}) + cn^2 & n > 2 \end{cases}$$

b, c = consth.

$$\boxed{T(n) = O(n^3)}$$

Since matrix multiplications are more expensive than matrix additions $[O(n^3)$ v/s $O(n^2)]$.

Volker Strassen has discovered a way to compute the C_{ij} using only 7 multiplications & 18 additions or subtractions.

His method involves

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + cn^2 & n > 2 \end{cases}$$

b, c are constt.

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

eg: $A = \begin{bmatrix} 3 & 2 \\ 4 & 8 \end{bmatrix}$ $B = \begin{bmatrix} 1 & 5 \\ 9 & 6 \end{bmatrix}$

$A_{11} = 3$ $B_{11} = 1$
$A_{12} = 2$ $B_{12} = 5$
$A_{21} = 4$ $B_{21} = 9$
$A_{22} = 8$ $B_{22} = 6$

P $P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = (3+8) \times (1+6) = [77]$

Q $P_2 = (A_{21} + A_{22}) \times B_{11} = (4+8) \times 1 = [12]$

R $P_3 = A_{11} \times (B_{12} - B_{22}) = 3 \times (5-6) = -[3]$

S $P_4 = A_{22} \times (B_{21} - B_{11}) = 8 \times (9-1) = [64]$

T $P_5 = (A_{11} + A_{12}) \times B_{22} = 3*2 \times 6 = [30]$

U $P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = (4-3) \times (1+5) = [6]$

V $P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = (2-8) \times (9+6) = [-90]$

$C_{11} = P_1 + P_4 - P_5 + P_7 = [77] + [64] - [30] + [-90] = 21$

$C_{12} = P_3 + P_5 = [-3] + [30] = 27$

$C_{21} = P_2 + P_4 = [12] + [64] = 76$

$C_{22} = P_1 + P_3 - P_2 + P_6 = [77] + [-3] - [12] + [6] = 68$

$$C = \begin{bmatrix} 21 & 27 \\ 76 & 68 \end{bmatrix}$$