UNIT → 1

INTRODUCTION

Module 1

1/oct/2021

→ What is an Algorithm ? what are the characteristics of an Algorithm?

① Algorithm is a step by step method to solve a problem

② It is a sequence of well defined steps to solved in a finite time.

③ A Algorithm is the sequence of unambiguous instructions for solving a problem for obtaining a required output for any legitimate input in a finite amount of time is called algorithm.

④ The branch of study of algorithms is algorithmics . In computer scienc the role of algorithms is very important.

Algorithm + DS ⇒ Program

Note - Algorithm must satisfies the following criteria.

Input ⇒ Given Values

Output ⇒ produce.

⑤ An algorithm has a well defined sequence of steps, it gives you an output and it well eventually terminate

⑥ An algorithm is a seq contained step by step of operations to be performed to solve a specific problem or a class of problems.

For example- an algorithm for driving to a friend's house could be

step 1 - find our keys.
step 2 - walk out of the house
step 3 - close the door.
step 4 - open the car door.
step 5 - Get into the car.
step 6 - Put the key into the ignition etc.

* In terms of computer program is a sequence of instructions that comply the rules of a specific programming language written to perform a specified task with a compiler.

CHARACTERISTICS OF AN ALGORITHM

(1) Input ⇒ Algorithm must obtain /contain '0' or more input

(2) Output ⇒ Algorithm must contain '1' or more output

(3) Finiteness ⇒ Algorithm must complete after finite no of steps.

(4) Definiteness ⇒ The step of the algorithm must be defined precisely or clearly unambigous.

(5) Effectiveness ⇒ The step of an algorithm must be effective (can be done sucessfully).

Time complexity

(Factorial No)

Non- Recursive program

```
int fact (int n)
{
1   int i, j ;
2   i=1
3   for j= 2; j<=n; j++)
4       i= i* j ;
5   return i ;
    }
```

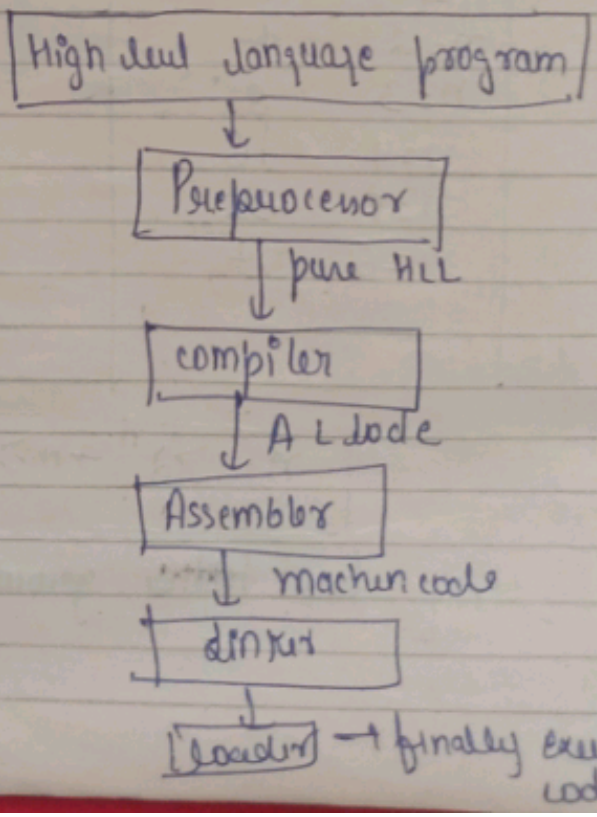
Measurements -

1. No of lines of code
2. easy to implement / understand
3. Time to execute
4. space to execute
5. no of statements

(Factorial No)

Recursive program

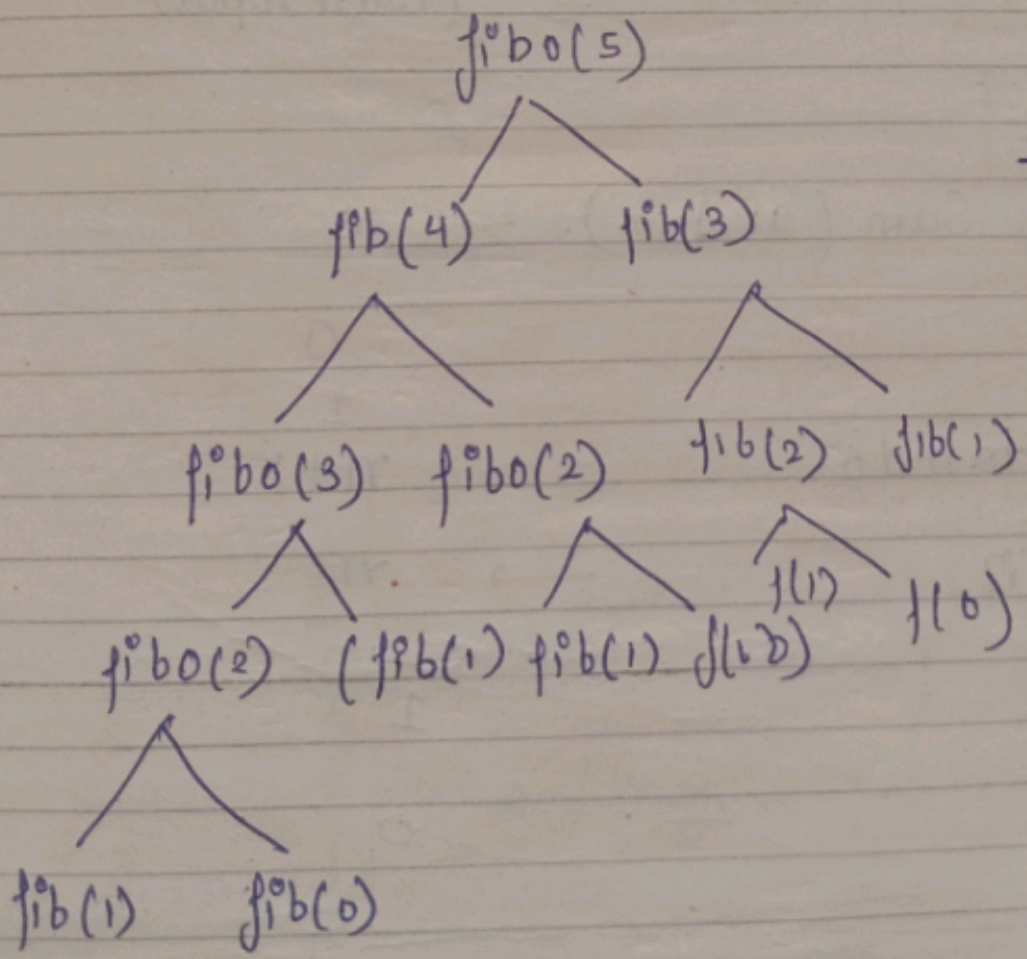
```
Int fact (int n)
{
1  if (n<=1) return 1 ;
2  Else return n* fact(n-1)
   }
```



Note - preprocessor #define
typedef int ll;
remove this line

Another Example - Fibonacci Number

```
int fibo(int n)
{
if(n==0) return 0;
if(n==1) return 1;
return fibo(n-1)+fibo(n-2);
```



fibo(5) = 1
fibo(4) = 1
fibo(3) = 2
fibo(2) = 3
fibo(1) = 5
fib(0) = 3

15

Module 2

It is defined as the number of steps required to solve the entire problem using some efficient algorithm

$$T(P) = C + t_{(P)}(\text{instance})$$

↳ constant

where

$T(P)$ = Time complexity of program P.
C = compile time → fixed part
$t_{(P)}$ = run time (instance) (variable part)

for ex- add^n of two numbers
but add^n of two matrix, (more input)

for example-

```
Algorithm Sum (a[], n)        -    0
{                             -    0
    s = 0;                    -    1
for  i = 1 to n do            -    n+1
    s = s + a[i];             ⟶    n
return s;                     -    1
}                             -    0
```

time complexity = $2n+3$

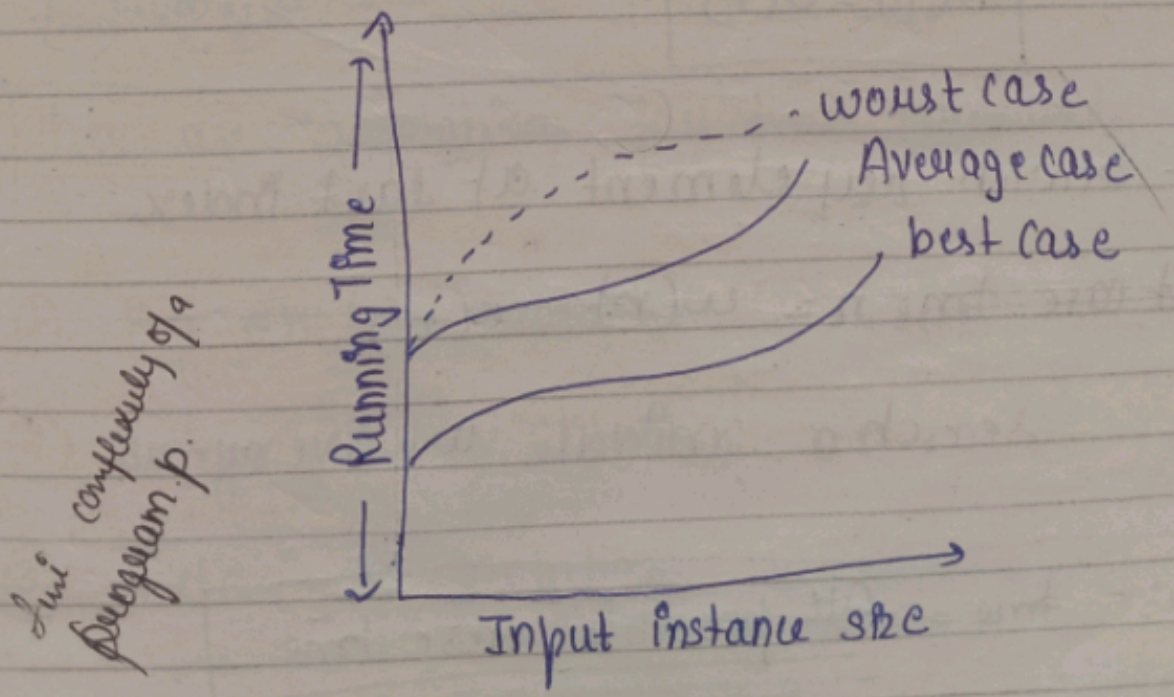
Another example

```
Sum(a[][],n,m) - 0
{                          - 0
S = 0;                     - 1

for i=1 to n do            - n+1
for j=1 to m do =          n(m+1)
S = S+a[i][j];             n x m

return s;                  1
}
```

Time complexity.

$2n + 2n \times m + 3$ Answer

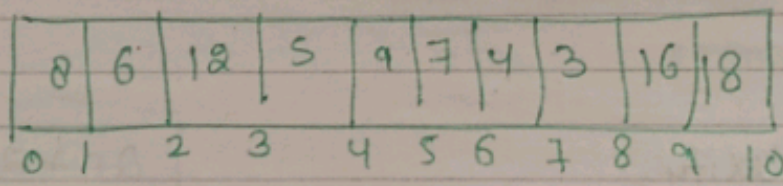
Graph on types of Analysis



① Best case ⇒ 1) Takes smallest time
2) Its function which peyorms the minmum no of steps on input data of n element

② Average case ⇒ Take average running time.
* The function which perform average no of steps on input data of n elements

③ worst case ⇒ * It takes maximum possible time
* It is the function which perform the maximum no of steps on input data of n element



linear search

Best case - Search key element present at first index

$$B(n) = O(1)$$

worst case - Search key element at last index

worst case time $n = w(n) = O(n)$

Average case - search a middle or average no

Average case time = $\frac{\text{All possible case time}}{\text{no of cases}}$

$$\Rightarrow \frac{1+2+3+---n}{n} \rightarrow \text{formula}$$

$$\Rightarrow \frac{n(n+1)/2}{n} \rightarrow \text{formula}$$

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

Note - Arrange the following according to the increasing order of growth rate

$n, 2^n, n\log n, n!, n^3, n^5, n^2, 1$

solution - $1, n, n\log n, n^2, n^3, n^5, 2^n, n!$

constant $\leq$ logarithm $\leq$ polynomial $\leq$ exponential $\leq$ factorial

There are 3 notations for time complexity

1. Big oh (O) notation
2. Omega (Ω) notation
3. Theta (Θ) notation.

Module 3

Asymptotic Notation

1 Big oh (O) notation

Let $f(n)$ & $g(n)$ are two functions

$f(n) = Og(n)$ [reaaded as $f(n)$ is big oh of $g(n)$]

when $\boxed{f(n) \leq c \cdot g(n)}$ where c is a constant

for example

we say that it requires max^m 1 hr from cuttack to bhubaneshwar.

② Omega (Ω) notation

Let $f(n)$ & $g(n)$ are two functions

$f(n) = \Omega g(n)$ [read as $f(n)$ is omega Ω of $g(n)$]

where $\boxed{f(n) \geq c \cdot g(n)}$

Here C is a constant

for example - we say that it require min 30 minuts from cuttack to bhubaneshwar.

③ Theta (Θ) notation

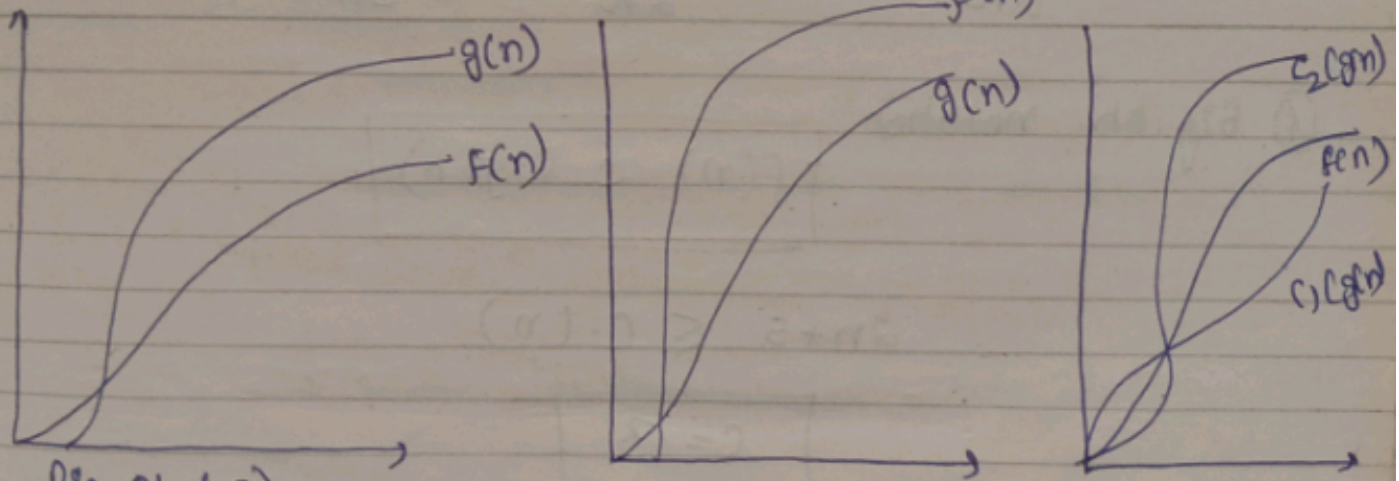
Let f(n) and g(n) are two functions

$f(n) = \Theta(g(n))$ [read as f(n) is theta of g(n)]

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

Here C_1 and C_2 are constants

for example we say that it require approximately 40 minutes from cuttack to bhunaheswar



Big oh (O)

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

omega (Ω)

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

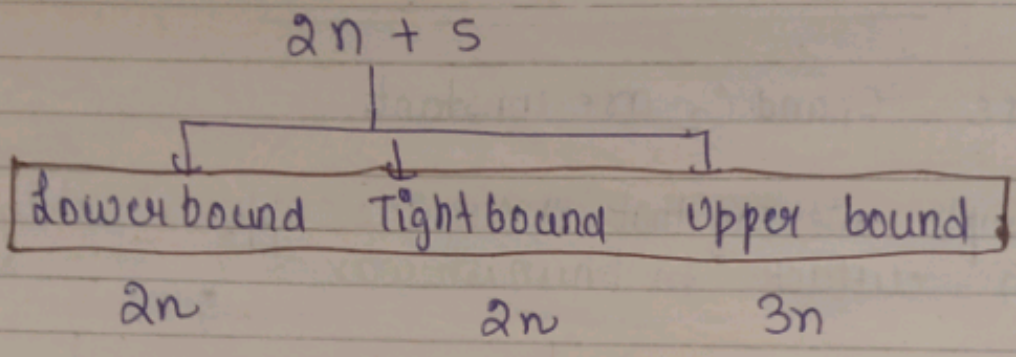
Theta Θ

Big oh (O) - worst case - upper bound) (UB)
omega (Ω) - best case - lower bound (LB)
Theta (Θ) Average case - tight bound (TB)

Ques Find upper bound, lower bound, tight bound
Range for the following function.

$2n+5$

consider $f(n) = 2n+5$ (n degree $= 1$)
$g(n) = n$)



① Big oh notation

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

$$2n+5 \leq c \cdot (n)$$

$$c = 3$$

$$2n+5 \leq 3 \times n$$

for $n = 1$

$7 \leq 3$ false

put $n = 2$

$9 \leq 6$ false

for $n=3$ $11 \leq 9$ false
for $n=4$ $13 \leq 12$ false
for $n=5$ $15 \leq 15$ True

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

$2n+5 = O(n)$ for all $n \geq 5$ & $c=3$

② Omega (Ω) Notation

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

$2n+5 \geq c \cdot (n)$

$c = 2$

$2n+5 \geq 2 \cdot n$

for $n=1$
$9 \geq 2$ true

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

$2n+5 = \Omega(n)$ for all $n \geq 1$ & $c=2$

③ Theta (Θ) notation

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$$C_1 \cdot n \leq 2n+5 \leq C_2 \cdot n$$

$$2n \leq 2n+5 \leq 3n(n)$$

for $n=1$

$2 \leq 7 \leq 3$ false

for $n=2$

$4 \leq 11 \leq 6$ false

for $n=3$

$6 \leq 11 \leq 9$ false

for $n=4$ $8 \leq 13 \leq 12$ F

for $n=5$

$10 \leq 15 \leq 15$ true

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

$\boxed{2n+5 = \theta(n)}$ for all $n \geq 5$, $c_1 = 2$, $c_2 = 3$

Ques - prove that $5n+3 = O(n)$

Ans - $f(n) = 5n+3$

$5n+3 \leq 6 \cdot n$

$\boxed{5n+3 = O(n)}$

Ques prove that $5n+3 = \Theta(n)$

Ques - Prove that $6n^2 + 2n + 3 = \Omega(n^2)$

Ques - prove that $n + 6n + 3$.

Ques - prove that $n^3 + 2n^2 + 3$.

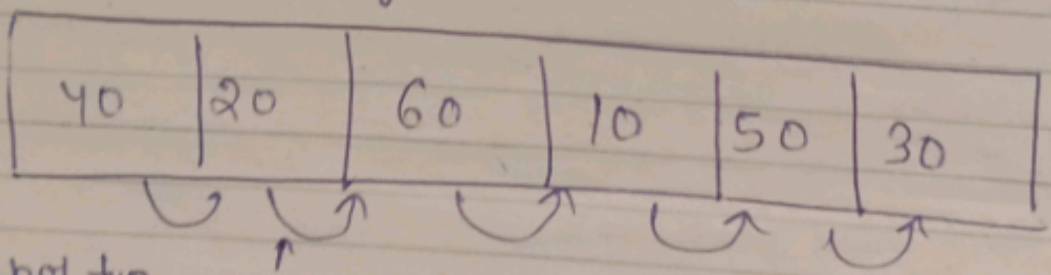
→ compare two elements for example- card. →
comparision between Insertion, mergesort
→ Quick sort and heap sort complexity) →

Insertion Sort (Time complexity)

① - INSERTION SORT (A)

1) for j = 2 to A.length
2) key = A[j]
3) // insert A[j] the sorted sequence A[1---j-1]
4) i = j-1
5) while i > 0 and A[i] > key
6) A[i+1] = A[i]
7) i = i-1
8) A[i+1] = key

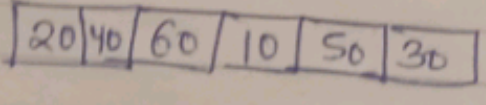
i > 0 ||
4 > 2



swap of two compare

i

Step2



1 > 0
means - 40 > 20

Step6 means A[i+1] = A[i]

Best case

10	20	30	40	50	60

comparision

swap

comparision

swap

0

0

1

(n-1)

1

1

1

O(n) upper bound ki value hoti swapping O(1) hai

O(n)

O(1)

worst case

60	50	40	30	20	10

~~to 2~~

comparision

swap

0

0

1

1

2

2

3

3

(n-1)

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

highest degree

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

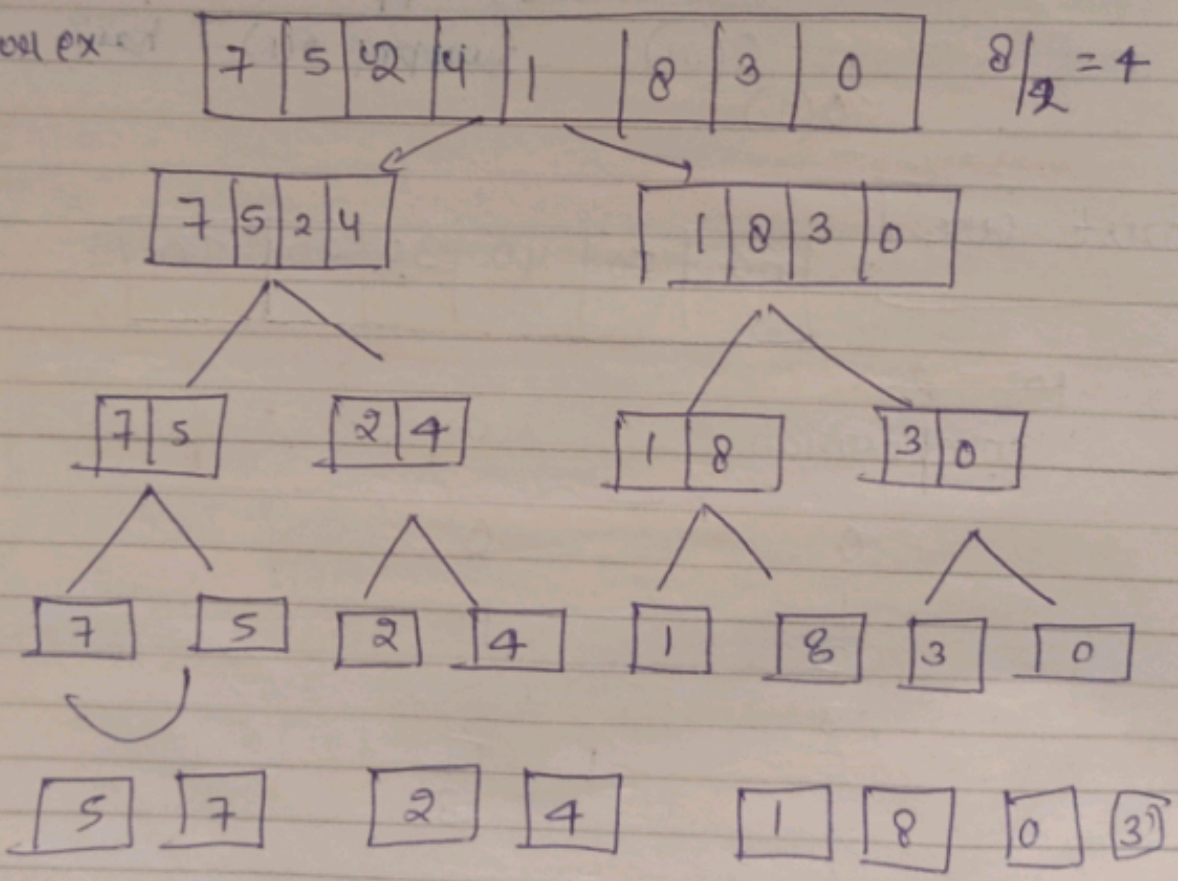
Worst case performance - $O(n^2)$ comparision and swaps.

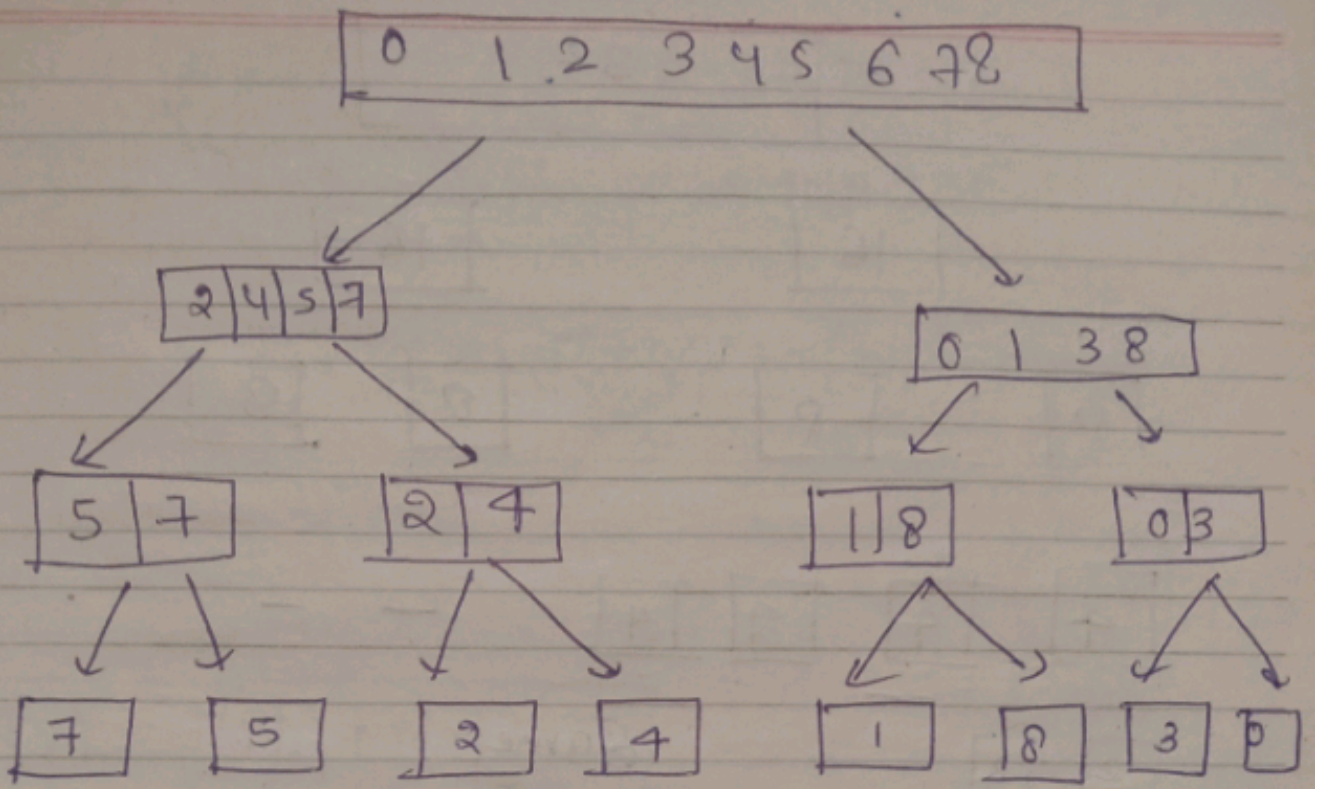
Best case performance - $O(n)$ comparision, $O(1)$ swaps.

Average performance - $O(n^2)$ comparision and swaps

Merge sort Example

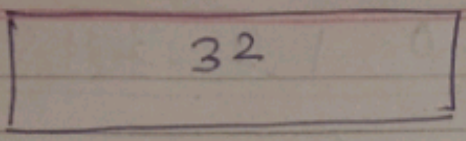
for ex.

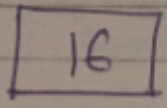


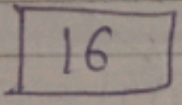


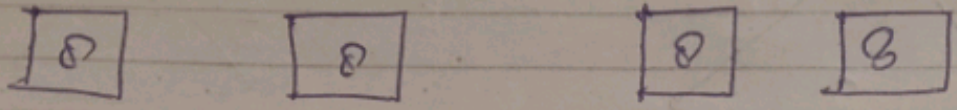
Algorithm Merge sort

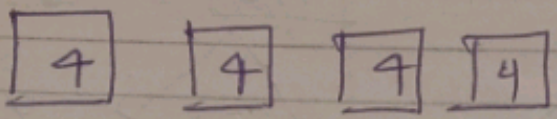
Step 1 Merge sort (array A, int p, int r)
Step 2 If ($p < r$)
Step 3 then
Step 4 $q \leftarrow (p+r)/2$
Step 5 Merge - sort (A, p, q) // sort A[p---q]
Step 6 Merge sort (A, q+1, r) // sort A[q+1---r]
Step 7 Merge sort (A, p, q, r)



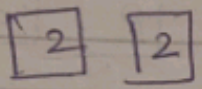




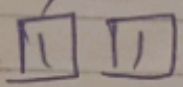




— —

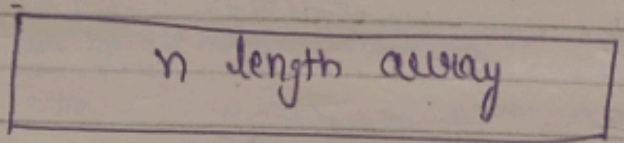


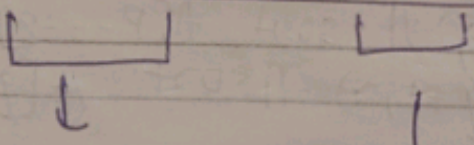
same



finding Merge sort complexity

a) Assume we have n length array





$n/2$ $n/2$

$T(n/2)$ $T(n/2)$

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

Note 1 -

$T(1) = 1$, its base case

If value is greater than 1 then we put it in following equation

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

for $n = 2$

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

put value of n in equation

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

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

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

$$T(2) = 4$$

for $n = 4$ $\quad T(n) = T(n/2) + T(n/2) + n$

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

$T(4) = 4 + 4 + 4$

$T(4) = 12$

$T(1) = 1$

$T(2) = 4$

$T(4) = 12$

$T(8) = 32$

$T(16) = 80$

There is no i/p and o/p pattern in above results if we divide both sides by i/p

$T(1)/1 = 1/1 = 1$:-

$T(2)/2 = 4/2 = 2$

$T(4)/4 = 12/4 = 3$

$T(8)/8 = 32/8 = 4$

$T(16)/16 = 80/16 = 5$

if we take $\log_2$ of input then

$T(1)/1 = 1$ $\quad \log_2 1 = 0+1 = 1$

$T(2)/2 = 2$ $\quad \log_2 2 = 1+1 = 2$

$T(4)/4 = 3$ $\quad \log_2 4 = 2+1 = 3$

$T(8)/8 = 4$ $\quad \log_2 8 = 3+1 = 4$

$T(16)/16 = 5$ $\quad \log_2 16 = 4+1 = 5$

$T(n)/n = \log_2 n +$

mergesort - nlogn

Bubble sort - $O(n^2)$ - AC
$O(n^2)$ WC
$O(n)$ - BC

Selection sort $O(n^2)$ - AC
$O(n^2)$ WC
$O(n^2)$ - BC

Insertion sort - $O(n)$ BC
$O(n^2)$ - AC & BC

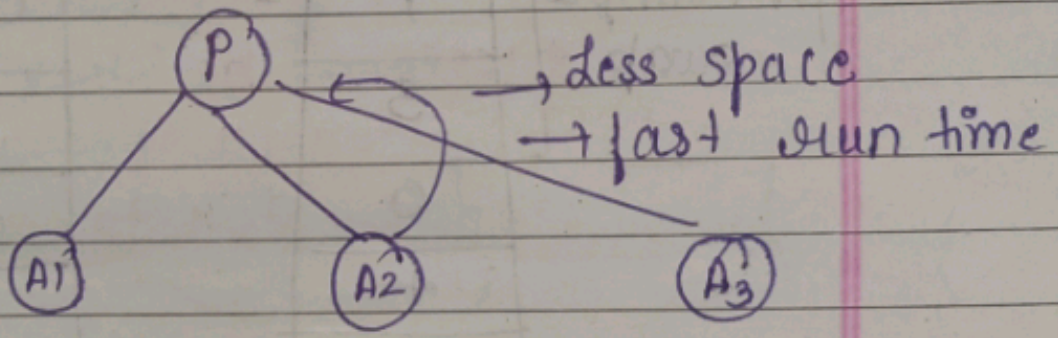
Quicksort - $O(n \log n)$ BC &
WC - (n^2) AC

forex - Real life example - work money

Design and Analysis of Algorithm -

Unit-1 (Lecture 01)

Algorithm - An algorithm is a step by step procedure in order to solve a give problem Generally it is written in english language.

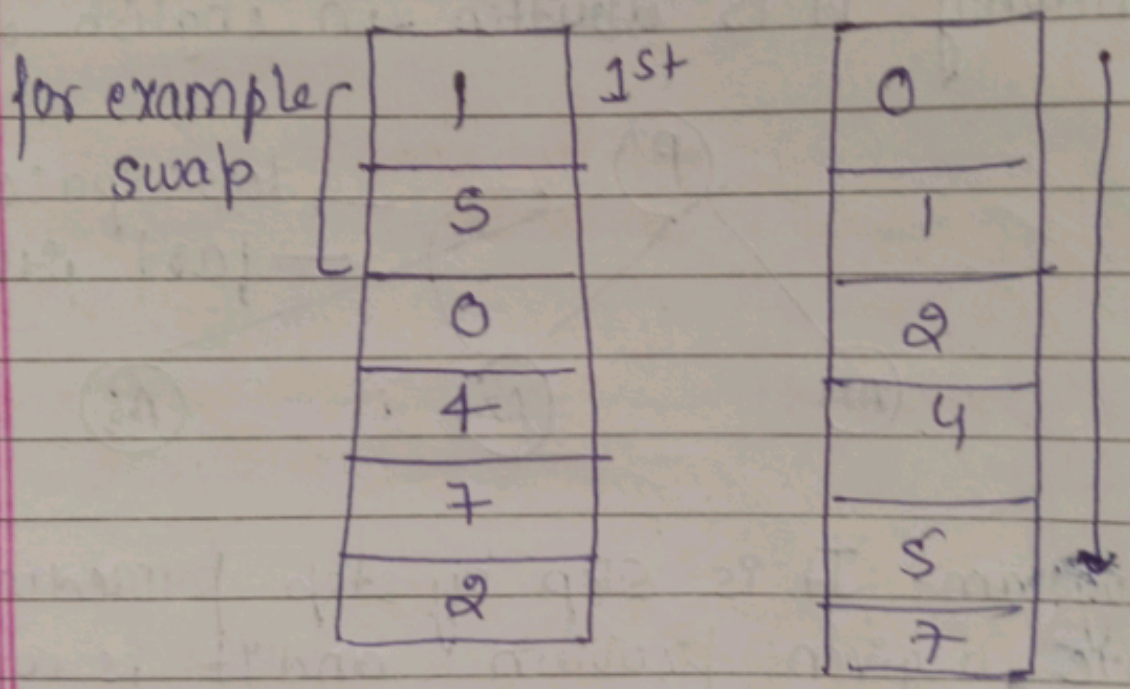


Program - It is step by step procedure to solve a given problem and it is written in the form of any computer language.

Ways of Design an Algorithm

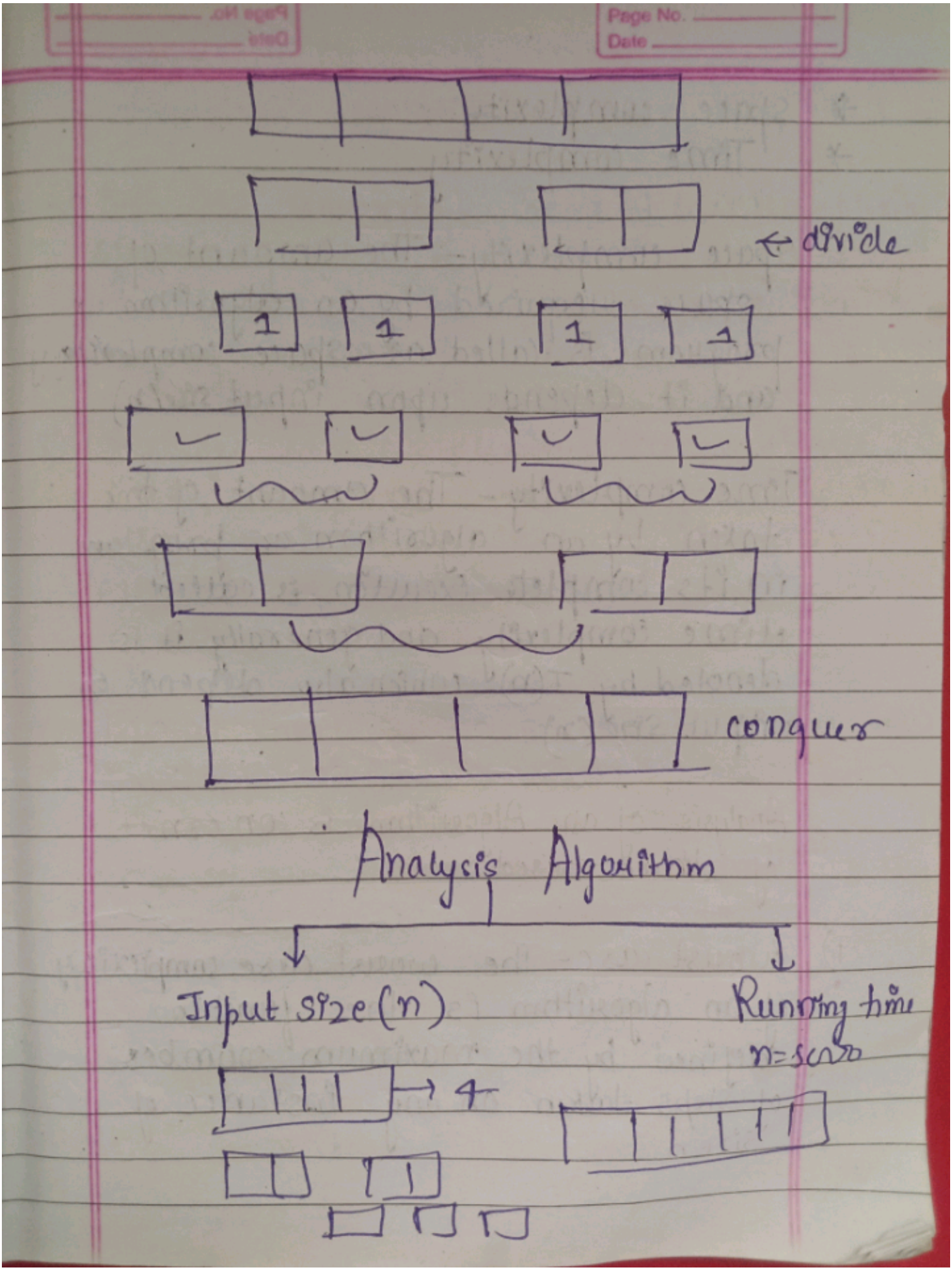
1) Incremental Approach
2) Divide and conquer approach.

→ Incremental approach - In this approach we follow a top down approach in which we proceed from the start and compare elements iteratively and we finally get the solution eg - bubble sort, insertion sort, selection sort etc



Divide and conquer - In this approach, we divide a large problem into sub-problems and each sub problems is solved recursively and combining these solutions we get final solution of the problem.

eg - merge, quick, heap sorts etc


divide
1
1
1
1
conquer
Analysis Algorithm
Input size (n)
Running time
n=scan

* Space complexity
* Time complexity

Space complexity - The amount of space required by an algorithm or program is called as space complexity and it depends upon input size(n)

Time complexity - The amount of time taken by an algorithm or program in its complete execution is called time complexity and generally it is denoted by T(n) which also depends on input size (n)

Analysis of an Algorithm is concerned of three based

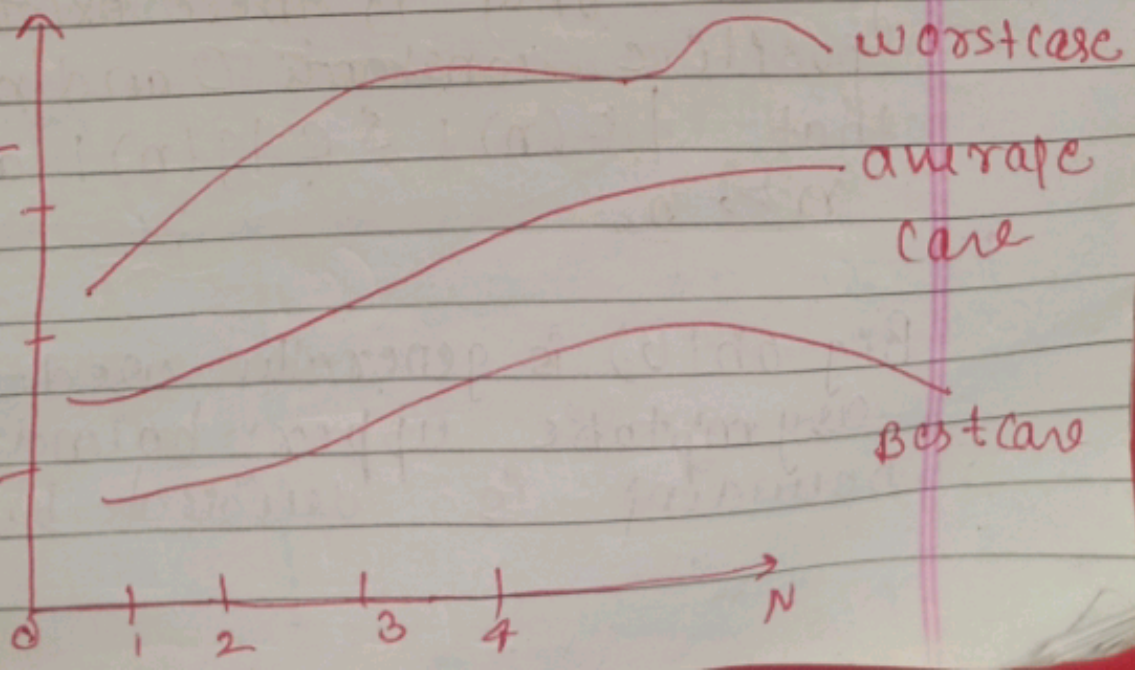
1) worst case - The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.

Asymptotic Notations -

2) Best case complexity - The best case complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.

3) Average case complexity - The average case complexity of an algorithm is the function defined by the average number of steps taken on an instance of size n.

Each of these functions defines a numerical function 1) Time 2) Space



Program step -

	steps per execution
eg - (1) Algorithm Add (a, n)	0
(2) {	0
(3) s = 0;	1
(4) for (i = 1 to n) do	n+1
(5) s = s + a [i]	n
(6) return s;	1
(7) }	0
complexity	2n+3

Asymptotic Notations

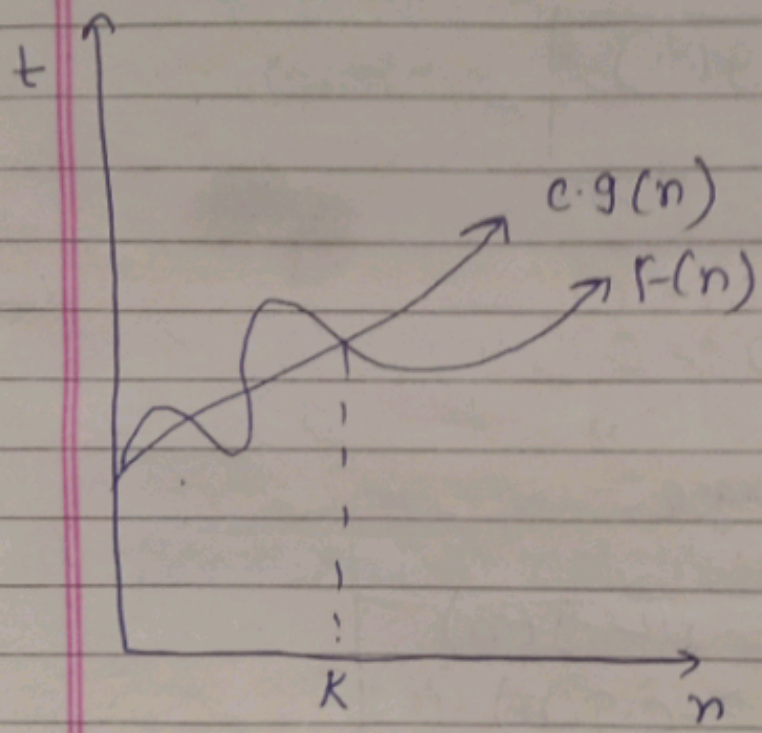
(1) Big oh (O): → $f(n) = O(g(n))$ if and only if there exists two positive constants C and no such that $|f(n)| \leq C|g(n)|$ and $n \geq n_0$

Big oh (O) is generally used to express asymptotic upper bounds A bounding is denoted by the

Note :- $f(n)$ and $g(n)$ are two functions

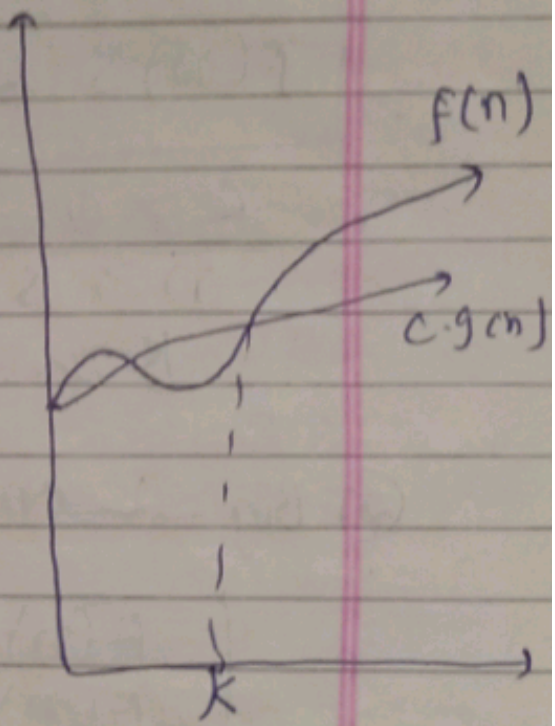
growth of above functions :-

1) Big-oh (O)



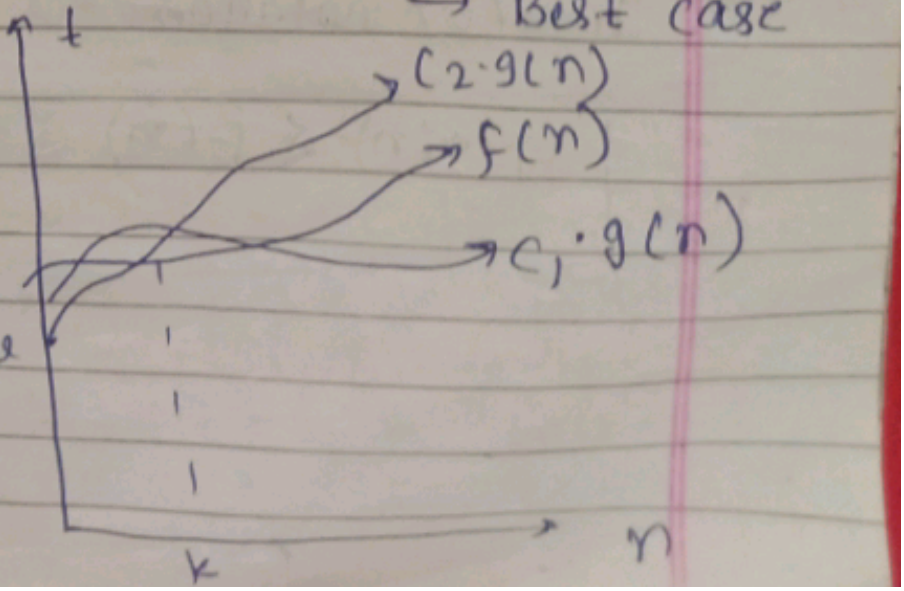
→ upper case (at most)
→ worst case

2) Big-omega (Ω)



→ Lower bound (at least)
→ Best case

3) Theta (Θ)



→ average case,
→ exact time

① Big-oh (O)

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

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

$c > 0$

$n \geq k$

$k \geq 0$

② Big Ω omega

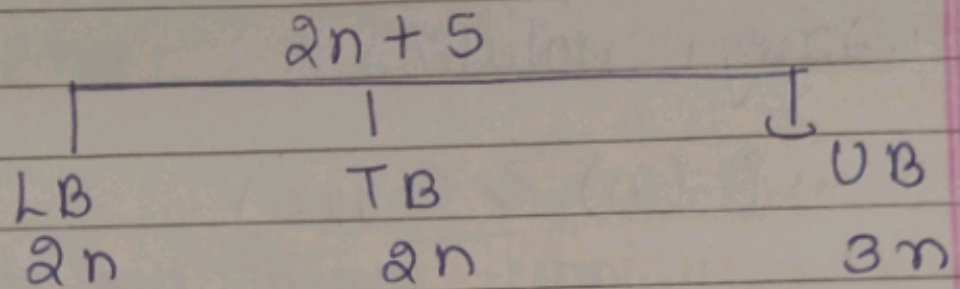
$$f(n) = \Omega \cdot g(n)$$
$$f(n) \geq c \cdot g(n)$$

③ Theta (Θ) notation

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Ques - find upper bound, lower bound and tight range after for the following function.

solution - consider $f(n) = 2n+5$



① Big - oh (O) notation

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

$$2n+5 \leq c \cdot n$$

$$\boxed{c = 3}$$

$$2n+5 \leq 3 \times 1$$

for $n=1$, $7 \leq 3$ - false
for $n=2$, $9 \leq 6$ false
for $n=3$, $11 \leq 9$ false
for $n=4$, $13 \leq 12$ false
for $n=5$ $15 \leq 15$ true

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

$2n+5 = O(n)$

for $n \geq 5$ and $C = 3$

(2) omega notation

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

$2n+5 \geq C \cdot n$

$C = 2$

$2n+5 \geq 2n$

for $n = 1$

$7 \geq 2$ true

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

$2n+5 = \Omega(n)$; $n \geq 1$, $C = 2$

(3) Theta (Θ) notation

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

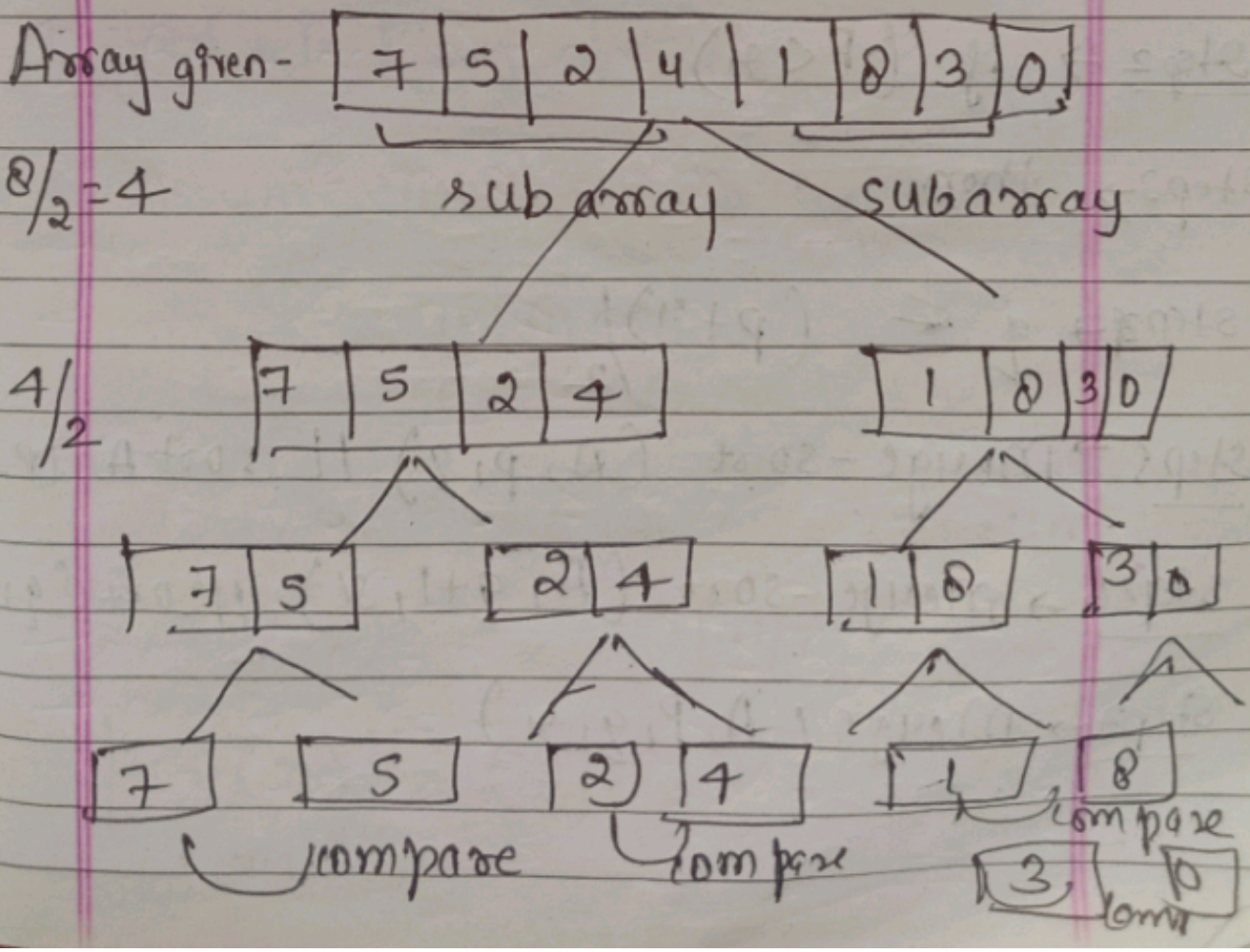
$C_1 \cdot n \leq 2n+5 \leq C_2 \cdot n$

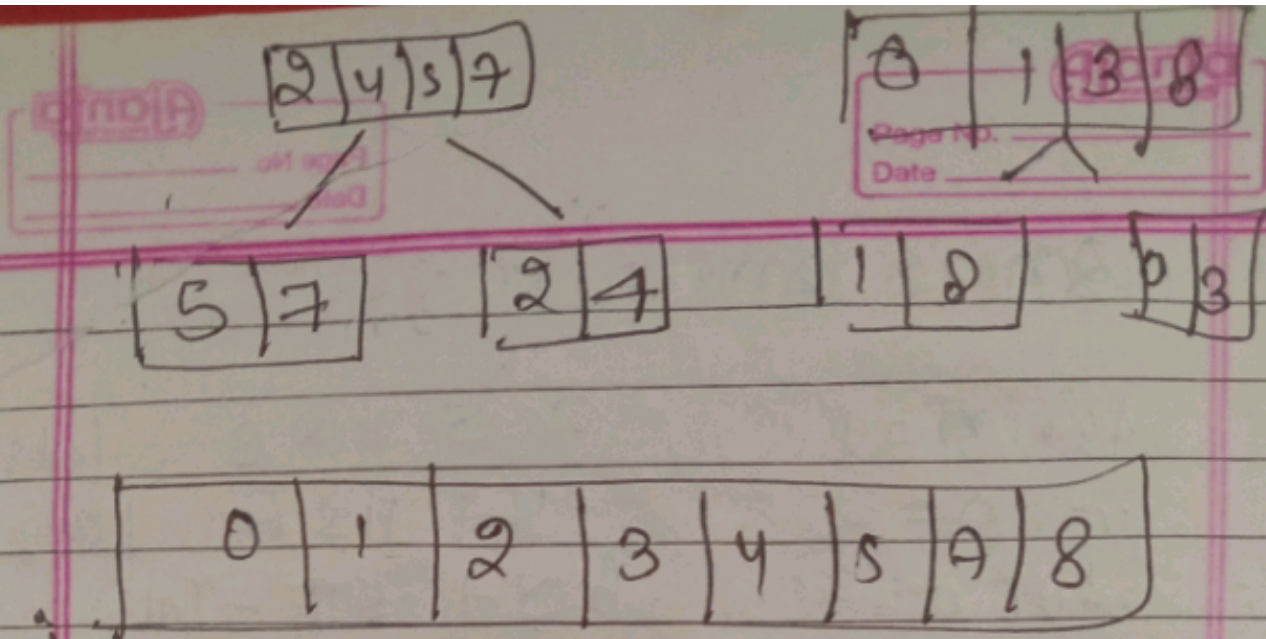
$2n \leq 2n+5 \leq 3n$

for $n=1$, $2 \leq 7 \leq 3$ false
for $n=2$ $4 \leq 11 \leq 6$ false
for $n=3$ $6 \leq 11 \leq 9$ false
for $n=4$, $8 \leq 13 \leq 12$ false
for $n=5$, $10 \leq 15 \leq 15$ true

Ques → Merge sort time complexity -

→ ① Divide and conquer technique.





Sorting procedure

Algorithm

Step1 → Merge sort (array A, int p, int r)

Step2 → if (P < r)

Step3 → then

Step4 → q ← (p+r)/2

Step5 → Merge-sort (A, p, q) // sort A (p-

Step6 → Merge-sort (A, q+1, r) //sort (q+

Step7 → Merge (A, P, q, r)

n/2 (1/2)

for ex - [32] → array element. $32/2 = 16$

[16] [16]

[8] [8] [8] [8]

[4] [4] [4] [4] [4] [4] [4] [4]

[2] [2] [2] [2]

[1] [1] [1] [1]

$\therefore T(1) = 1$ (base case)

• 1) → Assume we have n length array for ex -32

[n length array]

2) $n/2$ $n/2$ (In form of time complexity)

1-2) $T(n/2)$ $T(n/2)$

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

Total complexity

fun^n apne aap se baar baar call kar rahaa ho

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

constant time (assume)

recursive function

$T(1) = 1$ its base case (assume compare)

if value is greater then 1 we put it in the following eqn

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

for $n = 2$

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

put value of n equation no 1

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

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

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

$$T(2) = 4$$

for n = 4

for n = 4

$$T(n) = T(n/2) + T(n/2) + n \quad (1)$$

put value of n in equation (1)

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

$$T(4) = T(2) + T(2) + 4$$

$$T(4) = 4 + 4 + 4$$

$$T(4) = 12$$ Answer

for n = 8

$$T(8) = T(4) + T(4) + 8$$

$$T(8) = 12 + 12 + 8$$

$$T(8) = 12$$

$$12_8$$

$$32$$

Page No.
Date

$T(8) = 32$

for $n = 16$

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

$T(16) = T(8) + T(8) + 16$

$T(16) = 32 + 32 + 16$

$$T(16) = \begin{array}{r} 32 \\ 32 \\ 16 \\ \hline 80 \end{array}$$

$T(16) = 80$

squre ban nahi raha hai

$T(1) = 1$

$T(2) = 4$

$T(4) = 12$

$T(8) = 32$

$T(16) = 80$

$n = 1, 2, 4, 8, 16$

→ There is no i/p and o/p pattern in above results If we divide both side by i/p.

$T(1)/_1 = 1/_1 = 1$ —

$T(2)/_2 = 4/_2 = 2$ — (sequence ban sakte hogaye)

$T(4)/_4= 12/_4 = 3$ —

$T(8)/_8 = 32/_8 = 4$ —

$T(16)/_{16} = 80/_{16} = 5$ —

Input ka log lenge

If we take $\log_2$ of input then $n = 1, 2, 4, 8, 16$

$\log_2 1 = 0 + 1 = 1$

$\log_2 2 = 1 + 1 = 2$

$\log_2 4 = 2 + 1 = 3$

$\log_2 8 = 3 + 1 = 4$

$\log_2 16 = 4 + 1 = 5$

$T(16/16) = 5$

$$T(n)/n = \log_2 n + 1$$

- we only concerned with $T(n)$

multiply both side with n

$$n \cdot T(n)/n = n(\log_2 n + 1)$$

$$T(n) = n(\log_2 n + 1)$$

(Back value)

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

worst case (maximum time)

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

Insertion sort

(1) worst-case performance = $O(n^2)$
comparison and swap

(2) Best-case.

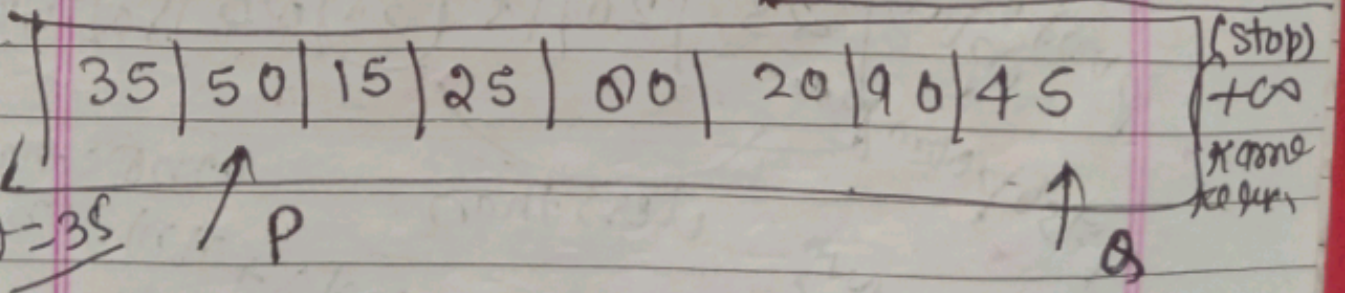
Quick sort

How to calculate time complexity of Quick sort?

→ Quick sort is also divide and conquer approach like merge sort.

(1) P → greater than pivot element

Best case - $n \log n$.
average case - $n \log n$.
worst case -



(2) - q → lesser than the pivot element

worst case

Q → 45 decrement kar
< (less than 45 ...)

v = 35 → pivot

→ both check

35	50	15	25	80	20	90	45

Q

swap kar do

↑ Q

35	20	50	15	25	80	50	90	45

P

stop P Q

cross kar diya, 45 position ... 5 ≥ 35

pivot element replace

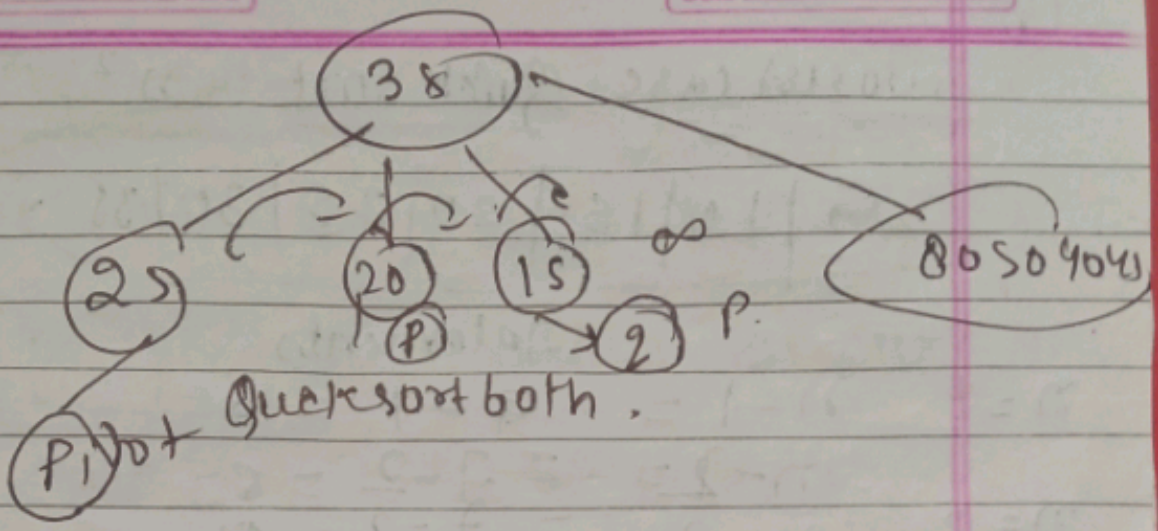
1st step complete

25	20	15	(35)	80	50	90	45

less than — more than

1st pass

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



P > 25

Pivot replaced element

15 20 25 35 80 80 45 90

15 20 25 35 80

15 20 25 35 45 50 80 90

$2T(n/2) + n$

(Best case aproximately pata)

Worst case Quick sort n^2.

5	10	15	20	25	30	35	∞

7 elements

$n=1$ $n-1 = 7-1 = 6$

$n-2 = 7-2 = 5$

$n=2$ $n-3 = 7-3 = 4$

$n-4 = 3$

$n=3$ $n-5 = 2$

$n-6 = 1$

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

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

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

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

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

$$O(n^2)$$

Quick sort

Quick sort -

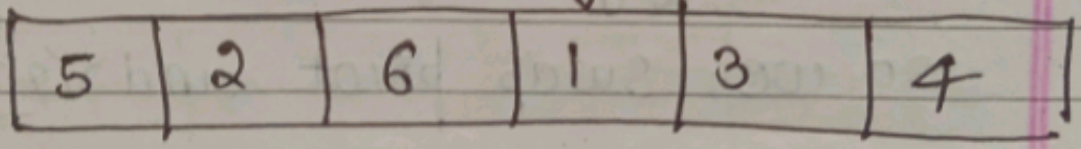
we follow three steps recursively -

a) find pivot that divides the array into two halves.

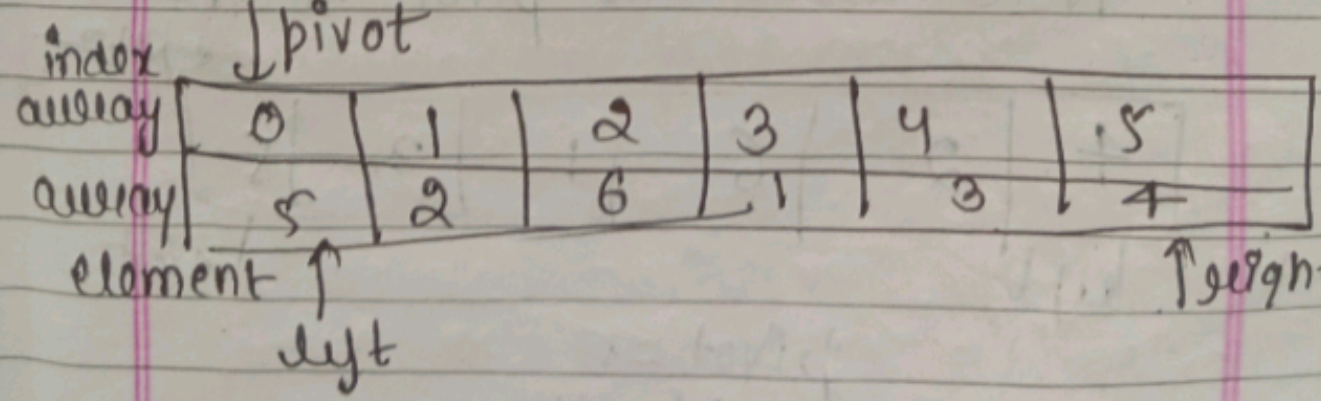
b) Quick sort the left half./halves

c) Quick sort the right half.

consider an array having 6 elements

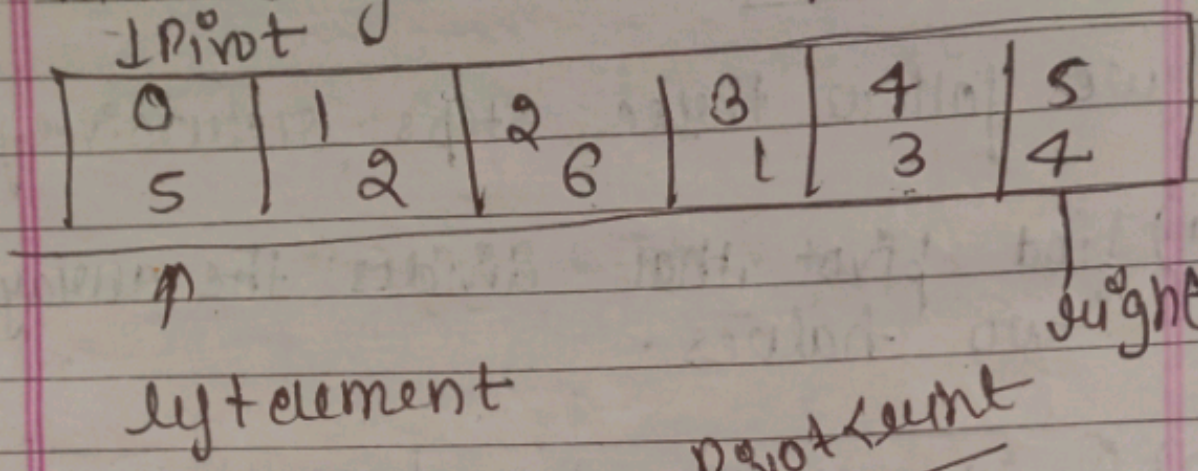


This is our unsorted array



* All elements to the right of pivot

must be greater than pivot.



Pivot < right

P = 5

right = 4

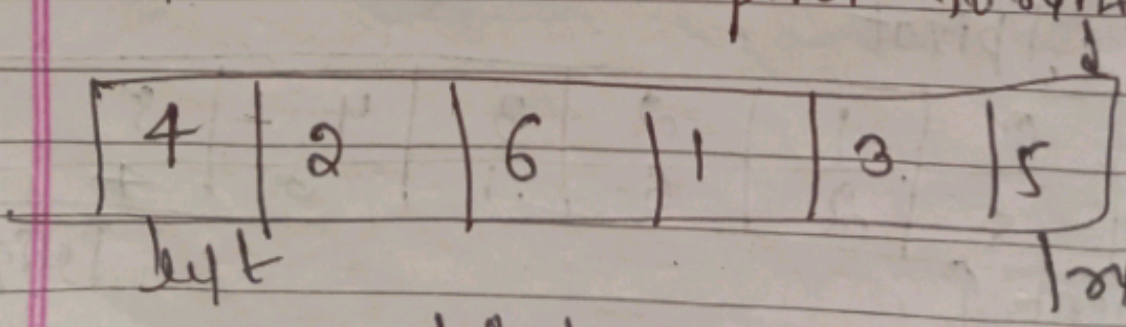
5 < 4

Is pivot < right

NO

so we swap pivot and right

Now move the pivot to mid



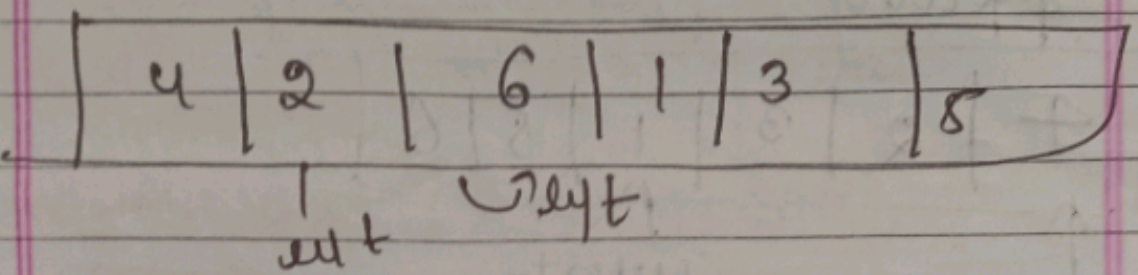
pivot = 5

left = 4

is pivot > left

yes 5 > 4

so we move left one position towards right

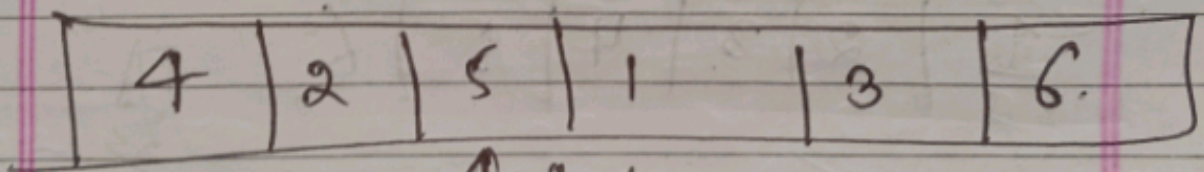


is pivot > left

No

so we swap pivot and left

Now the pivot is pointing at left pivot.

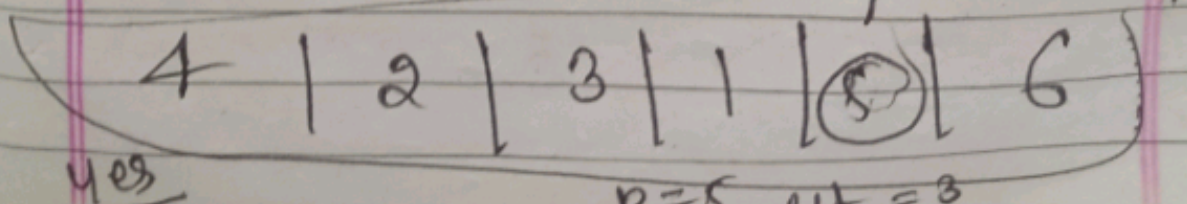


Is pivot < right

5 < 3

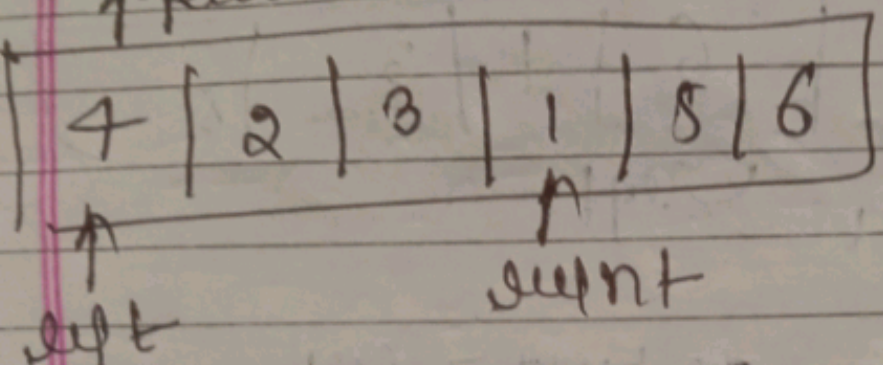
no.

so we swap pivot

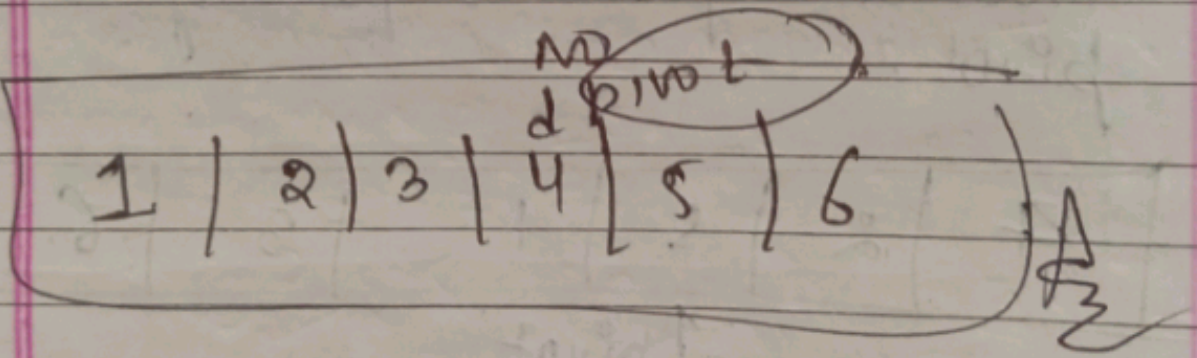


yes

elements left of pivot are smaller



pivot < right
so we swap
4 & 1



this means 4 is the pivot and

→ Quick sort follows recursive algorithm
→ we find the pivot which divides the array into two halves
→

Algorithm of Quick sort.

beg = first index of array
end = last index of array

quicksort (a, beg, end)
Begin
if (beg < end) then

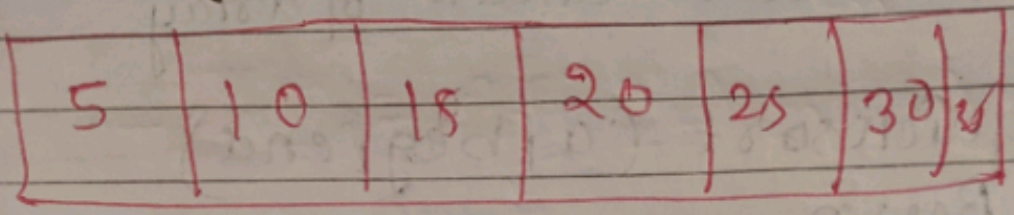
call PartitionArray (a, beg, end, pivotloc);

call Quicksort (a, beg, pivotloc-1);
quicksort left subarray)

call Quicksort (a, pivot loc+1, end);

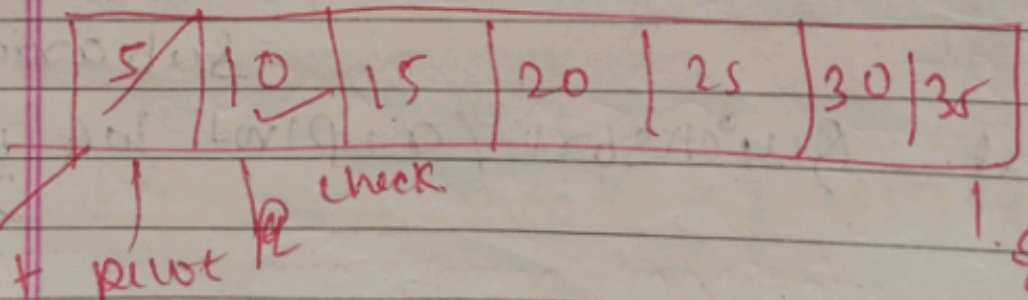
for example - worst case complexity.

Ascending order

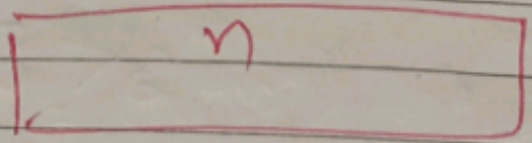


1) suppose we have sorted array but algorithm do not know it.

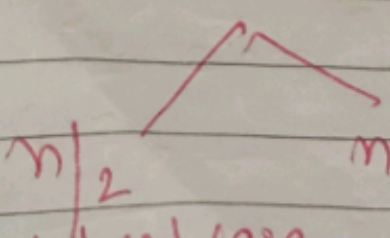
2) Algorithm have to do its complete procedure to check it out.



Exit main to know?



n-1
or n-2



average and best case

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

Substitution method to solve

$n-1 = 7-1 = 6$

$n-2 = 7-2 = 5$

$n-3 = 7-3 = 4$

$n-4 = 7-4 = 3$

$n-5 = 7-5 = 2$

$n-6 = 7-6 = 1$

$n-7 = 7-7 = 0$

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

Substitute n with $n-1$ in equation (1)

$$T(n-1) = T(n-1-1) + (n-1) \quad \text{— (3)}$$

put value of $T(n-1)$ from eq (3) in eq (1)

$$T(n) = T(n-2) + (n-1) + n \quad \text{-- eq (3')}$$

derive the value of $T(n-2)$ from (1)

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

$$T(n-2) = T(n-3) + (n-2) \text{ ---}$$

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

$T(n) = n^2$

Recursion relation solve karne ke liye

Master theorem / master method

$$T(n) = aT(n/b) + \theta(n^k \log^p)$$

$f(n)$

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

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

a and b are positive constants.

formula:

if $f(n) = \theta(n^d)$, $d \geq 0$

There are three conditions formula's-

(1) $T(n) = \theta(n^d)$ if $a < b^d$

(2) $T(n) = \theta(n^d \log n)$ if $a = b$

(3) $T(n) = \theta(n \log_b^a)$ if $a > b^d$

$O(n^d)$ $d \geq 0$

Page No.
Date

Sol $\boxed{T(n) = 4T(n/2) + n}$ ---(1)

$T(n) = aT(n/b) + f(n)$ --- (2)

comparing both eqn (1) and (2)

$\boxed{a = 4 \quad b = 2 \quad f(n) = n}$

$f(n) = n^1$

$\theta(n^1)$

$d = 1$

$4 > 2^1$

$\boxed{a > b^d}$

↳ It is matching with condition (3)

$T(n) = \theta(n^{\log_2 4})$

$\Rightarrow \theta(n^{\log_2 (2)^2})$

$\Rightarrow \theta(n^2)$ Ans

$(a > b^d)$ time complexity of this particular relation.

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

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

→ $T(n) = aT(n/b) + f(n)$ — (1)

$T(n) = 3T(n/2) + n^2$ — (2)

Comparing both these equations

$a = 3$

$b = 2$

$f(n) = n^2$

$f(n) = n^2$

$\Theta = n^2$

$d = 2$

$3 > 2$

$a > b^d$

$3 > 2^2$

$3 > 4$ condition false

$3 < 4$

apply condition (3)

$T(n) = \theta(n^d)$ if $a < b^d$

$3 < 4$ (yes)

$T(n) = \theta(n^d)$

$\boxed{T(n) = \theta(n^2)}$ Ans

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

$a = 2$

$b = 2$ $\quad (d = 1)$

if $a = b$ then apply

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

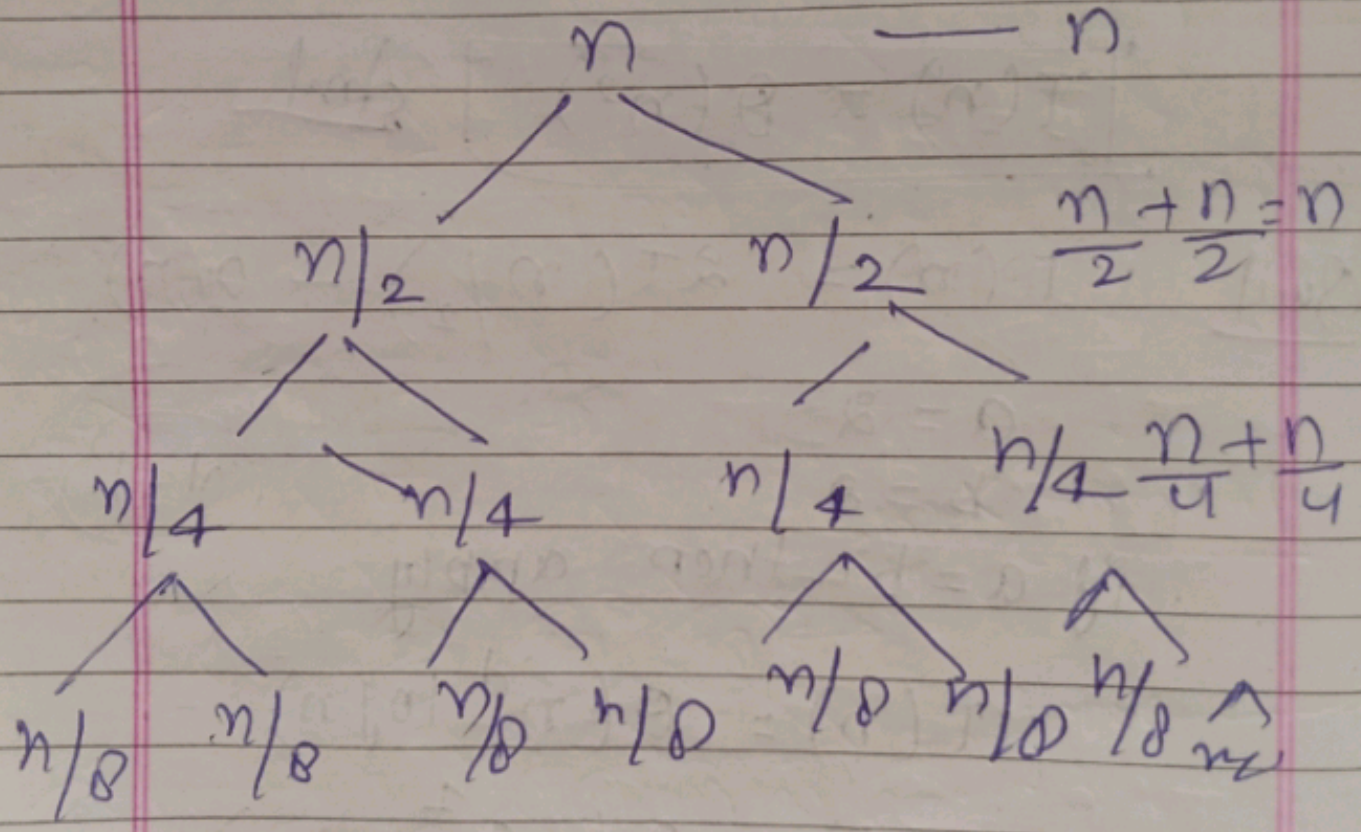
$T(n) = \theta(n1^1 \log n)$

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

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

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

recurrence tree for the above



Page No.
Date

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

$a = 3$

$b = 4$

$f(n) = n$

$n^1 = 1$

$3 < 4$

$T(n) \Rightarrow \Theta(n^d)$

$T(n) = \Theta(n^1)$

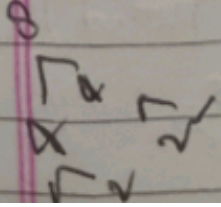
$\boxed{T(n) = \Theta(n)}$

$+\frac{n}{4} \quad +\frac{n}{4}$

$\frac{n}{2} \quad n$

n

Ques $\boxed{T(n) = 2T(n/2) + 3n^2}$



Time complexity = O(n log n)

Max Heap - Min Heap

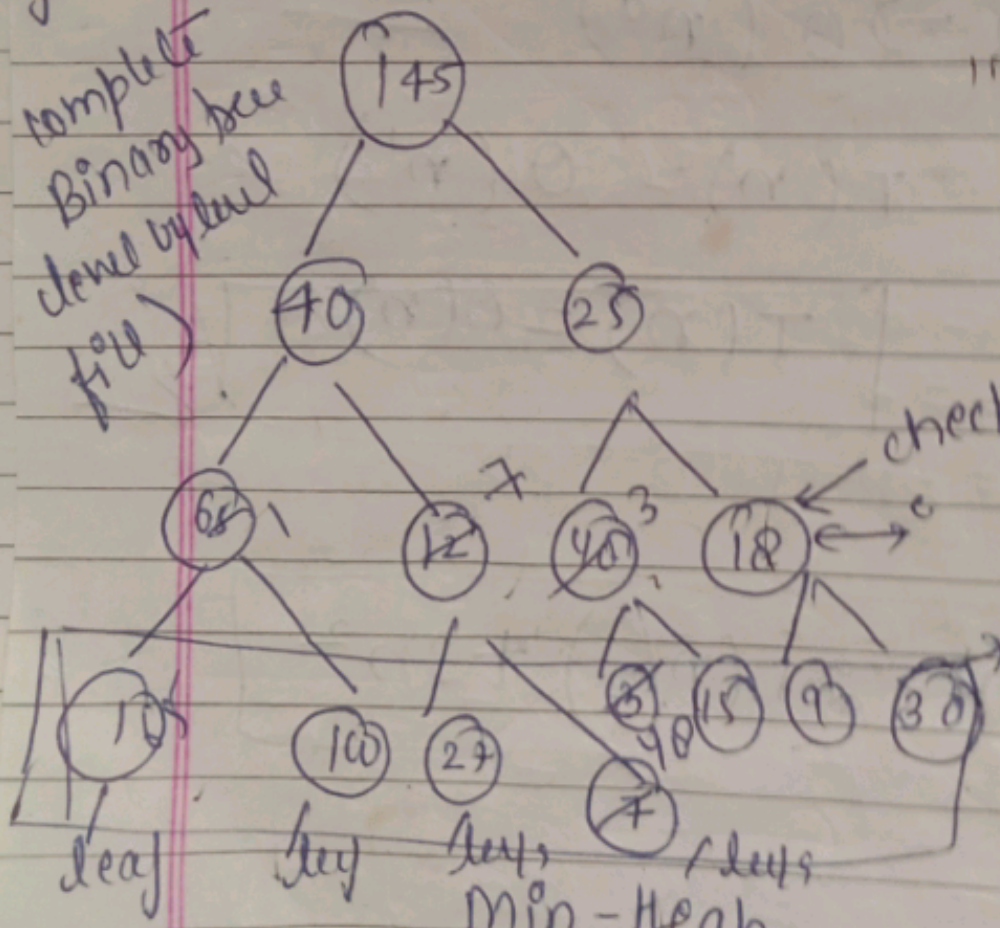
Heap sort. (Heapify method)

How to calculate time complexity of Heap

Build Heap sort ? in O(n)

no of Keys given	145	40	25	65	12	40	18	1 to 10

complete Binary tree (level by level fill)



Min-Heap.

How to convert min-Heap by using Heapify method

min

GP (Common) 1, 2, 3

AP

GP

max heap - min heap

Ajanta

Page No.

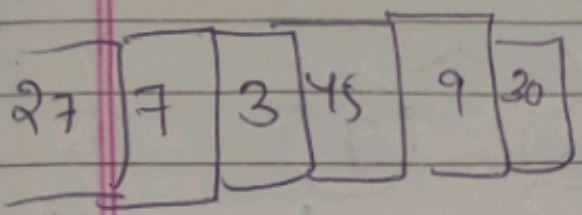
Date

master theorem

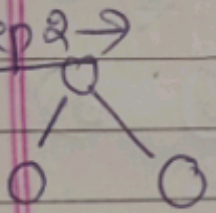
(1) Leaf nodes = 0 (no swapping)

(2) no of elements $\frac{15}{2} = 7.5 = 8 \rightarrow$

leaf (total no of elements)



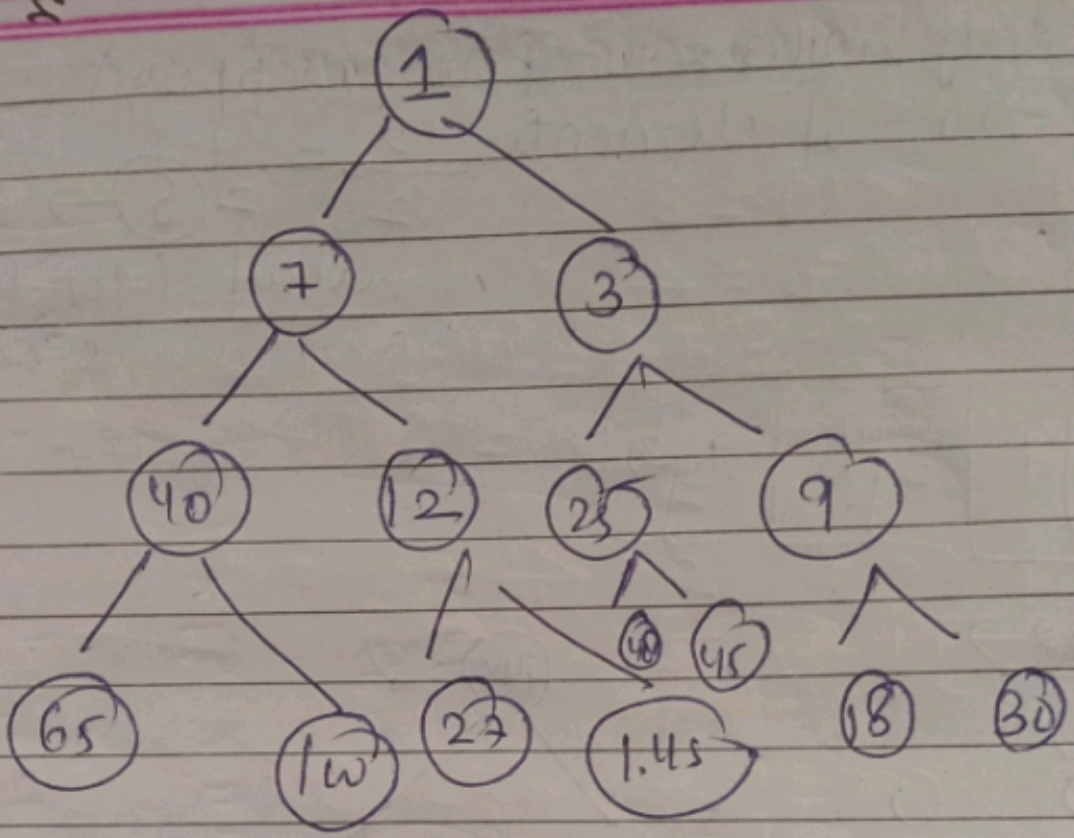
Step 2 →



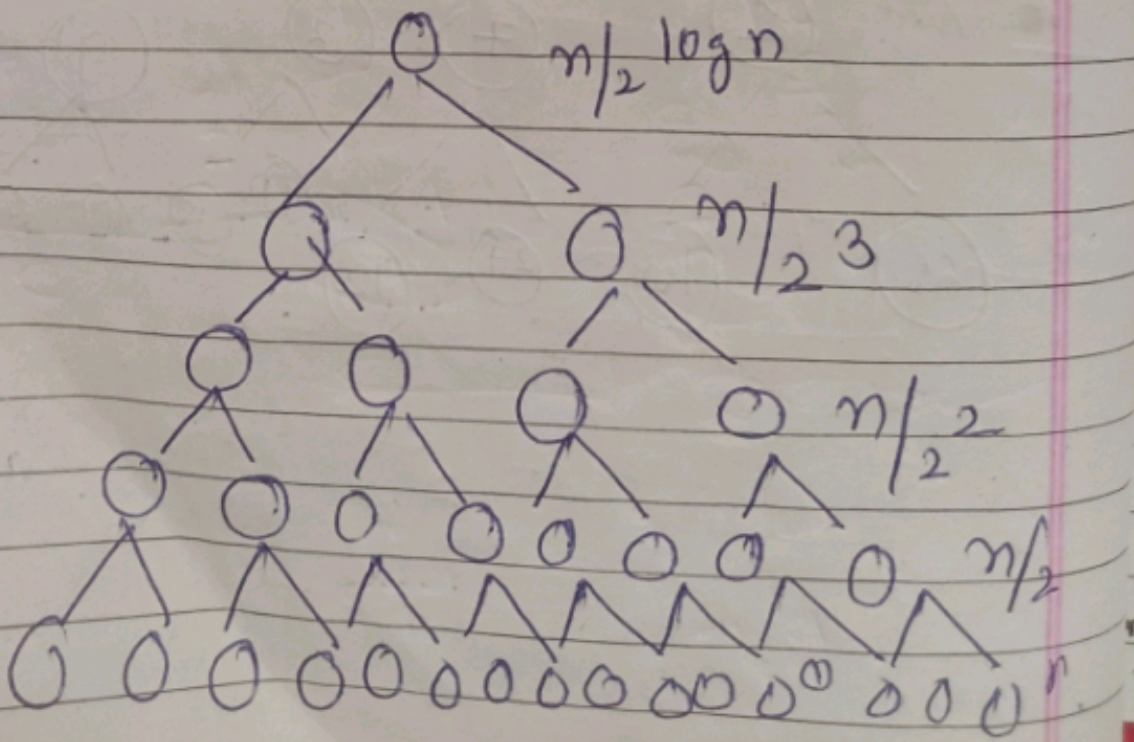
step 2

Heapify method

Step3 →



Heapify method



Total no of swaps = for S

$$S = \frac{n \times 0}{2^0} + \frac{n * 1}{2^1} + \frac{n * 2}{2^2} + \frac{n * 3}{2^3}$$

$$\cdots \quad \frac{n * \log n}{2^{\log n}}$$

$$S = n\left[\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots \frac{\log n}{2^{\log n}}\right]$$

$$\cdots\cdots\cdots (1)$$

multiply equation 1 by $1/2$

$$\frac{S}{2} = n\left[\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + \cdots \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n + 1}}\right]$$

$$\cdots (2)$$

Formula $S = a\frac{(1-r^n)}{1-r}$

$$\frac{S}{2} = n\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots + \frac{1}{2^{\log n}}\right] - \frac{\log n}{2^{\log n+1}}$$

[use GP formula $r < 1$]

$$\frac{S}{2} = n\left[\frac{\frac{1}{2}\left(1 - \frac{1}{2^{\log n}}\right)}{1/2} - \frac{\log n}{2^{\log n+1}}\right]$$

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

$2^{\log n}$ subtract (2) from (1)

$= n\log 2$

$= n^2$

$= n^2$

$$\frac{S}{2} = n \cdot \left(\frac{n-1}{n} - \frac{\log n}{n \cdot 2} \right)$$

$$\frac{S}{2} = n - 1 - \frac{\lg n}{2}$$

$$S = 2\left(n - 2 - \frac{\lg n}{2}\right)$$

$$S = 2n - 2 - \lg n$$

highest power.

Need of Algorithm

1. To understand the basic idea of the problem.

2. To find an approach to solve the problem.

3. To improve the efficiency of existing techniques.

4. To understand the basic principles of designing the algorithms.

5. To compare the performance of the algorithm with respect to other techniques.

6. It is the best method of description without describing the implementation detail.

7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.

8. A good design can produce a good solution.

9. To understand the flow of the problem.

10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)

11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.

12. With the help of algorithm, we convert art into a science.

13. To understand the principle of designing.

14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

ALGORITHM: Introduction to Algorithms

Algorithm is a step by step method to solve a problem.

characteristics of algorithm:

1. Input : Algorithm must contain '0' or more input.
2. Output : Algorithm must contain '1' or more output.
3. Finiteness : Algorithm must complete after finite no. of steps.
4. Definiteness : The step of the algorithm must be defined precisely or clearly unambiguous
5. Efectiveness : The step of an algorithm must be effective (can be done successfully)

Analysis of Algorithm:

Analysis of algorithm depend upon the time & memory taken by the algorithm.

Time Complexity:

The amount of time required by the algorithm is called time complexity.

Space Complexity:

The amount of space / memory required by the algorithm is called as space complexity.

Growth functions:

	2^n	$n!$
n=1	2	1
n=2	4	2
n=3	8	6
n=4	16	24
n=5	32	120

$n! > 2^n \forall n \geq 4$

$n!$ has higher growth rate than 2^n

Q: Arrange the following according to the increasing order of growth rate.

$n, 2^n, n\log n, n!, n^3, n^5, n^2, 1$

Ans: $1, n, n\log n, n^2, n^3, n^5, 2^n, n!$

constant ≤ Logarithm ≤ polynomial ≤ ≤ exponential ≤ factorial

Date - 13/1/18

⇒ There are 3 notations for time complexity

1. Big oh (O) notation :

Let f(n) & g(n) are two functions

$f(n) = O(g(n))$ [read as f(n) is big oh of g(n)] when $f(n) \leq c \cdot g(n)$. Here c - constant

Example :

We say that it requires Max^m 3hrs [from Cuttack to Bhubaneswar.

2. Omega (Ω) notation

Let f(n) & g(n) are two functions

$f(n) = \Omega(g(n))$ [read as f(n) is Omega of g(n)] when $f(n) \geq c \cdot g(n)$

where, c is constant.

Example :

We say that it require Min^m 30 mins from Cuttack to Bhubaneswar.

3. Theta (Θ) notation:

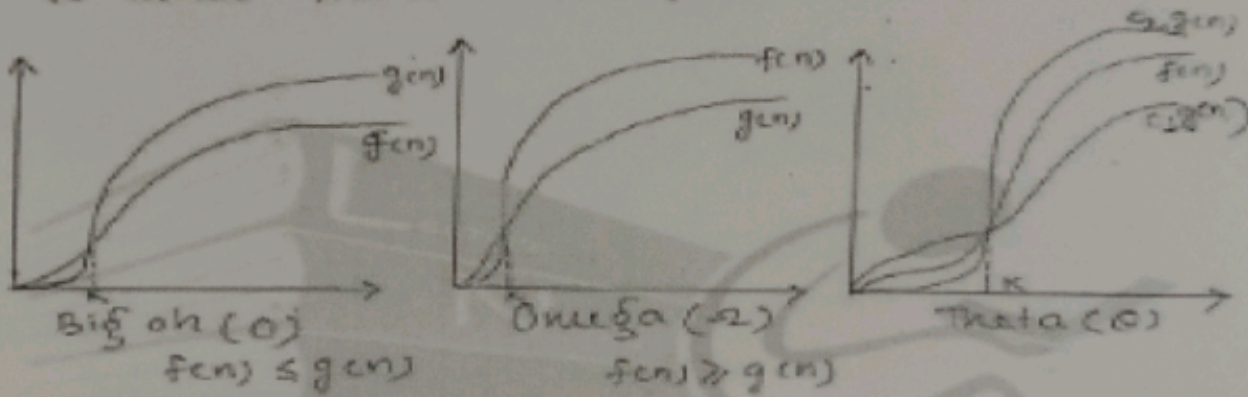
Let f(n) & g(n) are two functions.
f(n) = Θ g(n) [read as f(n) is theta of g(n)]

$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Here c_1, c_2 are constants.

Examples:

We say that it require approximately 40 mins from cuttack to Bhubaneswar.



Example:

f(n) = n+5
f(n) = 2n+3
f(n) = n^2+6
f(n) = n^3+$2n^2$+3
f(n) = n+6n+3

Big oh → Low $\frac{\text{upper bound}}{\text{Max}^m}$

Omega → High $\frac{\text{Lower bound}}{\text{Min}^m}$

Ex:

2n + 3 = O(n)
= O(n^2)
= O(n^3)

Q Prove that $5n+3 = O(n)$

Ans: $f(n) = 5n+3$ $\quad \therefore f(n) \leq c \cdot g(n)$ (c: constant)

$5n+3 \leq 6 \cdot n$

$= O(n)$

Q Prove that $6n^2+2n+3 = O(n^2)$

Ans: $6n^2+2n+3 \leq 7 \cdot n^2$

$= O(n^2)$

Q Prove that $5n+3 = \Omega(n)$

Ans: $5n+3 \geq 4 \cdot n$ here 'c' = 4

$= \Omega(n)$

Q Prove that $6n^2+2n+3 = \Omega(n^2)$

Ans: $6n^2+2n+3 \geq 5 \cdot n^2$ here c = 5

$= \Omega(n^2)$

'c' is decided by user. Means user have to put a value for 'c'.

Q Prove that $5n+3 = \Theta(n)$

Ans: $4 \cdot n \leq 5n+3 \leq 6 \cdot n$

c_1 $\quad\quad$ c_2

$= \Theta(n)$

Recurrence:

→ The word recurrence is derived from "recurssion" or "repeatation"

→ Recursion is a technique which call the same function repeatedly.

Example: Factorial problem can be solved using recursion.

Recurrence tree method :

This method is used to guess the solution.

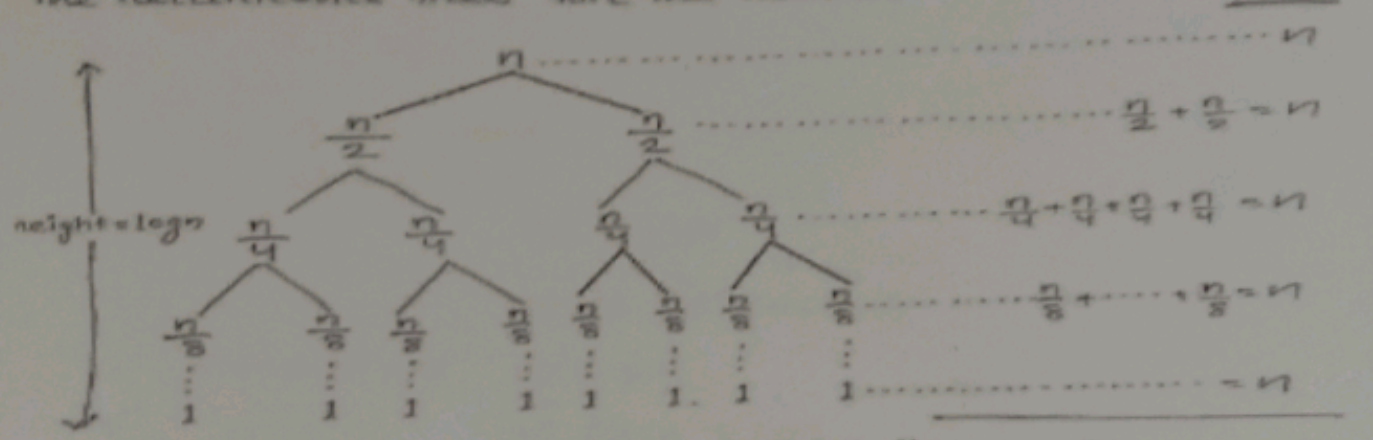
- Recurrence tree gives a graphical view of recurrence.

Consider the following recurrence :

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

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

The recurrence tree for the above :



cost of level = addition of nodes present in the level

total cost = addition of the all level.

[∵ cost indicates time]

Example-3 : $T(n) = 2T(\frac{n}{2}) + n$

Solve the above recurrence by Master Method.

Ans:

We know that, $T(n) = aT(\frac{n}{b}) + \Theta(n^k \log n^p)$

Here, $a = 2$ $\quad \because n^1 \log n^0 = n$

$b = 2$

$k = 1$

$p = 0$

$a = 2$ and

$b^k = 2^1 = 2$

$\Rightarrow a = b^k$

$\Rightarrow$ It is case 2

Here $P = 0$ $\Rightarrow P > -1$

$\Rightarrow$ case 2(a)

$\Rightarrow T(n) = \Theta(n^{\log_b a} \log n^{p+1})$

$= \Theta(n^{\log_2 2} \log n^{0+1})$

$= \Theta(n^1 \cdot \log n^1)$

$= \Theta(n \cdot \log n)$ Ans)

2. Substitution Method :

Substitution method has two steps,

1. Assume that solution is correct.
2. Prove this assumption by substitution method.

✓

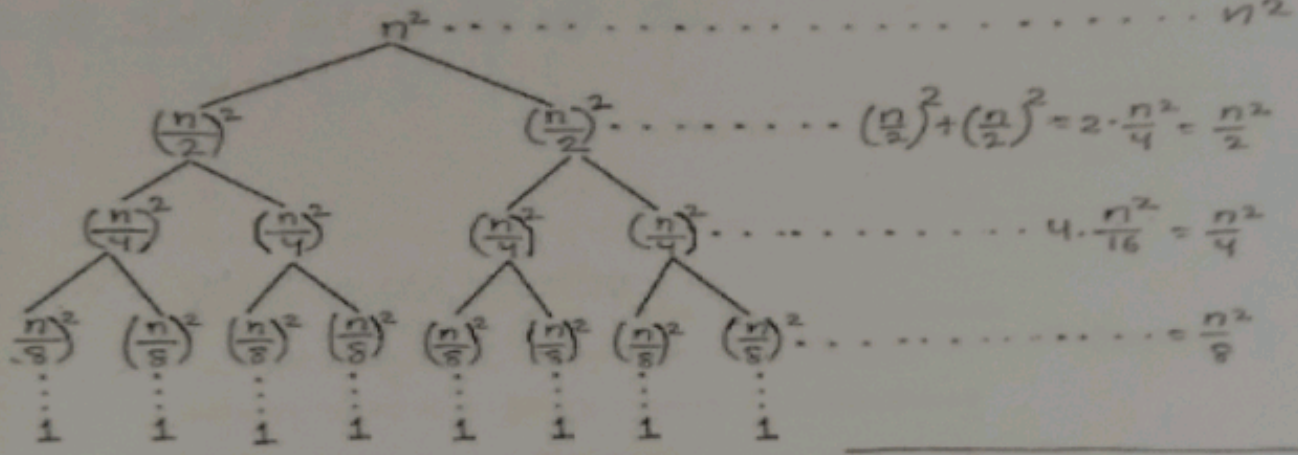
Example 1 :

Consider the following recurrence

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

Design the recurrence tree.

Find total time complexity.



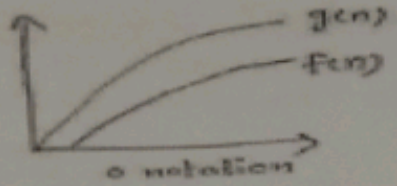
Total time $= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} \cdots)$

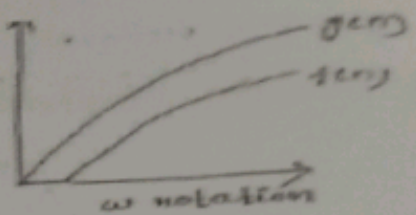
$= n^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$

$= n^2$ (constant)

$= O(n^2)$

$=$





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

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

Divide and conquer method :

→ Divide and conquer is also called DANDC

→ It is one of the algorithm design method

→ DANDC has 3 parts

1. Divide the problem into number of subproblems.
2. Conquer means solve the subproblem recursively
3. Combine the solution of subproblems

Example :
Binary search, quick sort, merge sort

Note :
DANDC is normally recursive

Merge Sort :

→ Merge means combine.

→ Sorting means arranging elements in increasing or decreasing order.

→ Merge sort has two operations,

1. Divide
2. Merge

Consider an array of n elements,

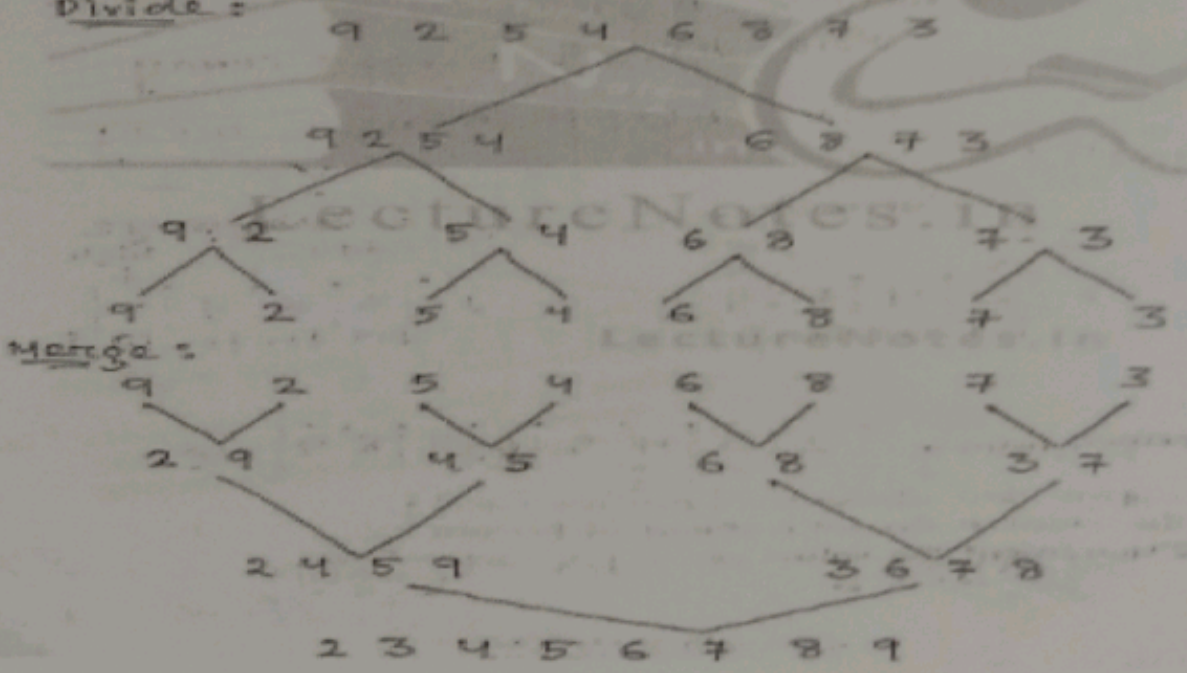
1. Divide the array into sub array recursively
2. Merge the subarray recursively.

Merging operation creates sorted elements.
Hence, the name is merge sort

01-05/02/18

Example : 9 2 5 4 6 8 7 3

Divide :



Merge :

Divide Operation :

1. Divide the array into two sub array at mid position.

   Subarray 1 is from low to mid.
   Subarray 2 is from mid+1 to high.

2. Divide the subarray recursively until more than one element is present.
   (low < high)

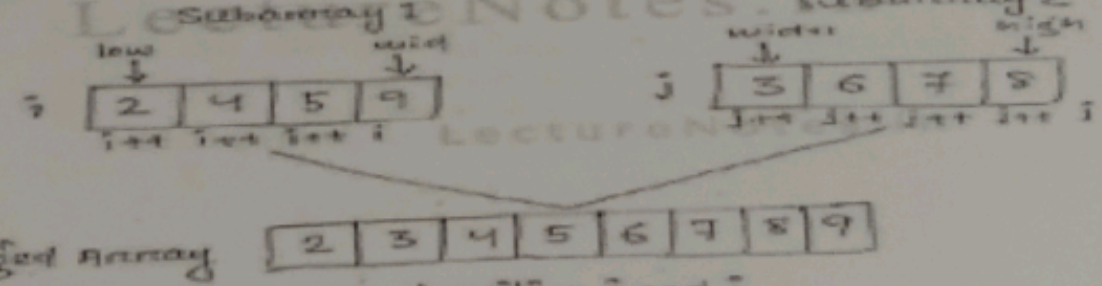
Merge Operation :

// A[i] = element of subarray 1
// A[j] = element of subarray 2

Compare A[i] and A[j] as following

```
if A[i] < A[j]
        store A[i] in merged array
Else
        store A[j] in merged array
```

Example :



Compare two elements at position i and j.
The smaller element is stored at position k.
Increament the value of i, j, k accordingly.

Q. Write a procedure (Algorithm or pseudo code) for divide and merge operation.

```
Algorithm divide (low, high)
{
  // low = index of 1st element of array
  // high = index of last element of array
    low = 0
    high = n-1                    // divide until more than one element
    while (low < high)
    {  // divide the array at mid position.
       divide (low, mid)
                  // Subarray 1 is from low to mid
       divide (mid+1, high)
                  // Subarray 2 is from mid+1 to high.
       merge (low, mid, high)
    }
}

Algorithm merge (low, mid, high)
{
   // A is an array
   // B is an array to store result (merged array)
   while (i<=mid and j<=high)
   {                  // element at position i   // element at position j
      if (A[i] < A[j])
      {
          B[k] = A[i]
          k++
          i++
      }
      else
      {
          B[k] = A[j]
          k++
          j++
      }
   }
}
```

Analysis of Merge sort:

Merge sort apply a common technique in all cases.
Hence, Merge sort has one case for all set of Problems.
⇒ Merge sort does not have best, worst or average case.

The recurrence, equation of merge sort is given by:

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

- $T(n)$: Time to sort n elements
- $2T\left(\frac{n}{2}\right)$: time to sort 2 subarray of size n/2
- $O(n)$: merge the 2 subarray

Solve the above recurrence by master method

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

Q: What is the time complexity of merge sort.
Ans: The time complexity of merge sort is $O(n \log n)$.

Quick Sort:

Quick sort is an "Divide and conquer (resolve)" algorithm

Steps:
1. Select a pivot element. Pivot means target. Any element can be taken as pivot.
2. Partition Operation: Partition means divide.
   → Partition operation devide the array into 2 subarray. [Left & right subarray]
   → Partition operation places the pivot element at proper position.

that means, all element before pivot is smaller. and all element after pivot is greater.

3. Recursively apply quicksort to subarrays

left subarray, right subarray

01-10/02/18

Example 1

(11)	9	2	6	19	14	18
↓ Pivot	i++	i++	i++ → j	i--	i--	i-- ←

9 < 11 => i++
2 < 11 => i++
6 < 11 => i++
19 ≮ 11 -> stop i

18 > 11 => j--
14 > 11 => j--
19 > 11 -> j--

Now i and j cross each other => stop
Swap 11 and 6 [i.e swap pivot and element at j]
Elements after swapping are shown below.

6	9	2	(11)	19	14	18
left			↓ pivot	right		

Now, pivot is placed in proper position

Example-2

Arrange following elements by appling quickshot.

(8)	7	11	6	4	5	9
Pivot	i++	i	←		i	j--

7 < 8 -> i++
11 ≮ 8 => stop i

9 > 8 => j--
5 ≯ 8 => stop j

Swap the element at i and j. elements after swapping ar shown below.

(8)	7	5	6	4	11	9
			i →	i	→ j	

6 < 8 => i++
4 < 8 -> i++

Now, i and j cross each other -> stop.

Swap 8 and 4 [i.e swap pivot and element at j]
Elements after swapping are shown below.

4 7 5 6 (8) 11 9
left subarray | pivot | right subarray

Write an algorithm for quick sort.

Ans:
// A is an array of n elements
// low = index of 1st element of array
// high = index of last element of array

Step 1 : low = 0
high = n-1

Step 2 : Consider 1st element is pivot

Step 3 : i = 1
j = high

Step 4 : Continue i++ while (A[i] < pivot) (element at i)
Continue j-- while (A[j] > Pivot) (element at j)
[i moves in forward direction
j moves in backward direction]

Step 5 : Swap A[i] and A[j]
Make i++ and j--

Step 6 :
Repeat step4 and step5 until i and j Cross each other.

Step 7 :
Swap pivot and A[j]
Now, pivot is placed in proper position.

Step 8 :
Divide the array into 2 subarray.
Left Subarray = Elements present in the left side of pivot
Right Subarray = Elements present in the right side of pivot.

Step 4: Apply Quicksort to both subarray recursively

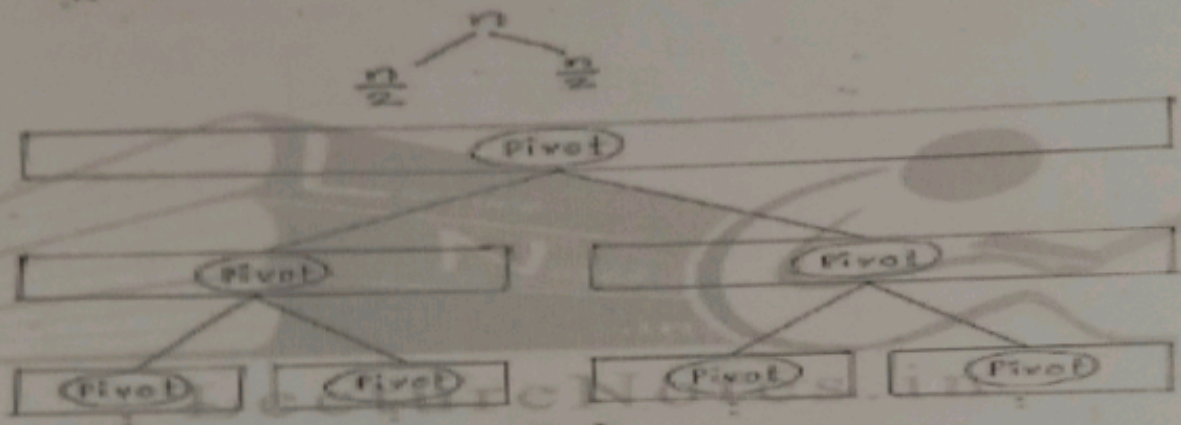
Analysis of Quick sort:

Time complexity depends on the position of Pivot element.

1) Best Case:

→ Pivot is placed in the middle position.

→ 'n' elements are divided into n/2 and n/2



The recurrence eqn is

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

Solving this recurrence we get,

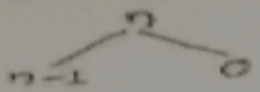
$$T(n) = O(n \log n)$$

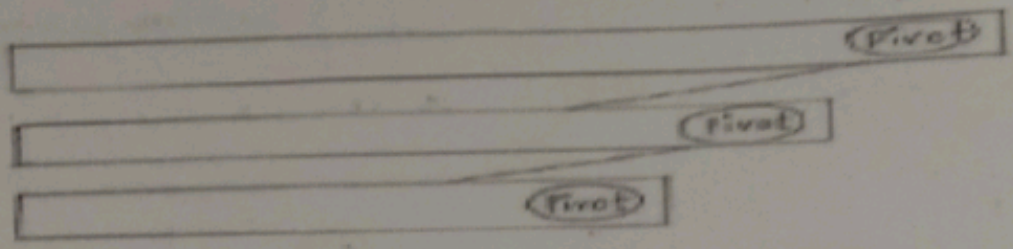
∴ Time complexity = $O(n \log n)$

2) Worst Case:

→ Pivot is placed in the first or last position

→ 'n' elements are divided into (n-1) and 0





Recurrence eqn is $T(n) = T(n-1) + O(n)$

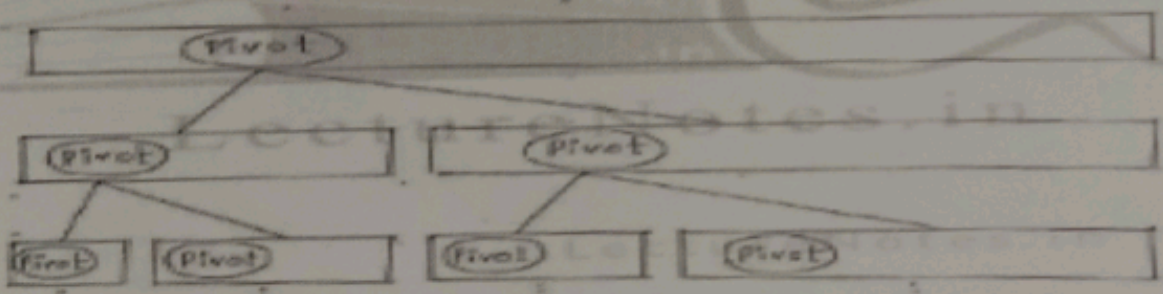
Solving this recurrence we get.

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

∴ Time complexity = $O(n^2)$

3. Average case :

Pivot is not in middle, first or last position. Let, n elements are divided into n/4 & 3n/4



Recurrence eqn is $T(n) = T(n/4) + T(3n/4) + O(n)$

Solving this recurrence we get,

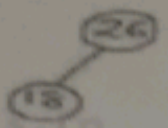
$T(n) = O(n \log n)$

∴ Time complexity = $O(n \log n)$

Q: Create max-heap with following elements.

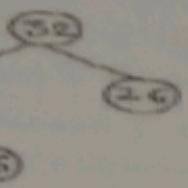
26 18 30 32 16 74

Insert 26, 18

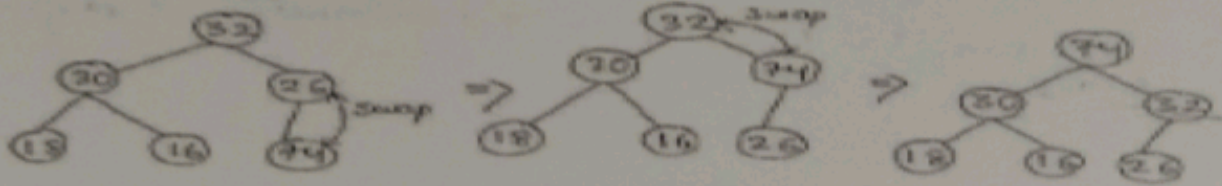


Insert 30

Insert 16



Insert 74



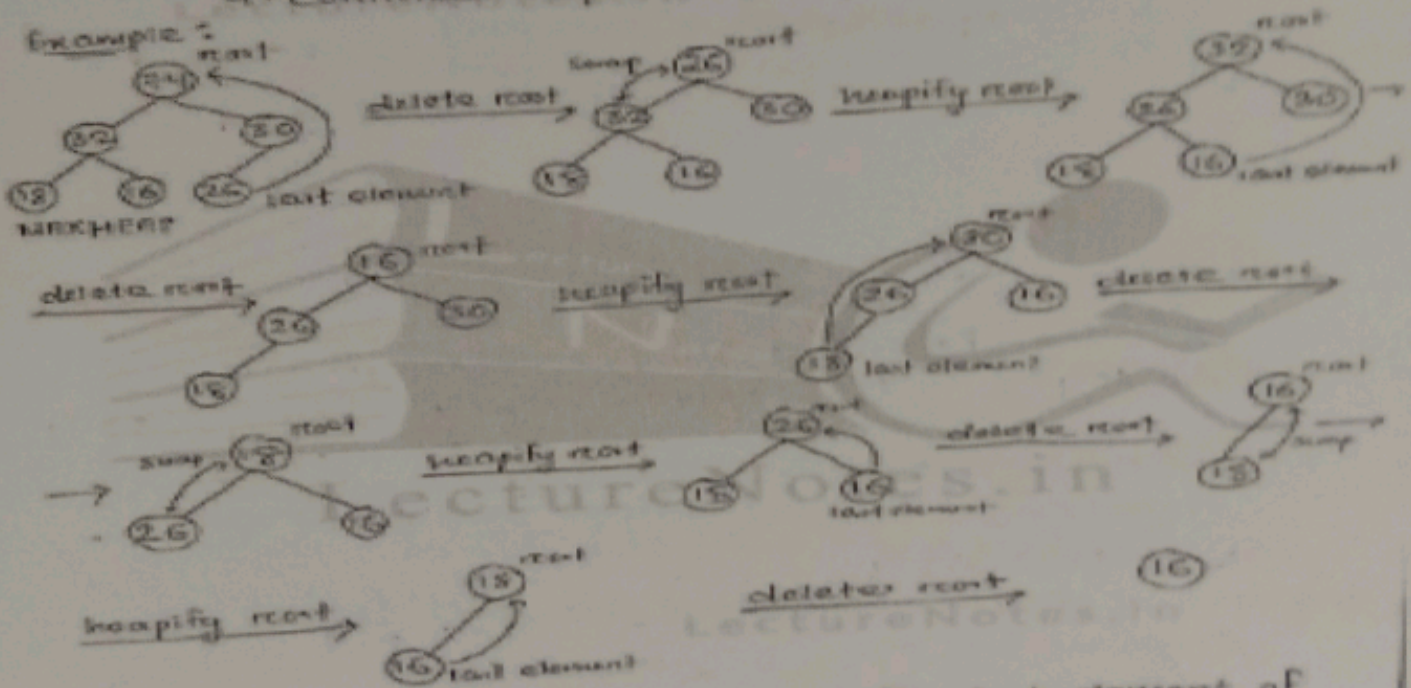
Heapsort : When we build a maxheap, the largest element is at root.

Delete root to get the Largest element. Again, make maxheap from remaining n-1 elements Delete root to get the 2nd Largest element. So on.

steps:

1. Build maxheap from n elements.
2. Delete the root.
3. heapify the new root.
4. Continue step2 and step3 till heapsize = 1

Example:



Deleted roots are stored at the last element of array.

Stored array is.

16	18	26	30	32	34

```
Algorithm heapsort (A)
{
    // A is an array of n elements
       heap size = n
       Build maxheap(n)
    While (heapsize > 1)
    {
          Swap (A[1] , A[heapsize])   } delete root
                (root)  (last element)
          heapsize = heapsize - 1     (
          heapify (1) ---------->{ heapify root
    }
}
```

Analysis of heapsort :

Time for Buildmaxheap = O(nlogn)

While loop executes O(n) times.

Time for a heapify() = O(logn)

=> Time for while loop = O(nxlogn)
= O(nlogn)

Total time for heapsort = O(nlogn+nlogn)
= O(2nlogn)
= O(nlogn)

Lower bound of sorting :

- Lower bound means minimum time.
- For minimum time, we use Ω notation.

We know that,

time taken by an algorithm = no. of comparisions.

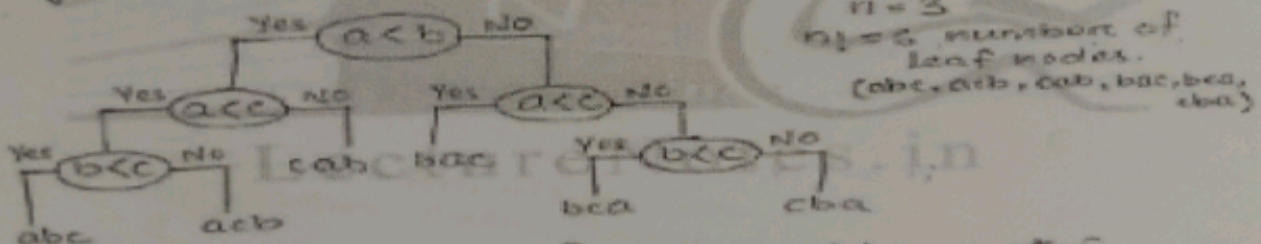
- Comparision is also called binary comparision. That means comparision betn two elements.
- Merge sort & heap sort use binary comparision These are called comparision based sorting.
- Any sorting algorithm have atleast nlogn comparisin

⇒ **Lower bound of comparision based sorting algorithm = $\Omega(n \log n)$**

Example :

Consider 3 elements a,b,c

the decission tree for a sorting algorithm is given below,



n = 3

n! = 6 number of leaf nodes.

(abc, acb, cab, bac, bca, cba)

Fig : Decission tree for comparision sorting

- A tree of height 'h' has maximum 2^h leaf nodes.
- It is found that total number leaf nodes = n!

that means, $2^h \geq n!$

$\Rightarrow \log 2^h \geq \log(n!)$

$\Rightarrow h \log 2 \geq \log(n!)$

$\Rightarrow h \geq \log n!$ ——→

$\Rightarrow h = \Omega(n \log n)$

$n! = n(n-1)(n-2)\cdots 1$

$n! = n^n\{1(1-\frac{1}{2})(1-\frac{2}{3})\cdots\frac{1}{n}\}$ (product < 1)

$n! \geq n^n$

$\log n! \geq \log n^n$

$\log n! \geq n \log n$

$\log n! = \Omega(n \log n)$

Here, h = height of decission tree

= no. of comparisions.