Amrita Naik
DBCE

Algorithm Intro

An algorithm is a finite set of instruction, if followed, accomplishes a particular task.

→ Algorithm Specification

ex: Algorithm SelectionSort (a, n)
// sorts array in nondecreasing order

```
{
    for i:=1 to n do
    {
        j:=i
        for k:=i+1 to n do
            if (a[k]<a[j]) then
                j:=k;
        t:=a[i];
        a[i]:=a[j];
        a[j]:=t;
    }
}
```

Recursive function / Algorithm:

- An algorithm is called recursive if same algorithm is called again in the body.

1) ex: Algorithm fact(x)

```
{
    if (x = 1)
        return 1
    else
        return x * fact(x-1)
}
```

2) ex: Algorithm TowersOfHanoi (n, x, y, z)

```
{ //move top 'n' disk from tower x
  //to tower y
  if (n >= 1) then
  {
      TowersOfHanoi (n-1, x, z, y);
      write ("move top disk
      from tower", x, "to tower
      ", y)
      TowersOfHanoi (n-1, z, y, x);
  }
}
```

Discuss the criteria for designing an effecient algorithm.

Sol:- An algorithm has to satisfy following criteria:

(i) Input: Zero or more quantities are externally supplied.

(ii) Output: At least one quantity is produced.

(iii) Definiteness: Each instruction is clear and unambiguous.

(iv) Finiteness: If we trace out the instuction of an algorithm, then the algorithm should terminate, for all cases, after a finite no of steps.

(v) Effectiveness: Every instruction must be basic, traceable and feasible.

Performance Analysis:

Algorithm are judged based on:

(i) Space Complexity:- is the amount of memory it needs to run to completion

(ii) Time Complexity - is the amount of computer time it needs to run to completion.

i) Space Complexity: (S(P))

The space needed by each algorithm is sum of

(i) fixed part → i.e space for code, space for fixed variable, space for constants.

(ii) Variable part → i.e space needed by reference variable, recursion stack

ex: $S(P) = c + S_p$

- c → Constant
- S_p → instance characteristic [variable]

ex:

1) Algorithm abc(a, b, c)

```
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}
```

Explanation:-

→ Here we use 3 fixed variable a, b, c

Hence Space Complexity $S(P) = 3 + 0$

- 3 → fixed variable
- 0 → no instance characteristic

2) Algorithm Sum (a, n)

```
{
    s := 0.0;
    for i := 1 to n do
        s := s + a[i];
    return s;
}
```

Explanation: The space needed for a is n. The space needed for fixed variable n, i and s is 3.

$\therefore S(p) = 3 + S_p$

$= 3 + n$

3) Algorithm RSum (a, n)

```
{
    if (n<=0) then
        return 0;
    else
        return RSum(a, n-1) + a[n];
}
```

Explanation: The recursion stack requires space for local variable, return address and formal parameters.

→ Function has 3 variables → n, pointer to a and return address

- Recursive function is called $(n+1)$ times.

- Hence Space Complexity

$$S(P) = c + S_P$$
$$= 0 + 3(n+1)$$

Time Complexity:

- Time taken by program 'P' is sum of compile time and run time.
- We are concerned with run time & is denoted by t_P

$$t_P(n) = c_a * ADD(n) + c_s * SUB(n) + c_m * MUL(n) + c_d * DIV(n) + c_c * COM(n) + \text{----}$$

where c_a, c_s, c_m, c_d, c_c are time needed for addition, substraction, multiplication, division and comparision.

- 'n' denotes the instance characteristics.

→ Since execution time $t_P(n)$ can be obtained only experimentally, we consider only the program steps

→ Program steps is defined as a meaningful segment of program that has execution time independent of the instance characteri

-istics:

→ There are 2 ways to determine step counts.

1st method:

1) A global variable 'count' is initialized to zero. Each time a statement in program is executed, count is incremented by step count.

2) 2nd method: A table is built in which we list the total number of steps contributed by each statement. This is determined by number of steps per execution of statement and total number of times each statement is executed.

st method examples

er> Algorithm Sum(a,n)

```
{
    s:=0.0;
    count:=count+1;    // count for assg
    for i:=1 to n do
    {
        count:=count+1;    // count for for
        s:=s+a[i];
        count:=count+1;    // count for add
    }
    count:=count+1;    // count for last time
    count:=count+1;    // in for
                       // count for return
```

```
    return s;
}
```

$\therefore t_{sum}(n) = 2n+3$

2 ex) Algorithm Rsum (a, n)

```
{   count := count + 1;          // for if condition
    if (n <= 0) then
        count := count + 1;      // for return
        return 0.0;

    else
        count := count + 1;      // Addition & return statement
        return Rsum (a, n-1) + a[n];
}
```

$$\therefore t_{Rsum}(n) = 2 + t_{Rsum}(n-1)$$

$$= 2 + 2 + t_{Rsum}(n-2)$$

$$= 2 + 2 + \cdots + 2 + t_{Rsum}(n-3)$$

$$= 2(n) + t_{Rsum}(0)$$

$$t_{Rsum}(n) = 2(n) + 2$$

3 ex) Algorithm Add (a, b, c, m, n)

```
{
    for i := 1 to m do
    {   for j := 1 to n do
        {
            c[i, j] := a[i, j] + b[i, j];
        }
    }
```

method 1 solution

```
Algorithm Add(a, b, c, m, n)
{
    for i:=1 to m do
    {
        count:=count+1;    // for i

        for j:=1 to n do
        {
            count:=count+1;    // for j
            c[i,j]:=a[i,j]+b[i,j];
            count:=count+1;    //for assg
        }
        count:=count+1;    // for last time
                           // of j
    }
    count:=count+1;    // for last time of i
}
```

$t_{Add}(m,n) = m*(2n+1)+1$

$= 2mn+2m+1$

2nd method:

ex> Algorithm Sum(a, n)

```
{
    s:=0.0;
    for i:=1 to n do
    {   s:= s+a[i];
    }
    return s;
}
```

Solution:

Statement	s/e	frequency	total steps
Algorithm Sum(a, n)	0	—	0
{	0	—	0
s:=0.0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return s;	1	1	1
}	0	—	0
Total →			2n+3

x)

Statements	s/e	frequency		Total	
		n=0	n>0	n=0	n>0
Algorithm RSum(a, n)	0	–	–	0	0
{					
if (n ≤ 0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else					
return RSum(a, n-1)+a[n];	1+x	0	1	0	1+x
}	0	–	–	0	0
				2	2+x

$\therefore t_{RSum} = 2 + x$

[where $x = t_{RSum}(n-1)$]

$t_{RSum} = 2 + t_{RSum}(n-1)$

$= 2 + 2 \cdots n \text{ times} + 2$

$= 2n + 2$ //

ex) Algorithm Add(a, b, c, m, n)

{

for i:=1 to n do

for j:=1 to n do

c[i,j]:=a[i,j]+b[i,j];

}

s/e	freq	total
0	–	0
0	–	0
0		
1	n+1	n+1
1	n(n+1)	n²+n
1	n²	n²
0	–	0
	Total →	2n²+2n+1

4 ex)

	s/e	freq		total	
Algorithm Fibonacci(n)					
{		n>1	n=1	n>1	n=1
if (n≤1) then	1	1	1	1	1
write(n);	1	-	1		1
else					
{					
fm2 := 0;	1	1	-	1	0
fm1 := 1;	1	1	-	1	0
for i := 2 to n do	1	n	-	n	0
{					
f1 := fm1 + fm2;	1	n-1	-	n-1	0
fm2 := fm1;	1	n-1	-	n-1	0
fm1 := f1;	1	n-1	-	n-1	0
}					
write(f);	1	1	-	1	0
}					
}				4n+1	2

Total count of (if statement)

$\Rightarrow 2$

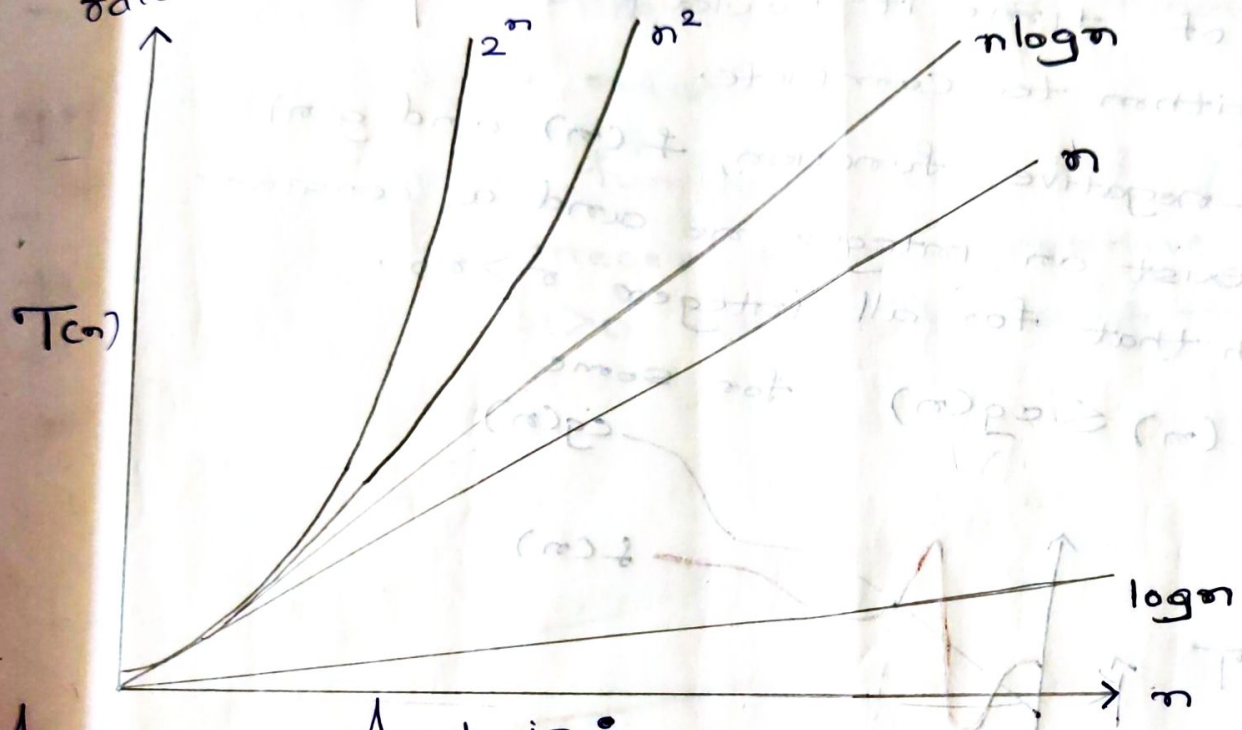
Total (else statement)

$\Rightarrow 3+n+3(n-1)+1$

$= 4n+1$

Growth rate of function:

- One of important problems in Computer Science is to get the best measure of growth rate of algorithm.



Asymptotic Analysis:

Asymptotic means a line that tends to converge to a curve which may or may not eventually touch the curve. Its a line that stays within bounds.

Asymptotic Notation

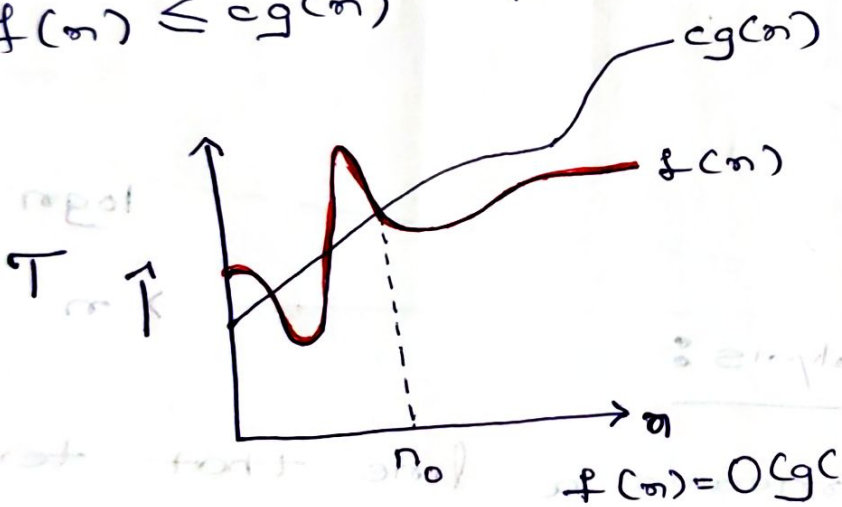
5 types

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

1) Big Oh notation: (O) is a formal method to determine upper bound of algorithm running time. It is measure of longest amount of time it could possibly take for algorithm to complete.

- For non-negative function, $f(n)$ and $g(n)$ if there exist an integer n_0 and a constant $c>0$ such that for all integer $n>n_0$.

$f(n) \leq cg(n)$ for some



$f(n) = O(g(n))$ for $n>n_0$

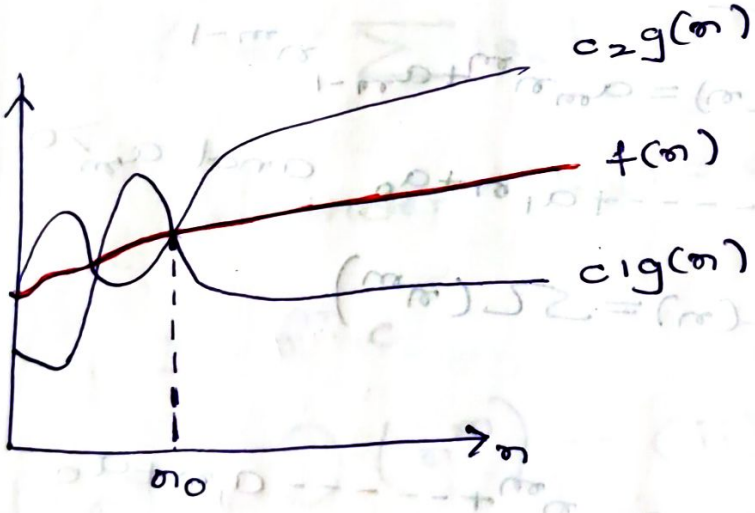
2) Big-omega notation: (Ω)

- For non-negative function, $f(n)$ and $g(n)$ if there exist an integer n_0 and a constant $c>0$ such that for all integer $n>n_0$

$f(n) \geq cg(n)$ for $n>n_0 \Longrightarrow f(n) = \Omega(g(n))$

(3) Theta notation: (Θ)

- The lower and upper bound for function is provided by theta notation.
- For non-negative function $f(n)$ and $g(n)$, it there exist an integer n_0 and positive constant c_1 and c_2 i.e $c_1 > 0$ and $c_2 > 0$ such that all integers $n > n_0$



if $c_1 g(n) \leq f(n) \leq c_2 g(n)$,

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

(4) Small omega (ω):

- is used to denote lower bound that is asymptotically tight.
- $\omega(g(n)) = \{f(n)$: for any +ve constant $c > 0$ if a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$

(5) little-oh notation (o):

- is used to denote upper bound that is asymptotically tight.

- $o(g(n)) = \{f(n)$: for any +ve constant $c>0$ if a constant $n_0>0$ such that $0 \le f(n) < cg(n) \quad \forall\ n \ge n_0\}$

Some problems:

1) Prove that if $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$ and $a_m > 0$

Show that $f(n) = \Omega(n^m)$

Solution: $f(n) = a_m n^m + \cdots + a_1 n + a_0$

$f(n) \ge a_m n^m$

Lets consider a_m as constant c

$\therefore f(n) \ge c\, n^m$

$\therefore f(n) = \Omega(n^m)$ — (i)

2) Prove that if $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$ then show that $f(n) = O(n^m)$

Solution:

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_m n^m$$

$$f(n) \leq \sum_{k=0}^{m} |a_k| n^k$$

$$\leq n^m \sum_{k=0}^{m} |a_k| n^{k-m}$$

$$\leq n^m \sum |a_k|$$

Consider $|a_k| =$ constant $\rightarrow c$

$$\leq n^m c$$

$$\therefore f(n) = O(n^m) \quad - (ii)$$

3) Prove that if $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$ then show that $f(n) = \Theta(n^m)$

Solution:- we know that

$f(n) = O(n^m)$ [from (ii)]

$f(n) = \Omega(n^m)$ [from (i)]

From (i) and (ii) we say $f(n) = \Theta(n^m)$

3) Let $f(x) = O(g(x))$ and $g(x) = O(h(x))$

Show that $f(x) = O(h(x))$

Solution: if $f(x) = O(g(x))$

then there are positive constant c_1 such that

$$0 \leq f(n) \leq c_1 g(n)$$

if $g(n) = O(h(n))$

then there exist a positive constant c_2 such that

$$0 \leq g(n) \leq c_2 h(n)$$

$$\therefore 0 \leq f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$$

$$\leq c_3 h(n)$$

$$f(n) \leq c_3 h(n)$$

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

Growth of Function :-

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Algorithm Test(n)	s/e	freq	Time
{	—	—	—
for i=1 to n do	1	O(n)	O(n)
for j=1 to n do	1	$O(n^2)$	$O(n^2)$
for k=1 to n do	1	$O(n^3)$	$O(n^3)$
x = x+1	1	$O(n^3)$	$O(n^3)$
}			$T_p = O(n^3)$

Randomized Algorithm :-

— An algorithm that uses random number to decide the logic in an algorithm is called Randomized algorithm.

— There are 2 types:

(i) Las Vegas

(ii) Monte Carlo

Difference b/w Las-Vegas and Monte Carlo Algo

Las-Vegas	Monte Carlo
1) May not return an answer at all but if they do return a answer, it is gaurant-eed to be correct.	1) May return an answer that is not correct.
2) May take longer time.	2) Has better performance.
3) Running time fluc-tuates	3) Running time is fixed.
Ex: Quicksort	Ex: Karger's algo

Las Vegas example:

Algorithm: Identifying repeated Elements

ex: Consider an array a[] of n numbers, it has n/2 distinct elements and n/2 copies of another element, the problem is to identify repeated elements.

$a = \{3, 5, 4, 3, 3, 6, 3, 2\}$

$n \rightarrow 8$

```
Algorithm RepeatedElem(a, n)
{
    while (true)
    {
        i = rand() mod n+1;
        j = rand() mod n+1;
        if (i≠j and a[i] = a[j])
            return a[i];
    }
}
```

Monte Carlo Algorithm example

ex: Randomized primality test

```
Algorithm RandPrimality-Test(n)
{
    Randomly choose a ∈ [2, n-1]
    Calculate a^(n-1) mod n
    if a^(n-1) mod n = 1 then
        n is possibly prime
    else
        n is definitely not prime
}
```

Different type of Algorithm are:

(i) Divide and Conquer Algorithm
(ii) Greedy Algo
(iii) Dynamic Algo
(iv) Backtracking Algo
(v) Branch and Bound Algo
(vi) Randomised Algo

(i) Divide and Conquer Algorithm

Given a function to compute on n input, the divide and conquer strategy suggest splitting the input into k distinct subsets $1 < k \leq n$ yielding k sub problems. These sub problems are individually solved and the solutions are combined to obtain the final solution.

```
Algorithm DAndC(P)
{
    if small(P)
        return Solutions
    else {
        divide P into smaller subproblem
                        (P1; ... Pk)
        Apply DAndC to each subproblem
        return combine(DAndC(P1), DAndC(P2)
                        ---- DAndC(Pk))
    }
}
```

Time Compexity of Divide and Conquer Algorithm

$$T(n) = \begin{cases} T(1) & \text{for } n=1 \\ aT(n/b) + f(n) & \text{for } n>1 \end{cases}$$

Master's Theorem:

$$T(n) = aT(n/b) + cn^d \quad \text{for } n>1$$

$$T(n) = O(?)$$

$$= \left\{ \begin{array}{ll} O(n^d) & \text{if } a<b^d \\ O(n^d \log n) & \text{if } a=b^d \\ O(n^{\log_b a}) & \text{if } a>b^d \end{array} \right\}$$

Ex: if $a=1$, $b=2$, $f(n)=c$

Sol $T(n)=?$

Solution: [using master's theorem]

$a = b^d$

$1 = 2^0$

$1 = 1$

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

$= O(n^0 \log n)$

$= O(\log n)$

ex 2) Solve the recurrence relation for

$a=2, b=2, f(n)=cn$

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

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

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

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

$= 4[2T(n/8) + cn] + 3cn$

$= 8T(n/8) + 7cn$ [Assume $2^K = n$]
$\therefore \log n = K$

$= 2^k T(1) + Kcn = n + cn * \log n$

$= O(n \log n)$

(Or)

Master's Theorem

$a=2, \quad b=2, \quad f(n)=cn, \quad d=1$

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

$a = b^d \Rightarrow 2 = 2^1$

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

$= O(n \log n)$

3) Solve the recurrence relation for $a=7$, $b=2$, $f(n)=18n^2$

Solution:

$a > b^d$

$7 > 2^2$

$7 > 4$ [using master's theorem]

$\therefore T(n) = O(n^{\log_b a})$

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

$= O(n^{2.8})$

Binary Search

→ Let a_i be a list of elements in ascending order. The problem is to determine whether a given element x is present in the array or not.

Algorithm Binsearch(a, n, x)

```
{
    low=1, high=n
    while (low<=high)
    {
        mid=⌊(low+high)/2⌋
        if (x<a[mid])
            high= mid-1
        else
        {
```

```
if (x > a[mid])
    low = mid + 1
else
    return mid
}
return "not found"
}
```

eg1 :- -18, -15, 0, 6, 7, 21, 22, 36, 58, 64, 110, 201
Search elem → 6

Solution: Search elem → 6

$$mid = \frac{low + high}{2} = \frac{1+12}{2} = \lfloor 6.5 \rfloor = 6$$

Search elem → 6 < a[6] → a[mid]

6 < 21

then low = 1, high = 6 - 1 = 5

$$mid = \frac{1+5}{2} = 3$$

Search elem → 6 > a[3] → a[mid]

6 > 0

then high = 5

low = mid + 1 = 3 + 1 = 4

$$mid = \frac{4+5}{2} = \lfloor 9/2 \rfloor = 4$$

Search elem $6 = a[mid] \Rightarrow a[4]$

$6 = 6$

element found at $i = 4$ //

Time Complexity =>

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

$= T(n/2) + c$

$a = 1 \quad b = 2 \quad d = 0$

$a = b^d$

$1 = 2^0$

$1 = 1$ By master's theorem

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

$= O(\log n)$

Best case $= O(1)$, Worst case $= O(\log n)$, Average case $= O(\log n)$

2. Finding Maximum and Minimum element in the array.

Algorithm Minimax (i, j, max, min)

{

if (i = j)

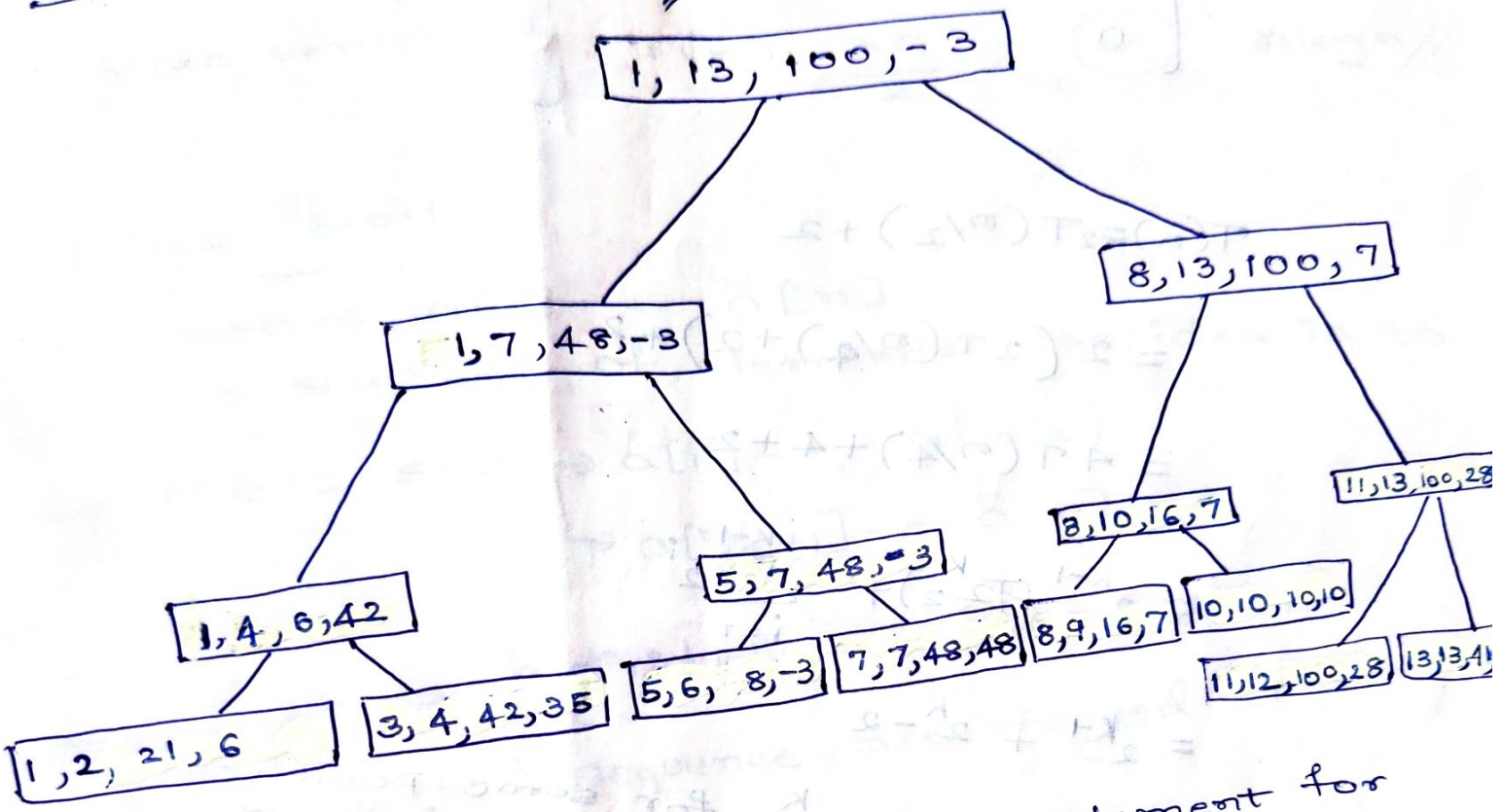
return max = min = a[i]

```
else if (i=j-1)
{
    if (a[i]<a[j])
    {
        max=a[j]
        min=a[i]
    }
    else
    {
        max=a[i]
        min=a[j]
    }
}
else
{
    mid=[(i+j)/2]
    minmax(i, mid, max, min)
    minmax(mid+1, j, max1, min1)
    if(max<max1)
        max=max1
    if(min>min1)
        min=min1
}
}
```

(1) Find minimum and maximum element for following set

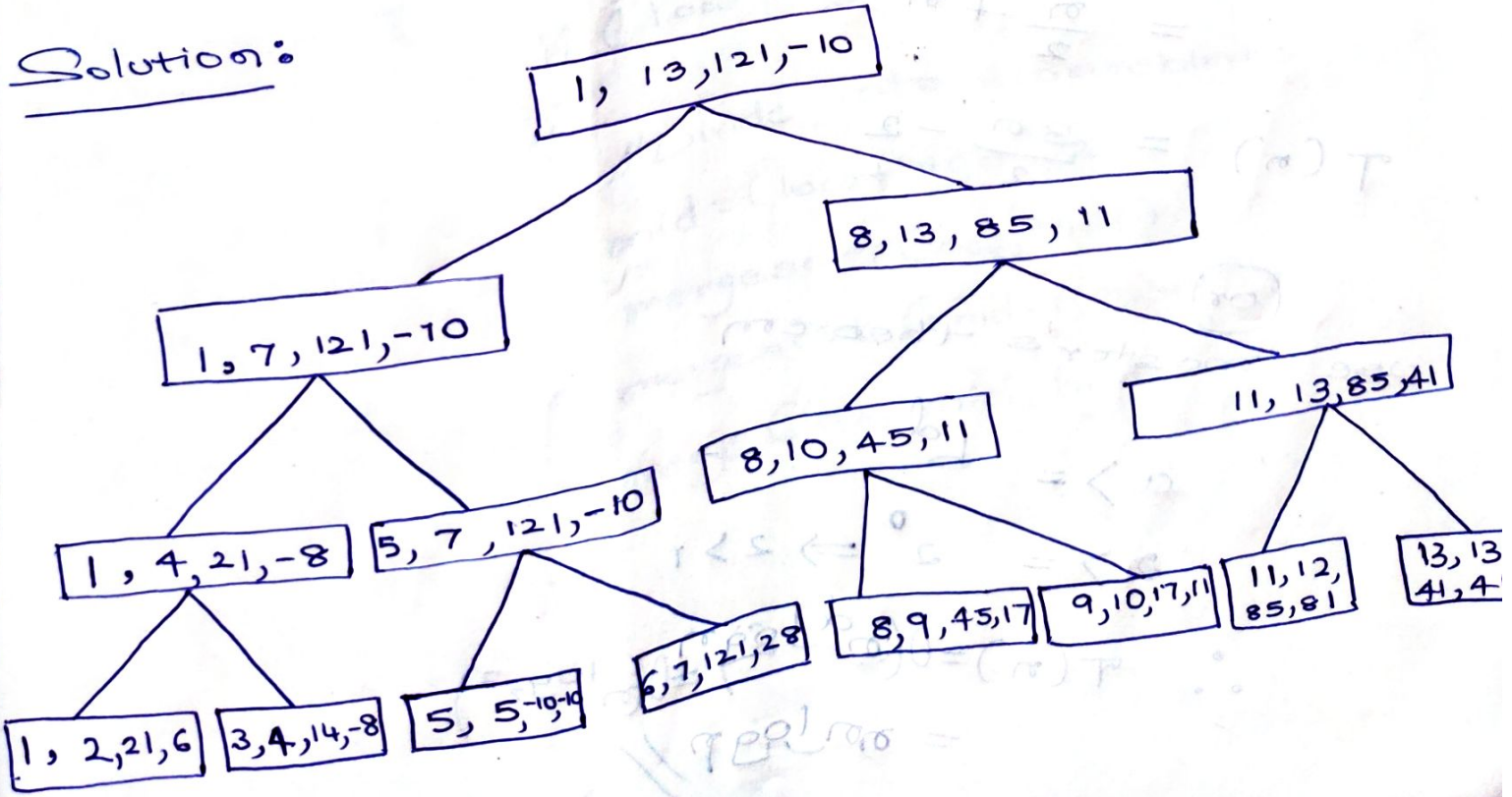
eg: 21, 6, 35, 42, -3, 18, 48, 16, 7, 10, 28, 100, 41

Solution: mid → $\frac{1+13}{2} = 7$ // (mid)



ex (2): Find minimum and maximum element for following set → 12, 16, 14, -8, -10, 121, 28, 45, 17, 11, 81, 85, 64, 29, 41 using min-max algorithm.

Solution:



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

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

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

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

$$= 2^{K-1} T(2) + \sum_{i=1}^{K-1} 2^i$$

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

Assume $n = 2^K$ for some positive integer

$$= \frac{2^K}{2} + 2^K - 2$$

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

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

(or)

use master's theorem

$a > b^d$

$2 > 2^0 \Rightarrow 2 > 1$

$\therefore T(n) = O(n^{\log_b a}) = O(n^{\log_2 2})$

$\therefore = O(n)$ //

If n is power of 2,

Best case = Worst case = Average case

time complexity $T(n) = \frac{3n}{2} - 2$ (or) $n \log n$ //

3. Merge sort

- Given a sequence of elements, the idea is to split into 2 sets → a[1] --- a[n]
  - a[1] --- a[n/2]
  - a[n/2 + 1] --- a[n]
- Each set is individually sorted and the resulting sorted sequence are merged to get a single sorted sequence.

Algorithm:

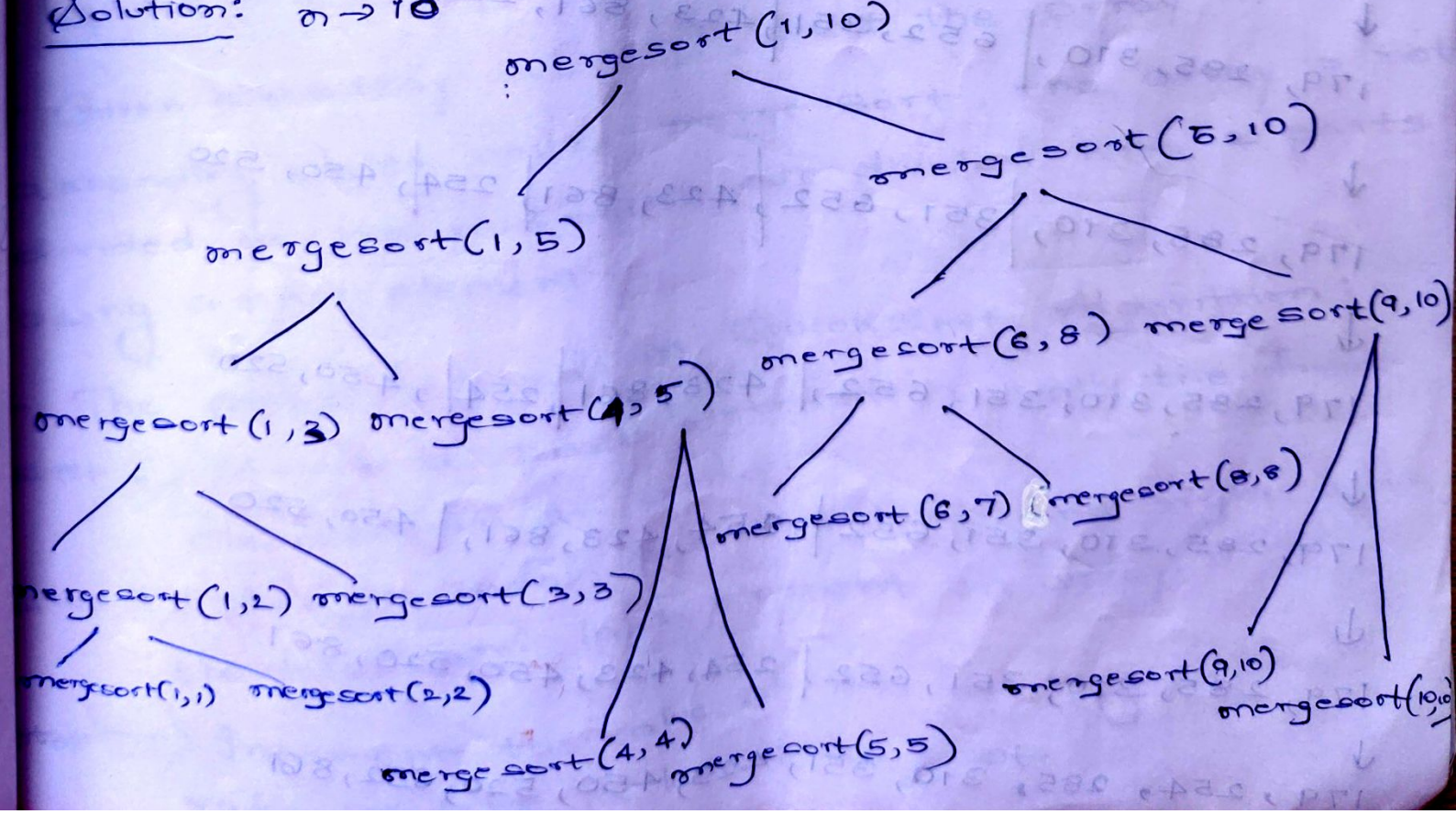
```
Algorithm Mergesort (low, high)
{
    if (low < high)
    {
        //Divide into subproblems
        mid = (low + high)/2
        mergesort(low, mid)
        mergesort (mid+1, high)
        merge (low, mid, high)
    }
}
```

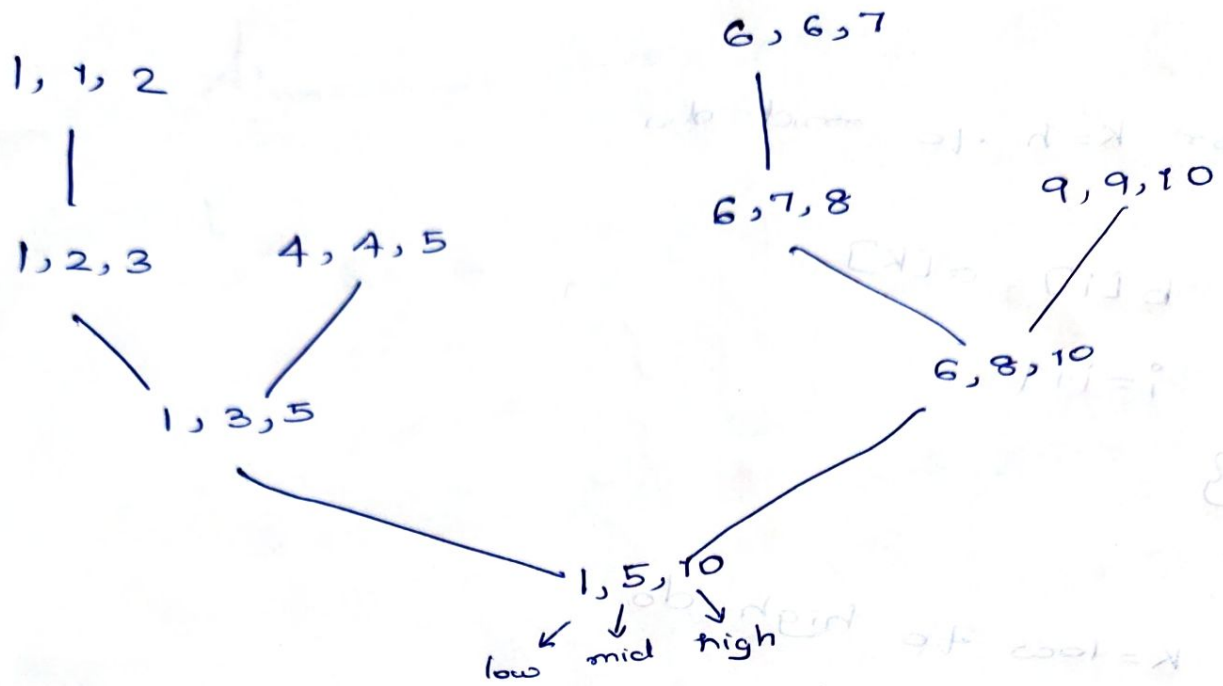
```
Algorithm Merge (low, mid, high)
// a[] -> actual array   b[] = auxillary array
{
    h = low;
    i = low;
    j = mid+1
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {   b[i] = a[h];
            h = h+1
        }
        else
        {
            b[i] = a[j];
            j = j+1;
        }
        i = i+1
    }
    if (h > mid) then
    [
        for K = j to high do
        {   b[i] = a[k]
            i = i+1
        }
    ]
}
```

```
else
{
    for k=h to mid do
    {
        b[i] = a[k]
        i=i+1
    }
}
for k=low to high do
    a[k] = b[k]
}
```

Ex1: Perform merge sort on following set of data → 310, 285, 179, 652, 351, 423, 861, 254, 450, 520

Solution: n → 10





h → ↓mid → j

310, 285, 179, 652, 351, 423, 861, 254, 450, 520

a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10]

285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520

179, 285, 310, | 652, 351 | 423, 861, 254, 450, 520

179, 285, 310, | 351, 652 | 423, 861 | 254 | 450, 520

179, 285, 310, 351, 652, | 423, 861 | 254 | , 450, 520

179, 285, 310, 351, 652 | 254, 423, 861, | 450, 520

179, 285, 310, 351, 652 | 254, 423, 450, 520, 861

179, 254, 285, 310, 351, 423, 450, 520, 652, 861

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

when $n = 2^k$,

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

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

$$a = 2,\ b = 2,\ d = 1$$

$$a = b^d$$

$$2 = 2^1$$

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

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

Quick Sort:

- Given an array $a[1..n]$, sort the array in ascending order unlike merge sort, the array is not divided at the mid point but divided into 2 parts using a pivot element.

- The steps involved in QuickSort Algorithm:

Step I) Select a pivot element usually the first element in an array.

Step II) Assign index position left and right with the first and last element in array.

Step III) Increment the value of left as long as the value at left is less than pivot.

Step IV: Decrement right as long as the value at right is greater than pivot.

Step V: If left is less than right interchange the values at left and right.

Step VI: Repeat step III till left is greater than right

Step VII: Interchange value at right with pivot.

```
Algorithm Partition ( a, m, p)
{
    r = a[m]
    i = m
    j = p
    repeat
    {
        repeat
        {
            i = i+1;
        }
        until(a[i] ≥ v);
        repeat
        {   j = j-1;
        }
        until (a[j] ≤ v);
        if (i<j) then
            interchange (a, i, j)
    }until (i ≥ j);
```

```
    a[m] = a[j]
    a[j] = v
    return j;
}
```

```
Algorithm Interchange (a, i, j)
{
    p = a[i]
    a[i] = a[j]
    a[j] = p
}
```

```
Algorithm
  QuickSort(p,q
{
  if (p<q)
  {
  j=partition
  (a, p, q+1)
  Quicksort(p
      j-1)
  Quicksort(
    j+1, q)
  }
}
```

Ex: Perform Quicksort on the following dataset

211, 108, 115, 68, 28, 111, 95,

Solution: pivot → 211

i → 2 j → 7

i → 2 3 4 5 6 7

j → 8 7

a[] → 211, 108, 115, 68, 28, 111, 95
↓p ↓j

a[] → 95, 108, 115, 68, 28, 111, 211
↓j

a[7]

Q.S(1, 6)

Q.S(8, 7)

i > j

stop

95, 108, 115, 68, 28, 111

↓

P

i → 2

j → 7 6 5

95, 28, 115, 68, 108, 111

i → 3

j → 5 4

95, 28, 68, 115, 108, 111

j i

a[3]

68, 28, 95, 115, 108, 111

j

Qs(1, 2)

68, 28

↓

P

i → ✗ 2

j → 2

28, 68

a[2]

Q.s(1, 1)

28 → a[1]

Q.s(4, 6)

115, 108, 111

↓

b

i → 4, 5, 6

j → 6

111, 108, 115

j

a[6]

Q.s(4, 5)

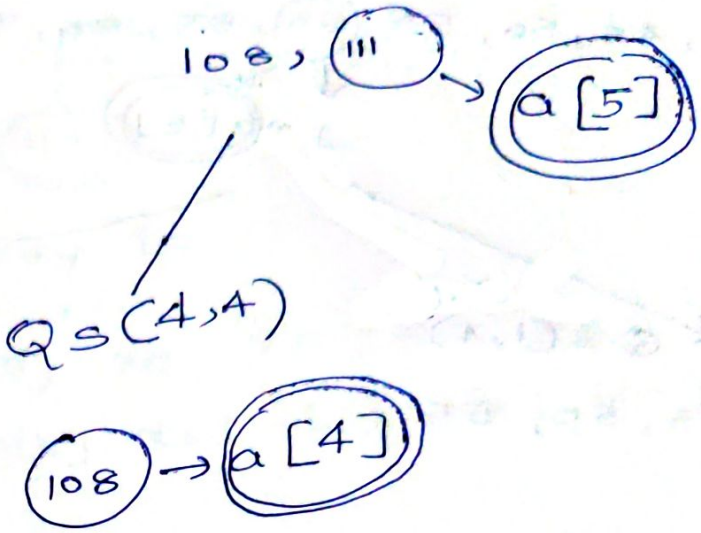
111 → 108

↓

b

i → 4 5

j → 5



∴ The sorted array is

28, 68, 95, 108, 111, 115

Average Time Complexity of Quick sort $T(n) = n \log n$

Worst Case time Complexity $T(n) = n^2$

Perform Quicksort on the following set of ments

65, 70, 75, 80, 85, 60, 55, 50, 45

1 2 3 4 5 6 7 8 9

lution: pivot → 65

i = 1

j = 9+1 = 10

65, 45, 75, 80, 85, 60, 55, 50, 70

65, 45, 50, 80, 85, 60, 55, 75, 70

65, 45, 50, 55, 85, 60, 80, 75, 70

65, 45, 50, 55, (60), 85, 80, 75, 70

↓ P j

60, 45, 50, 55, (65), 85, 80, 75, 70

↓ j a[5]

Q.S(1,4)

60, 45, 50, 55

↓ P

i → 1, ~~2~~, ~~3~~, 4

j → ~~3~~, 4

Q.S(6,9)

85, 80, 75, 70

↓ P

i → ~~6~~, ~~7~~, 8

j → ~~9~~ 8

55, 45, 50, (60) → a[4]

j

Q.S(1,3)

55, 45, 50

↓ P

i → ~~1~~ 3

j → ~~4~~ 3

Q.S(5,4)

i > j

stop

70, 80, 75, (85) → a[9]

j

Q.S(6,8)

70, 80, 75

i → ~~5~~ 6 7

j → ~~7~~ ~~8~~ ~~7~~, 6

QS(9,8)

i > j

stop

a[6] 70, 80, 75

↓ j

50, 45, (55) → a[3]

j

Q.S(1,2)

50, 45

↓ P

QS(4,3)

i > j

Stop

Q.S(6,5)

i > j

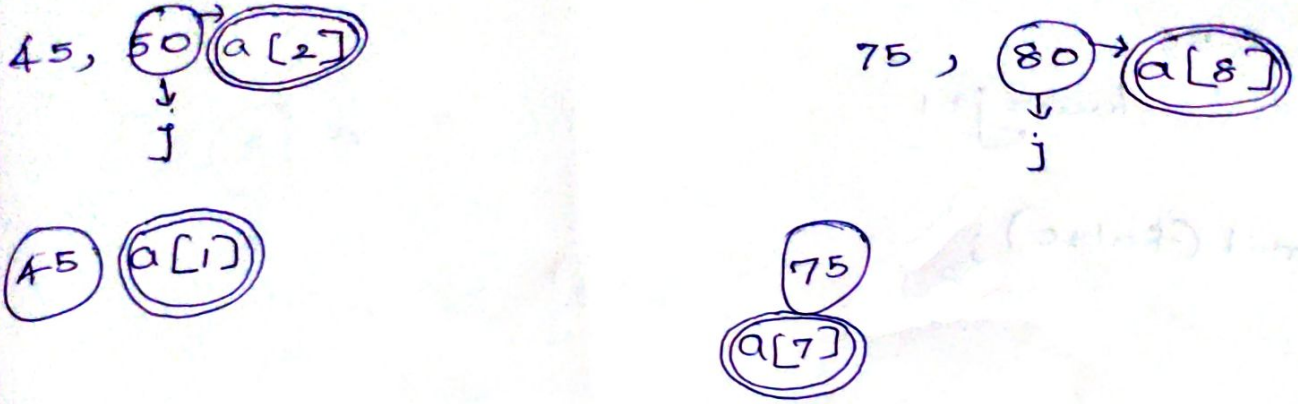
stop

Q.S(7,8)

80, 75

↓ P

i → ~~7~~ 8



∴ The sorted array is

45, 50, 55, 60, 65, 70, 75, 80, 85
a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

Selection Algorithm:

→ The problem is to find k^{th} smallest element in an array.

```
Algorithm Select1(a, n, k)
{
    low = 1
    up = n+1
    a[n+1] = ∞
    repeat
    {
        j = Partition(a, low, up)
        if (k=j) then
            return
        else if (k<j) then
            up = j
```