UNIT-1

Introduction

→ Algorithm is a greek word

→ Algorithm is a finite set of instructions i.e, followed accomplishes a particular task

(or)

→ Basically algorithm is a finite set of instructions that can be used to perform some task.

(or)

→ The algorithm is defined as collection of unambigious instructions occuring in specific sequence so that the algorithm output is produced for the given finite set of inputs in the specific amount of time.

* The word algorithm comes from the name of persian author "Abu Jafar mohammad ibn musa al khowazmi"

Properties (or) characteristics of an algorithm

Input: 0 (or) more inputs are supplied.

Output: Atleast one output is produced.

Definition: Each instruction is clear and unambigious.

Finiteness: An algorithm is a finite that means algorithm is terminated after finite no. of steps.

Effectiveness: Every instruction must be very basic it must be feasible.

Algorithm development Steps :

1) How to device an algorithm : writing algorithm using best design techniques

2) How to validate an algorithm : check for an algorithm whether it is working correctly or not by manually.

3) How to analyse algorithm : It refers to task of determining how much computing time and storage an algorithm requires.

4) How to test programs : Testing of programs consists of two phases.
   i) Debugging
   ii) Profiling (or) performance measurement

i) Debugging : It is the process of executing the programs on Sample data sets to determine whether faulty result occur and if so correct them.

ii) Profiling : It is the process of executing corect program on data sets and measuring the time and space it takes to compute result.

Pseudo code for expressing algorithm:

1) Comments begin with // and continue until the end of the line.

2) Blocks are indicated with matching braces { and }. Statements are delimited by ;

3) An identifier begins with a letter

4) Assignment of values to the variables is done using assignment Statement.

Syntax: x = 10;

<variable> := <expression>;

Ex: x := 10

5) They are two boolean values T and F. Inorder to produce these values the logical operators AND, OR, NOT and relational operators >, <, ≤, ≥, =, ≠ are provided.

6) Elements of multi-dimensional arrays are accessed using [and].

7) The following looping Statements are provided

FOR, WHILE, DOWHILE, REPEAT, UNTIL.

Syntax:

The general form of FOR loop is

```
for variable := value1 to value2
 step do
 {
 Statement 1
   :
   :
 Statement n
 }
```

Ex:

```
for i := 1 to n step do
 {
 Statement 1
   :
 Statement n
 }
```

The general form of while loop is

```
while <condition> do
 {
 Statement 1
    :
 Statement n
 }
```

The general form of repeat until statement

repeat

<statement 1>

⋮

<statement n>

until <condition>

8) A conditional statement has the following forms

if <condition> then <statement>

if <condition> then <statement>

else <statement>

9) Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.

10) An algorithm consists of heading and body. The heading takes the form

Algorithm Name <parameter list>.

write an algorithm to count the Sum of n numbers

```
Algorithm Sum(1,n)
// Problem description: This problem is for finding
   the Sum of given n numbers
// Sum of the given n numbers
// input : 1 to n numbers
// output: The Sum of n numbers

    result ← 0
    for i ← 1 to n do ; ← i+1
    result ← result + i
    return result.
```

write an algorithm to check whether the given number is even or odd.

```
// Problem description : To check whether the given
   number is even or odd.
// input : 1 to n numbers
// output : Even (or) odd

    if (n % 2 == 0) then
    write ("num even");
    else
    write ("num odd");
```

write an algorithm that finds and returns the maximum of n given numbers

```
// Problem description: This is the algorithm finding the maximum number of given n numbers

Algorithm Max(A,n)
// A is an array of size n
{
Result := A[1];
for i:=2 to n do
if A[i] > Result then Result i= A[i];
return result;
}
```

Recursion

Definition: A recursion function is a function that is defined in terms of itself, Similarly an algorithm is said to be recursive if the same algorithm is invoked in the body.

Ex: let $T(n) = \begin{cases} d & \text{if } n = 1 \\ c + T(n-1) & \text{if } n > 1 \end{cases}$

$n = 1 \Rightarrow T(1) = d$

$n = 2 \Rightarrow T(2) = c + T(2-1)$

$= c + T(1)$

$= c + d$

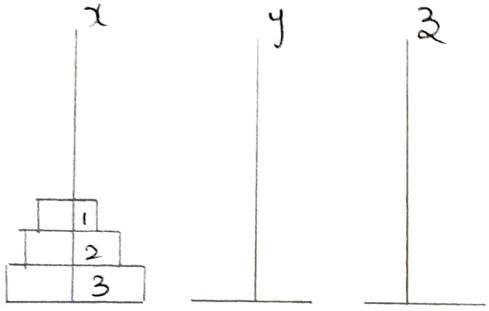
$n = 3 \Rightarrow T(3) = c + T(3-1)$

$= c + T(2)$

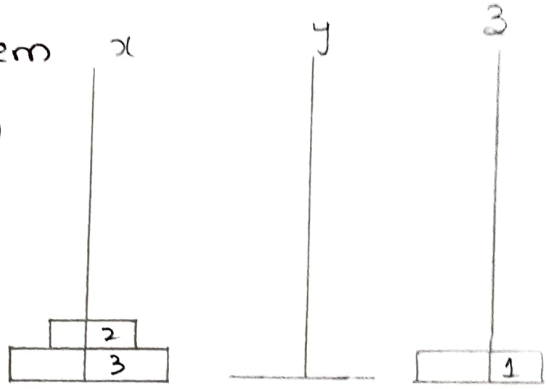
$= 2c + d$

Towers of Hanoi Problem

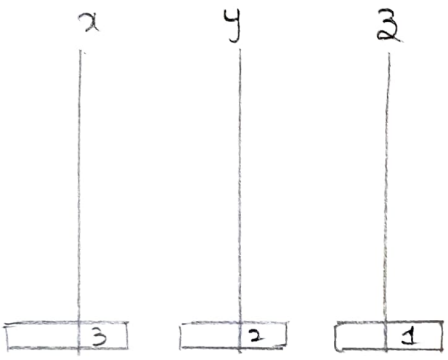
1)



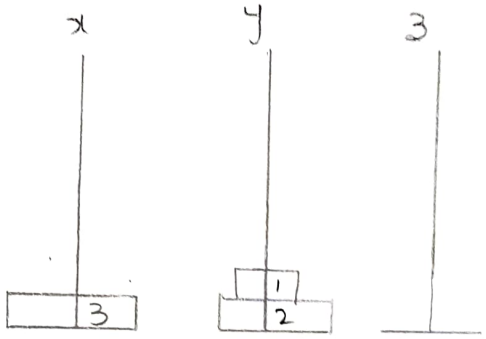
2)



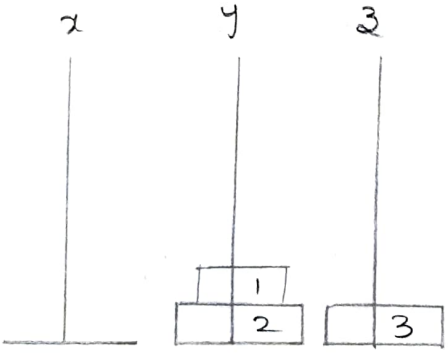
3)



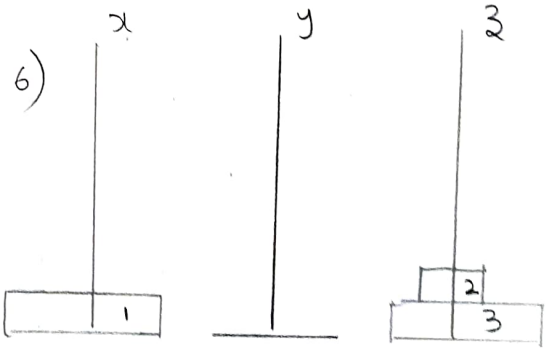
4)



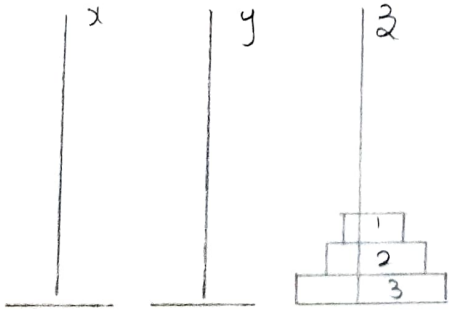
5)



6)



7)



The reccurence relation for the towers of hanoi is $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n-1)+1 & \text{if } n>1 \end{cases}$

$n=1$ $T(1)=1$

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

$= 2T(1)+1$

$= 2(1)+1$

$= 3$

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

$= 2T(2)+1$

$= 2(3)+1$

$= 6+1$

$= 7$

$n=4$ $T(4) = 2T(4-1)+1$

$= 2T(3)+1$

$= 2(7)+1$

$= 15$

$n=5$ $T(5) = 2T(5-1)+1$

$= 2T(4)+1$

$= 2(15)+1$

$= 31$

$n=6$ $T(6) = 2T(6-1)+1$

$= 2T(5)+1$

$= 2(31)+1$

$= 63$

```
Algorithm Towers of Hanoi (n,x,y,z)
// Move the top n disks from tower x to towery
   {
   if (n≥1) then
   {
   Towers of Hanoi (n-1,x,z,y);
   write ("move top disk from tower", x, " to top
                          of tower", y);
   Towers of Hanoi (n-1,z,y,x);
     }
   }
```

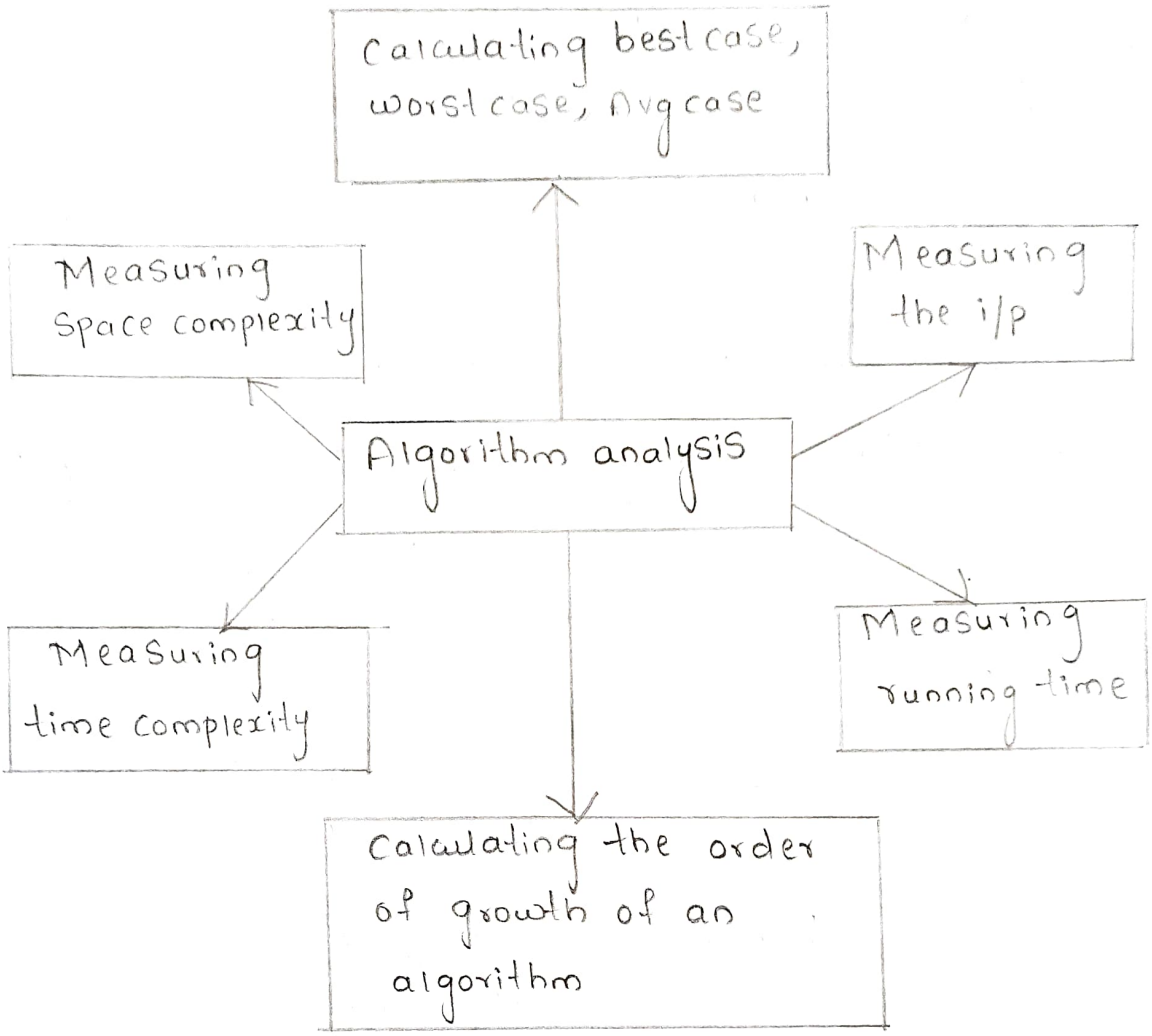
Performance Analysis

→ The efficiency of an algorithm can be measured with performance of the algorithm.

→ Basically performance of an algorithm can be measured with two factors.

1) Time complexity
2) Space complexity

Algorithm Analysis



1) **Space complexity**: The Space complexity of an algorithm is the amount of memory it needs to run to completion.

→ Space complexity can be measured using

$S(P) = C + SP$

→ where C is a constant and it is a fixed ~~path~~ part and is used to calculate Space of the input and output.

→ SP is the instance characteristics it is a variable ~~path~~ part that depends upon instances.

Ex: Algorithm addition of three numbers

```
Algorithm addition of three numbers
// This algorithm for calculating three numbers
// Input : values of x,y,z
// output : Sum of x,y,z
   {
    x+y+z;
   }
```

S(P) = C+SP

= 3+0

= 3

In the above algorithm we have three constants x,y,z and assume that each constant occupy one word size so. the total memory require is 3 words.

Ex: write an algorithm for expression Simplification

```
Algorithm expression Simplifying (a,b,c)
{
return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

S(P) = C+SP

= 3+0

= 3

Ex: write an algorithm for Sum of n numbers

Algorithm for Sum of n numbers (1 to n, S, i)

// This algorithm for Sum of n numbers

// Input : 1 to n numbers

// Output : Sum of n numbers

```
{
 S:=0;
 for i:=1 to n do
     S:= S+a[i];
     returns;
 }
```

S(P) = C+SP

Constant = S), i, n=3

Space = n

S(P) = 3+n

if n=5 then S(P) = 3+5

= 8

2) Time complexity: The time complexity can be defined as the amount of time required by an algorithm to execute or run.

→ we have two methods for measuring the time complexity of an algorithm

i) Count method
ii) Step-count method

i) Count method : In this method we are counting the each and every instruction in an algorithm.

→ Every time the instruction is incremented by '1'.

→ Count value can be measured by

Count = Count + 1

Ex:

```
Algorithm Sum(A,n)
{
S:=0;
for i=1 to n do
S=S+n[i];
return s;
}
```

ii) Step-count method : In this method we are evaluating the steps per execution, frequency and Total.

→ In steps per execution we are counting the no. of instructions in a given algorithm.

→ In frequency we are counting the no. of times that instruction going to be execute.

→ Total is the multiplication of steps per count and frequency.

Note: For name of algorithm, begin, end, else in these cases the count value is zero

	S/e	Frequency	Total
Algorithm Sum(A,n)	0	0	0
{	0	0	0
S:=0;	1	1	1
for i=1 to n do	1	n	n+1
S= S+n[i];	1	n	n
return s;	1	1	1
}	0	0	0

Ex:2

	S/e	Frequency	Total
Algorithm Add (a,b,c,m,n)	0	0	0
{	0	0	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	m(n+1)
c[i,j] = a[i,j]+b[i,j];	1	mxn	mxn
}	0	0	0
			2mn+2m+1

Ex:3

	S/e	Frequency	Total
Algorithm mul (a,b,c,n)	0	0	0
{	0	0	0
for i:=1 to n do	1	n+1	n+1
for j=1 to n do	1	n(n+1)	n^2+n
{	0	0	0
c[i,j]=0;	1	$nxn=n^2$	n^2
for k=1 to n do	1	$(n+1)n^2$	n^3+n^2
{	0	0	n^3
c[i,j]=c[i,j]+a[i,k]*b[k,j];	1	n^3	0
}	0	0	0
}	0	0	0
}	0	0	
			$2n^3+3n^2+2n+1$

write an algorithm to find out the fibonocci series of Ex:4 given numbers & find out the time complexity.

	S/e	Frequency n=1 or n<=1	n>1	Total n=1 or n<=1	n>1
Algorithm fib(n)	0	0	0	0	0
{	0	0	0	0	0
if (n≤1) then	1	1	1	1	1
write (n);	1	1	0	1	0
else	0	0	0	0	0
{	0	0	0	0	0
$f_2=0$; $f_1=1$;	1	0	1	0	0
for i=2 to n do	1	0	n	0	2 n
{	0	0	0	0	0
$F=F_1+F_2$;	1	0	n-1	0	n-1
$F_2=F$; $F_1=Fn$;	2	0	n-1	0	2n-2
}	0	0	0	0	0
write (f_n);	1	0	1	0	1
}	0	0	0	0	0
}	0	0	0	0	0
				2	4n+1

Note:

For i:=1 $\Rightarrow$ n+1

i=2 to n $\Rightarrow$ n+1-1=n

i=0 to n $\Rightarrow$ n+1

For above algorithm time complexity is 4n+1 & if n>1 and n=1 or n<1 is 2

Asymptotic Notations

To choose the best algorithm we need to check the efficiency of algorithm this efficiency can be measured by computing the time complexity of each algorithm.

→ Asymptotic Notation provide short hand way to represent time complexity

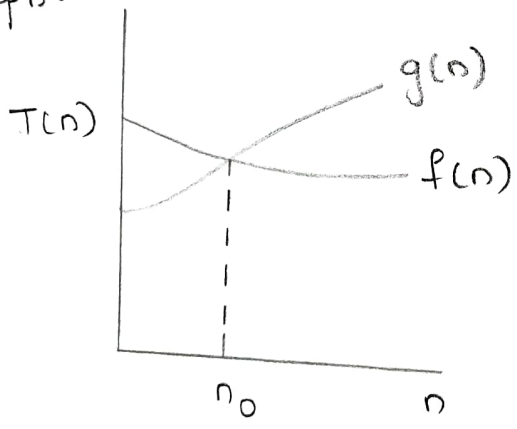
→ Various notations such as Big O, Omega Ω, Theta Θ notations which are called as Asymptotic notations.

i) Big 'O' notation:

The function $f(n) = O(g(n))$ if and only if there exist a positive constants c and n_0 such that

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

Graph:



This notation provide an upperbound for the function f i.e the function g(n) is an upperbound on the values of f(n) for all $n, n \geq n_0$.

Ex 1) let function $f(n) = 3n+2$ and $g(n) = 4n$ then show that $f(n) = Og(n)$

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

$$3n+2 \leq 4n$$

$n=0$ $\quad 3(0)+2 \leq 4n = 2 \leq 4(0) = F$

$n=1$ $\quad 3(1)+2 \leq 4(1) = 5 \leq 4 = F$

$n=2$ $\quad 3(2)+2 \leq 4(2) = 8 \leq 8 = T$

$n=3$ $\quad 3(3)+2 \leq 4(3) = 11 \leq 12 = T$

$n=4$ $\quad 3(4)+2 \leq 4(4) = 14 \leq 16 = T$

The function $f(4n)$ is upper bound to the function $f(3n+2)$ when $n \geq 2$

$\therefore f(n) = Og(n)$

Ex:2) let $3n+3 = O(n)$ as $3n+3 \leq 4n \; \forall n \geq 3$

$$3n+3 = O(n)$$

$$3n+3 \leq 4n$$

$n=3$ $\quad 12 \leq 12 \quad - T$

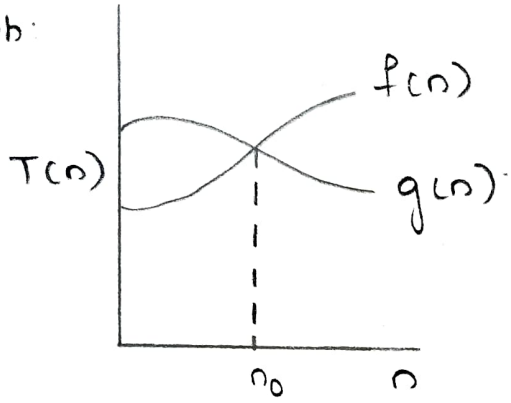
$n=4$ $\quad 15 \leq 16 \quad - T$

$n=5$ $\quad 18 \leq 20 \quad - T$

$\therefore f(n) = Og(n)$

2) Omega 'Ω' notation: The function $f(n) = \Omega g(n)$ if and only if there exist a positive constants c and n_0 such that $f(n) \geq c * g(n)$ $\forall n, n \geq n_0$.

Graph:



This notation provides a lower bound for the function f i.e the function g(n) is a lower bound on the values of $f(n)$ $\forall n, n \geq n_0$.

Ex1) The function $3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$.

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

$3n+2 \geq 3n$

$n=1$ $5 \geq 3$ - T

$n=2$ $8 \geq 6$ - T

$n=3$ $11 \geq 9$ - T

The function $3n$ is lower bound to the function $3n+2$ when $n \geq 3$.

Ex 2) let the function $f(n) = 2n^2+5$ and $g(n) = 7n$ find $f(n) \geq c * g(n)$

$2n^2+5 \geq 7n$

$n=0 \quad 5 \geq 0 - T$

$n=1 \quad 7 \geq 7 - T$

$n=2 \quad 13 \geq 14 - F$

$n=3 \quad 23 \geq 21 - T$

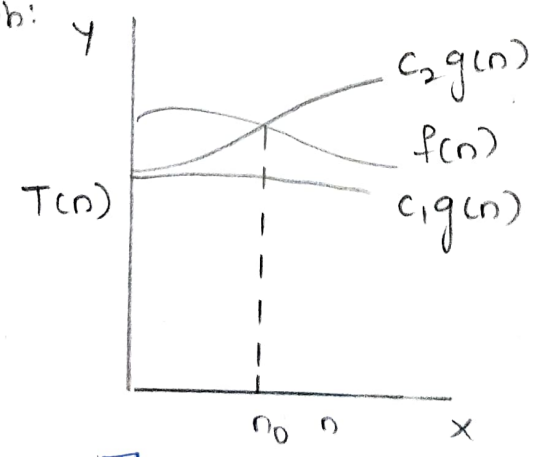
$n=4 \quad 37 \geq 28 - T$

$n=5 \quad 55 \geq 35 - T$

The function $7n$ is lower bound for the function $2n^2+5$ where $n \geq 3$

3) Theta 'Θ' Notation : The function $f(n) = \Theta(g(n))$ if and only if there exist a positive constants c_1, c_2 and n_0. Such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall n, n \geq n_0$

Graph:



This notation provides both lower and upper bound for the function $f(n)$ i.e $g(n)$ is both lower and upper bound on the values of $f(n)$. In other words Θ notation says that $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$ $\forall n, n \geq n_0$.

Ex: The function $3n+2 = \Theta(n)$ as $3n+2 \geq 3n \ \forall n \geq 2$ and $3n+2 \leq 4n \ \forall n \geq 2$ find c_1, c_2 and n_0.

$3n+2 \geq 3n \ \forall n \geq 2$

$n=2 \quad 8 \geq 6 \quad - T$

$n=3 \quad 11 \geq 9 \quad - T$

$n=4 \quad 14 \geq 12 \quad - T$

$c_1 = 3$

$3n+2 \leq 4n \ \forall n \geq 2$

$n=2 \quad 8 \leq 8 \quad - T$

$n=3 \quad 11 \leq 12 \quad - T$

$n=4 \quad 14 \leq 16 \quad - T$

$c_2 = 2$

$3n \leq 3n+2 \leq 4n$

$n=2$

$3 \cdot 2 \leq 3 \cdot 2 + 2 \leq 8$

$6 \leq 8 \leq 8$

Little o notation : The function $f(n) = o(g(n))$ if and only if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

Little ω notation : The function $f(n) = \omega(g(n))$ if and only if $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

Typical running times of algorithm:

$O(1)$ - The running time of algorithm is constant.

$O(n)$ - It means linear, typically achieved by examining each element in the input once.

$O(n^2)$ - It means quadratic, typically achieved by examining all pairs of data elements

Ex: Selection Sort

$O(n^3)$ - It means cubic, often achieved by combining algorithms with a mixture of previous running types.

Ex: Matrix multiplication

$O(2^n)$ - It means exponential, any exponential function with a base strictly greater than one grows faster than any polynomial function n^3 or n^2.

$O(\log n)$ - It means logarthamatic typically achieved by dividing the problems into Smaller Segments and only working at one input element in each segment

Ex: Binary Search.

$O(n \log n)$ - It means linear logarithamatic, typically achieved by dividing the problems into sub-problems, solving the sub-problems independently and then combining the result

Ex: Merge Sort

Divide and Conquor:

Divide and conquor algorithm consists of 3 steps

(i) Dividing the problem into several sub problem that are similar to the original problem but small in size.

(ii) Solve the subproblems recursively or independently.

(iii) Combine those solutions to sub-problems to create a solution to the original problem.

Control abstraction or general method for divide & conquor

```
Algorithm D and C(P)
{
if Small(P) then return S(P);
else
{
divide P into Smaller instances P1, P2 ... Pk, k≥1;
Apply D and C to each of these Subproblems;
return. Combine (D and C(P1), D and C(P2), ...
  D and C(Pk));
}
}
```

→ D and C is initially invoked as D and C(P) whose P is the problem to be solved.

→ S(P) is a boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting.

→ Otherwise the problem 'P' is divided into smaller subproblems these subproblems $P_1, P_2 \cdots P_k$ are solved by recursive applications of D&C.

→ Combine is a function that determines solution to P using the solutions to the k-subproblems

→ If the size of 'P' is 'n' and the size of k subproblems are $n_1, n_2 \cdots n_k$ respectively then the computing time of D and C is described by the reccurence relation.

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + (n_k) & \\ T(n) & \text{otherwise} \end{cases}$$

where T(n) is the time for D and C on only input size n and g(n) is the time to compute the answer directly for small inputs.

The functions f(n) is the time for dividing P and combining the solutions.

The complexity of many divide and conquor is given by the recurrence of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ T(n/b)+F(n) & n>1 \end{cases}$$

where a, b are known constants. we assume that $T(1)$ is known and n is power of

i.e $n = b^k$

let $a=2, b=2, T(1)=2, F(n)=2$

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

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

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

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

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

$$= Tn+3n$$

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

In general we see that $T(n) = 2^i T(n/2^i)+in$

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

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

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

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

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

$= T(n)+3n$

Recurrence Relation

The recurrence equation is a equation that defines sequence recursively.

→ It is normally in the following form

$$T(n) = T(n-1) + n, \quad n > 0$$

$$T(0) = 0$$

1st equation is a recurrence relation.

2nd equation is a Intialization.

Solving recurrence relation equation

i) Substitution method

ii) Masters method

i) Substitution Method

a) Forward Substitution

b) Backward Substitution

a) Forward Substitution

This method makes use of initial condition in the form of initial term and value for next term is generated

This process is continued until Same formula is generated.

Eg: Recurrence relation equation is $T(n) = T(n-1) + n$

Initial condition is $T(0) = 0$

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

$= T(0)+1$

$= 0+1$

$= 1$

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

$= T(1)+2$

$= 1+2$

$= 3$

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

$= T(2)+3$

$= 3+3$

$= 6$

By observing the above generated equation we can derive a formula $T(n) = \frac{n(n+1)}{2}$

$T(1)=1, T(2)=3, T(3)=6$

we can also denote 'n' in terms of Big O notation as follows:

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

b) Backward Substitution Method

In this method backward values are Substitued recursively in order to derive Some formula

Eg: $T(n) = T(n-1)+n$ —①

$T(0)=0$

$n = n-1$

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

$n = n-2$

$$T(n-2) = T(n-2-1) + n-2 \quad \text{---(3)}$$
$$= T(n-3) + n-2$$

eq (2) substitue in eq (1)

$$T(n) = T(n-1) + n$$
$$= \cancel{T(n-2) + (n-1) + n}$$
$$= T(n-2) + n-1 + n$$
$$= T(n-2) + 2n-1$$
$$= T(n-3) + n-2+2n-1$$
$$= T(n-K) + (n-K+1) + (n-K+2) + n$$
$$= T(n-3) + 3n-3$$

$K = n$

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

By observing above value we can guess the formula as, $T(n) = \frac{n(n+1)}{2}$

The time complexity in Big 'O' notation is $O(n^2)$.

ii) Masters method Consider the recurrence relation $T(n) = aT(n/b) + f(n)$

where $n \geq d$, b is constant

→ Then the masters theorem can be stated for efficiency analysis as if $f(n)$ is $\theta(n^d)$ where $d \geq 0$. in the recurrence relation equation

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

case ii) - $T(n) = \theta(n^d \log n)$ if $a = b^d$

case iii) - $T(n) = \theta(n \log_b a)$ if $a > b^d$

Ex:-i) $T(n) = 4T(n/2) + n$

$a = 4, b = 2, d = 1$

$4 > 2^1$

$T(n) = \theta(n \log 2^4)$

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

$= \theta(n^2)$

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

$a = 4, b = 2, d = 2$

$4 = 2^2$

$4 = 4$

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

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

(iii) $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2, d = 3$

$4 < 2^3$

$4 < 8$

$T(n) = \theta(nd) = \theta(n^3)$ //.

For quick & easy calculations of logarithmic values to base the following table can be measured.

$$k = \log_2 m$$

m	k
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8

Applications of divide and conquer:

Divide and conquer strategy can be applicable in various applications they are

i) Binary Search

ii) Merge Sort

iii) quick Sort

iv) Strassen's matrix multiplication etc.

i) **Binary Search** Binary Search is an efficient Searching method

Binary Search algorithm is a technique for finding a particular element or value in a sorted list.

→ Binary Search algorithm is a technique for finding a particular element or value in a sorted list.

→ Binary Search algorithm works based on divide and conquer strategy.

Eg: 5 6 9 24 36 48 52 100 150 180 200

1) [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

$$Mid = \frac{first + last}{2}$$

$$= \frac{0+10}{2} = 5 \qquad 48 < 100$$

low = mid + 1

Step 2: 52 100 150 186 201

6 7 8 9 10

low = mid + 1

$$mid = \frac{6+10}{2} = 8$$

150 > 100

Step 3: high = mid - 1

$$mid = Int\left(\frac{6+7}{2}\right) = 6$$

52 < 100

Step 4: $mid = \frac{7+7}{2} = 7$

100 = 100

Algorithm of Binary Search

```
Algorithm BinSearch(a,n,x)
// Given an array a[1:n] of elements in nondecreasing
//order, n≥0, determine whether x is present, and
// if so, return j such that x=a[j]; else return 0
 {
  low:=1; high:=n;
  while (low ≤ high) do
   {
   mid:= [(low+high)/2];
   if (x< a[mid]) then high:=mid-1;
   else if (x>a[mid]) then low:=mid+1;
   else return mid;
    }
   return 0;
    }
```

→ The computing time of Binary search is

$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Best case	Avg case	worst case

ii) **Merge Sort:** Merge Sort is another application of divide and conquer strategy. It contains following steps

→ Divide the sequence into two sequences of length $n/2$ and $n/2$.

→ Recursively Sort each of two subsequences

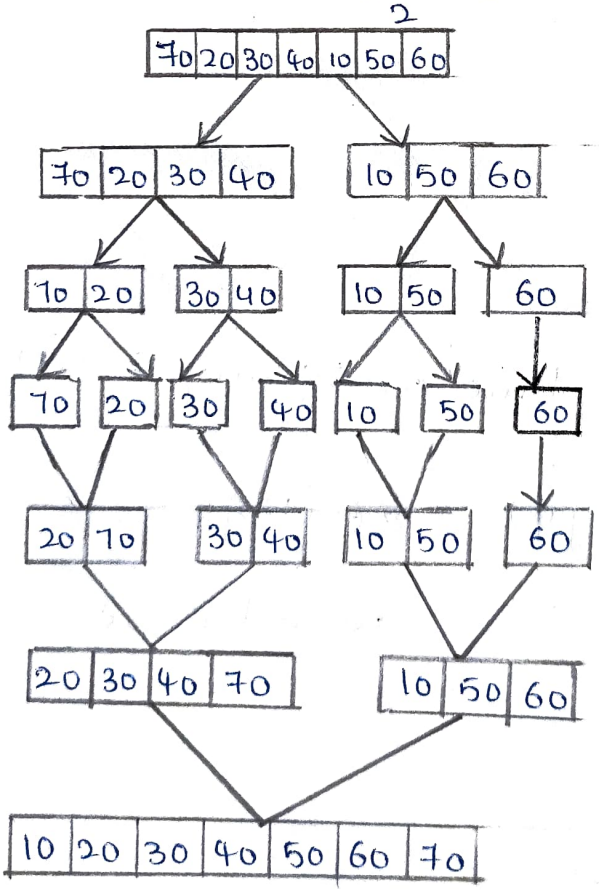
→ Merge the sorted subsequences to obtain the final result

Ex: Sort the following elements using merge sort

70	20	30	40	10	50	60
[0]	[1]	[2]	[3]	[4]	[5]	[6]

$$mid = \frac{low + high}{2}$$

$$= \frac{0+6}{2} = 3$$

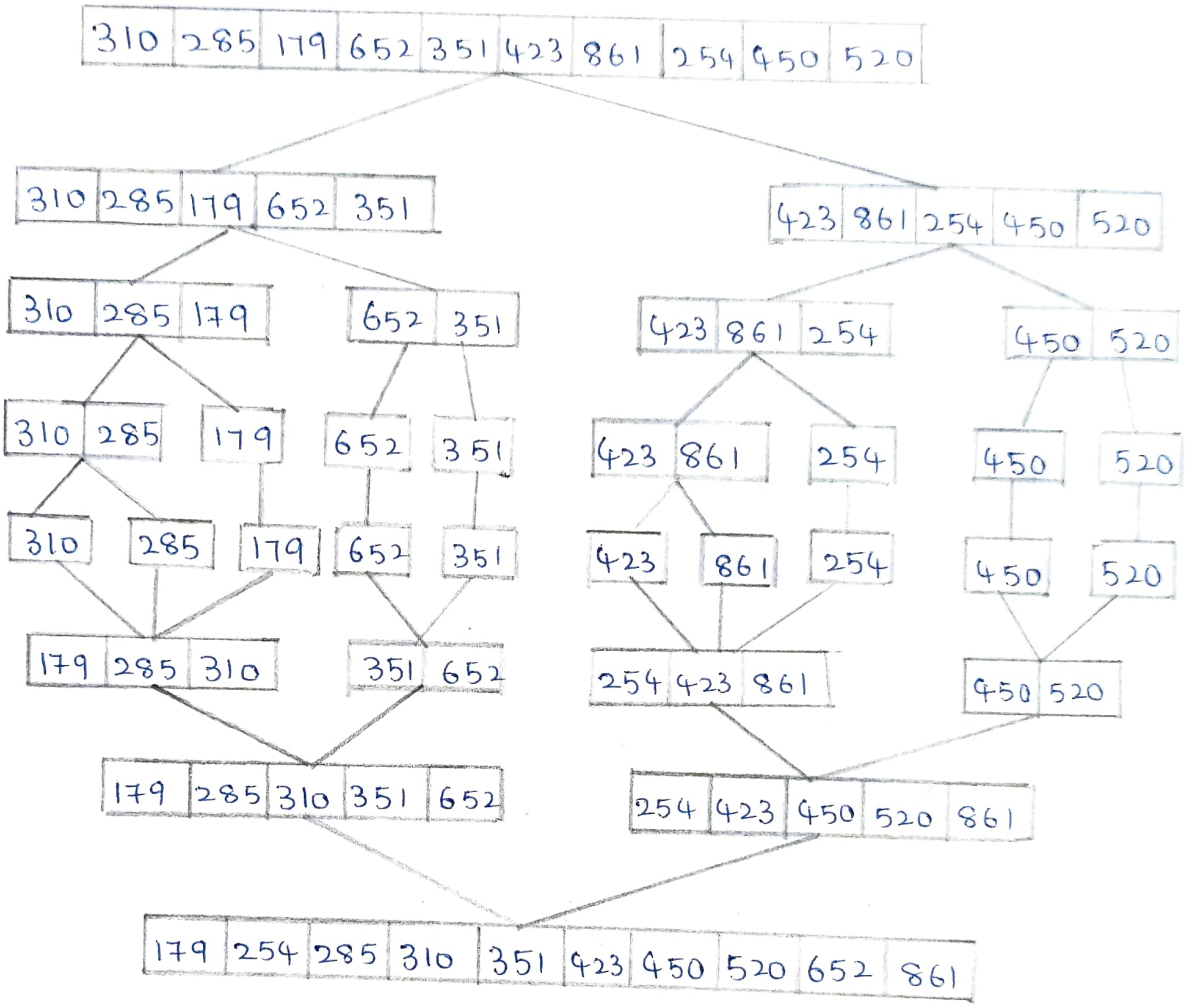


Ex 2) 310 285 179 652 351 423 861 254 450 520

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

$$mid = \frac{low + high}{2}$$

$$= \frac{0+9}{2} = 4$$



Algorithm of merge sort

```
Algorithm Merge Sort (low, high)
// a[low:high] is a global array to be sorted
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted
{
if (low > high) then // if there are more than
                        one element
{
// Divide P into Subproblems
// Find where to split the set
   mid := [(low + high)/2];
// Solve the Subproblems
  Merge sort (low, mid);
  Merge Sort (mid+1, high);
// Combine the Solutions
  Merge (low, mid, high);
  }
}
```

Analysis: The recurrence relation of merge Sort is

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

$T(n/2)$ time taken by the left Sublist to get Sorted.

$T(n/2)$ time taken by the right Sublist to get Sorted

$C(n)$ time taken to combine the two Sublists

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

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

$$a = b^d$$

$$2 = 2^1$$

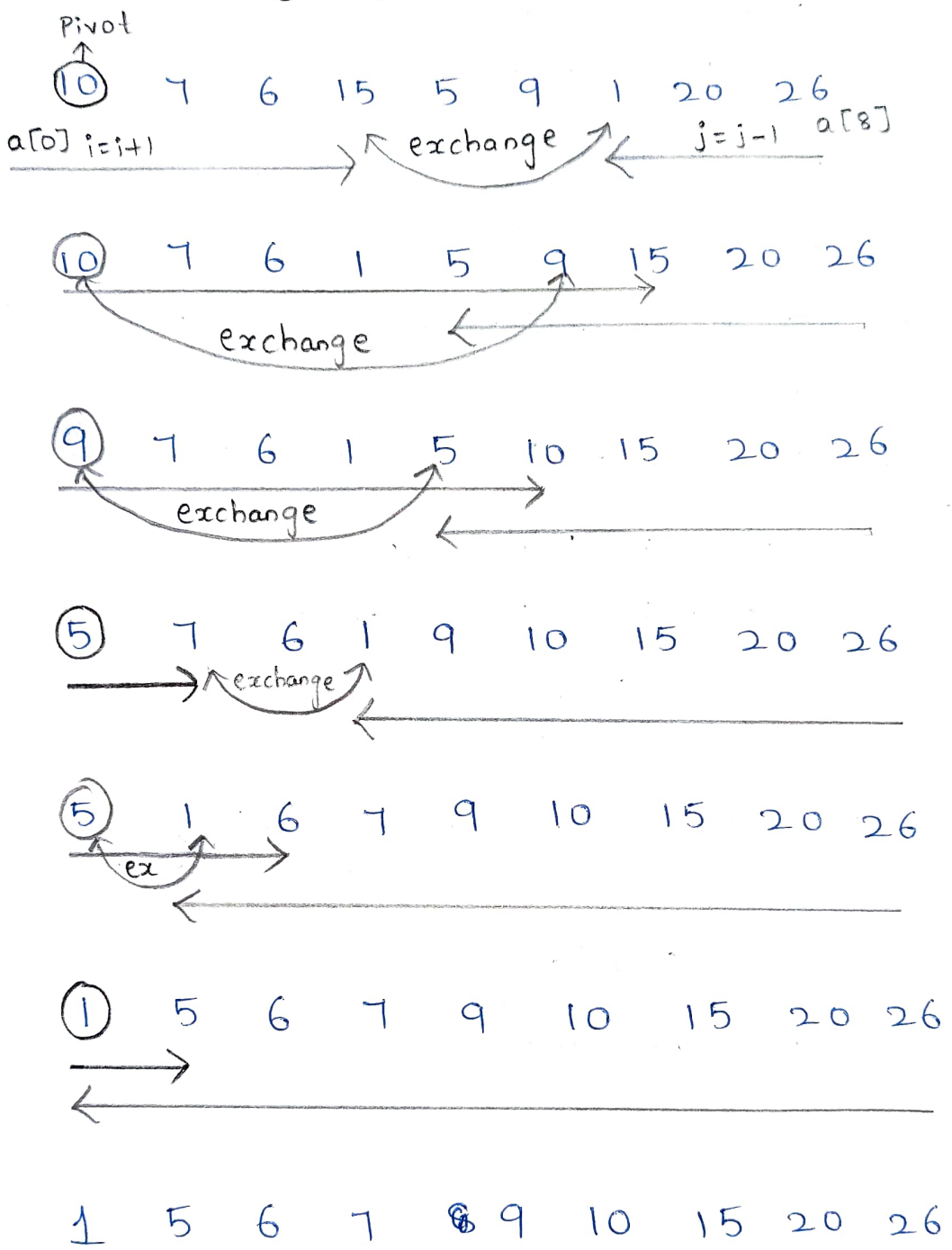
The time complexity of merge Sort is

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

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

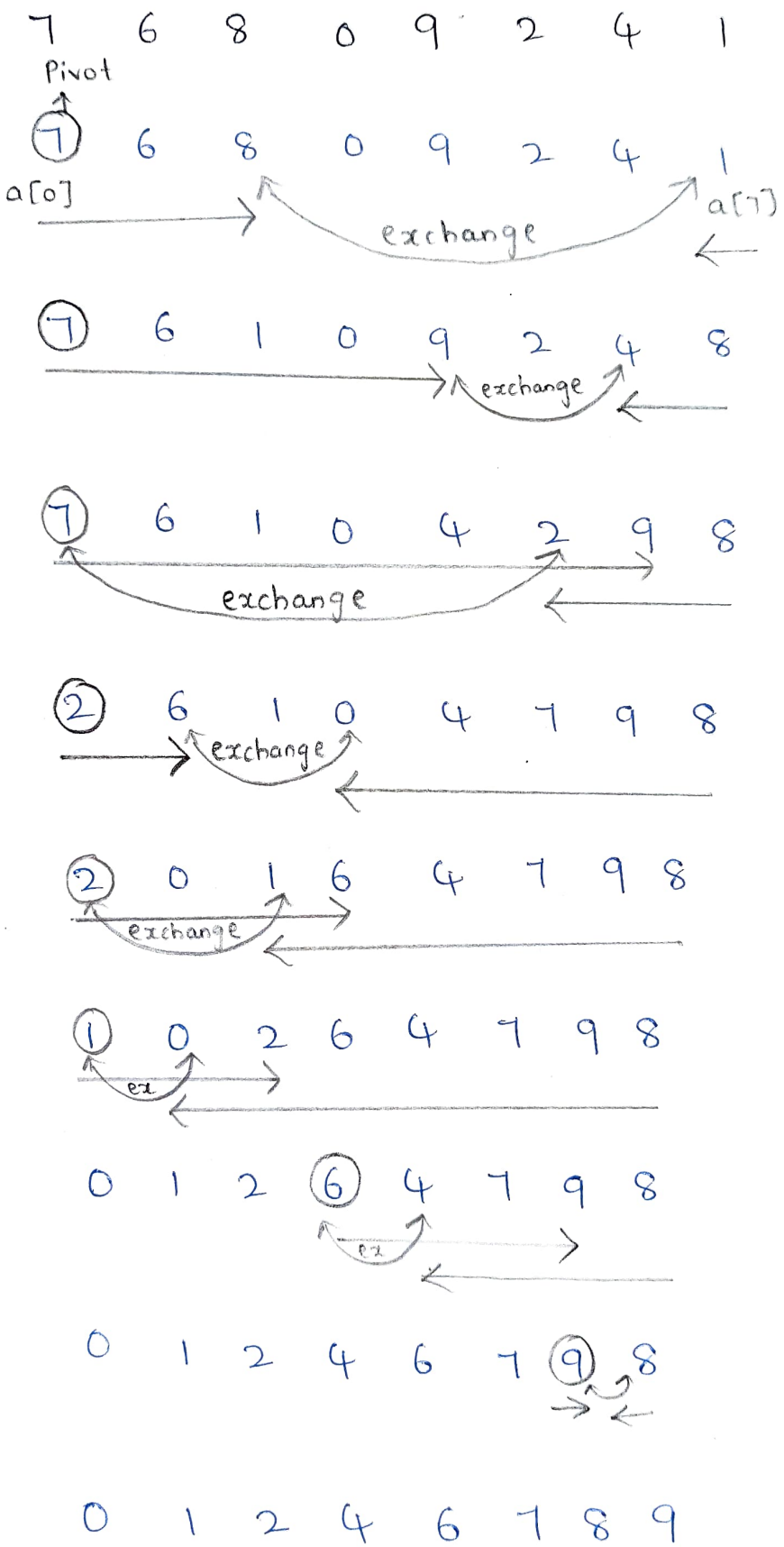
iii) Quick Sort (or) Partition Exchange Sort

Ex: 10 7 6 15 5 9 1 20 26



1 5 6 7 9 10 15 20 26

Ex:2 Arrange the following elements using in Sorted list using Quick Sort



Algorithm for Quick Sort

```
Algorithm partition (a,m,p)
// within a[m], a[m+1]... a[p-1] the elements are
// rearranged in such a manner that if
   initially t=a[m]
// then after completion a[q]=t for some q
        between m.
// and p-1, a[x] ≤ t for m ≤ k ≤ q, and a[k] ≥ t
// for q<k<p. q is returned Set a[p] = ∞
 {
  v:= 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)
// exchange a[i] with a[j]
{
P:= a[i];
a[i]:= a[j]; a[j]:=P;
}
```

→ Time complexity of Quick sort is same as Merge sort

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

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

$$aT(n/2) + F(n)$$

$a=2 \quad b=2 \quad n=1$

$$a = b^d$$

$$2 = 2^1$$

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

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

Strassen's matrix multiplication

Let A and B be two nxn matrices. The product of AB is also nxn matrix whose i,j element is formed by taking the elements in the i^{th} row of A and j^{th} column of B and multiplying we get

$c(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(j,k)$ $\forall$ i & j between 1 and n

→ To compute c(i,j) we need n multiplications. As the matrix c has n^2 elements

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

$$a = 8,\ b = 2,\ d = 2$$

$$a = b^d$$

$$8 = 2^2$$

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

$$= \theta(n \log 2^8)$$

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

$$= \theta(n^3)$$

→ Volker Strassen has discovered a way to compute c(i,j) using only 7 multiplications, 18 addition (or) Subsbractions.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

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

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

→ $C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

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

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

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

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

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

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

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

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

Compare with master's theorem

$a = 7 \quad b = 2 \quad d = 2$

$a \; b^d$

$7 > 2^2$

$= \theta(n \log b^a)$

$= \theta(n \log 2^7)$

$= \theta(n^{2.81})$

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

$P = (A_{11} + A_{22})(B_{11} + B_{22})$, $S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$, $V = (A_{12} - A_{22})(B_{21} + B_{22})$

$= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22}$
$+ (A_{12} - A_{22})(B_{21} + B_{22})$

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

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

$C_{12} = R + T$

$= A_{11}(B_{12} - B_{22}) + (A_{11} + A_{12})B_{22}$

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

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

$C_{21} = Q + S$

$= (A_{21} + A_{22})B_{11} + A_{22}(B_{21} - B_{11})$

$= A_{21}B_{11} + A_{22}B_{11} + A_{22}B_{21} - B_{11}A_{22}$

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

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

$= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{11}(B_{12} - B_{22}) - (A_{21} + A_{22})$
$B_{11} + (A_{21} - A_{11})(B_{11} + B_{12})$

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

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

11.