* Algorithm:-

An algorithm is a finite set of instructions that is followed to accomplishes a particular task.

* Characteristics of an Algorithm:-

An algorithm must have the following characteristics.

i) Input
ii) output
iii) finiteness
iv) Definiteness
v) Effectiveness.

i) Input :- There are some input data which are externally supplied to algorithm to perform a task.

e.g:- i) we want to add 2 numbers
ii) To print a name → no input is required.

ii) output :- There is will be atleast one o/p as a result. The number of outputs to algorithm may be 1 or many.

iii) Definiteness :- Each instructions/ steps of the algorithm must be clear and unambiguous.

iv) Finiteness :- The algorithm should terminate after a finite number of steps

1. Start.
2. for (i=1; i<5; i++)
   display "GECJ"
3. Stop.

} This algo. is going to stop after 4 number of iterative steps. (involving repetations)

v) Effectiveness :- The statements/ instructions of an algorithm should be simple that, the output can be found out by pen and paper.

* Complexity:- There are 2 complexities of an algorithm.

1) Space complexity.
2) Time complexity.

1) Time complexity:-

Is the amount of time the computer needs to run the completion of an algorithm.

Ex:-

Statements	Total steps
i. Algorithm Sum(a, n)	0
ii. {	0
iii. S=0	1
iv. for i=1 to n do	n+1
v. S=S+a[i]	n
vi. return S	1
vii. }	0
	2n+3

So the complexity is 2n+3

Ex:-

Statements	Total steps
i. Algorithm add (a, b, c, m, n)	0
ii. {	0
iii. for i=1 to m do	m+1
iv. for j=1 to n do	mn+m
v. C[i,j] = a[i,j] + b[i,j]	mn
vi. }	0
	2mn+2m+1

So the complexity is 2mn+2m+1.

Space complexity :-

The space complexity of an algorithm is the amount of memory needs to run the completion.

Example :-

```
1. Algorithm abc(a,b,c)
2. {
3. return a+b+b*c+(a+b-c)/a+b+4;
4. }
```

This algorithm consists of 3 variables a, b, c. So fixed part is 3 and variable part is 0.

∴ Space complexity is 3+0 = 3.

i) Fixed Part :- This part is independent of the characteristics (e.g. numbers) of i/p & o/p's. This part typically includes the instruction space, space for variables, space for constants, etc.

ii) variable part :- consists of the space needed by component variables whose size is dependent on the particular problem

∴ S(P) = C + Sp.

where S(P) = Space required / space complexity

C = fixed part.

Sp = variable part.

Ex:-
```
i. Algorithm Sum(a,n)
ii. {
iii. S = 0
iv. for i = 1 to n do
v. S = S + a[i]
vi. return S.
vii. }
```

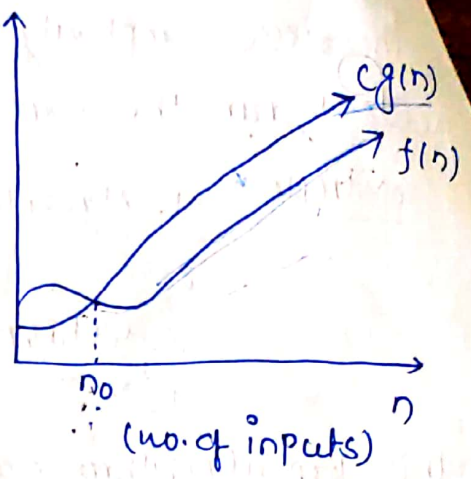
* Asymptotic Notations:-

i) **Big oh (O)** - give upper bound

after some value of n_0 the value of $c.g(n)$ is always greater than $f(n)$

$f(n) \leq c.g(n)$ $\quad n \geq n_0$ $\quad c>0, n_0 \geq 1$

c, n are real no.



$\therefore f(n) = O(g(n))$ ← $f(n)$ is smaller than $g(n)$.

e.g:- 1) $f(n) = 3n+2$ $\quad g(n) = n$ $\quad$ is $f(n)$ is $O(g(n))$?

$3n+2 \leq cn$

$3n+2 \leq 4n$ $\quad c=4$, $n \geq 2$

$\therefore$ if $f(n) = 3n+2$ $\quad g(n) = n$ $\quad f(n) = O(g(n))$.

$g(n) = \begin{cases} n^2 \\ n^3 \\ n^4 \end{cases}$ These also fine but always go for tightest bound.

ii) **Big omega (Ω)** - give lower bound

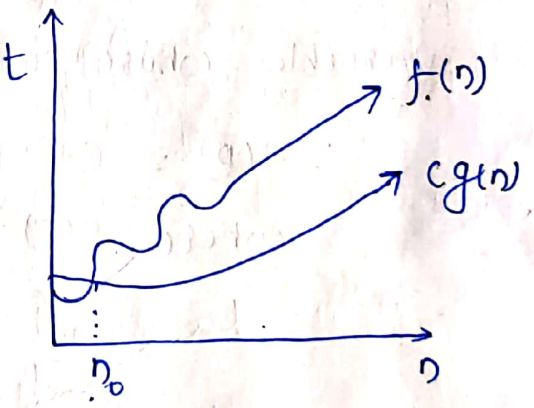
$f(n) \geq c \cdot g(n)$, $\quad n \geq n_0$, $\quad c>0, n_0 \geq 1$

$f(n) = 3n+2$ $\quad g(n) = n$

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

$f(n) \geq cg(n)$

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



$f(n) = 3n+2$ $\quad g(n) = n^2$ $\quad$ is $f(n)$ is lower bounded by $g(n)$ i.e. $3n+2$ is... gr^2

$3n+2 \stackrel{?}{=} \Omega(n^2)$, $\quad 3n+2 \geq cn^2$ this cond^n should satisfy at no.

iii) **Big Theta (Θ)**

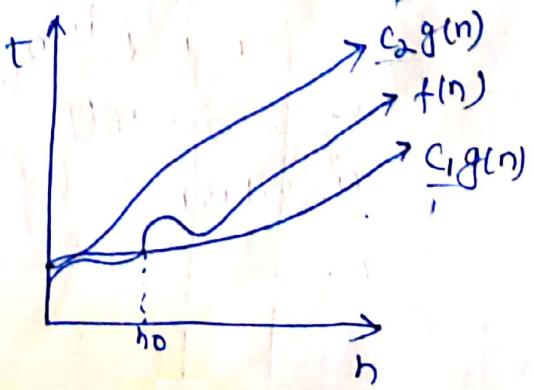
$f(n) = \Theta(g(n))$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\quad c_1, c_2 > 0$

$f(n) = 3n+2$ $\quad g(n) = n$

$f(n) \leq c_2 g(n)$ $\quad f(n) \geq c_1 g(n)$

$3n+2 \leq 4(n)$, $n_0 \geq 1$ $\quad 3n+2 \geq n$ $\quad n_0 \geq 1$



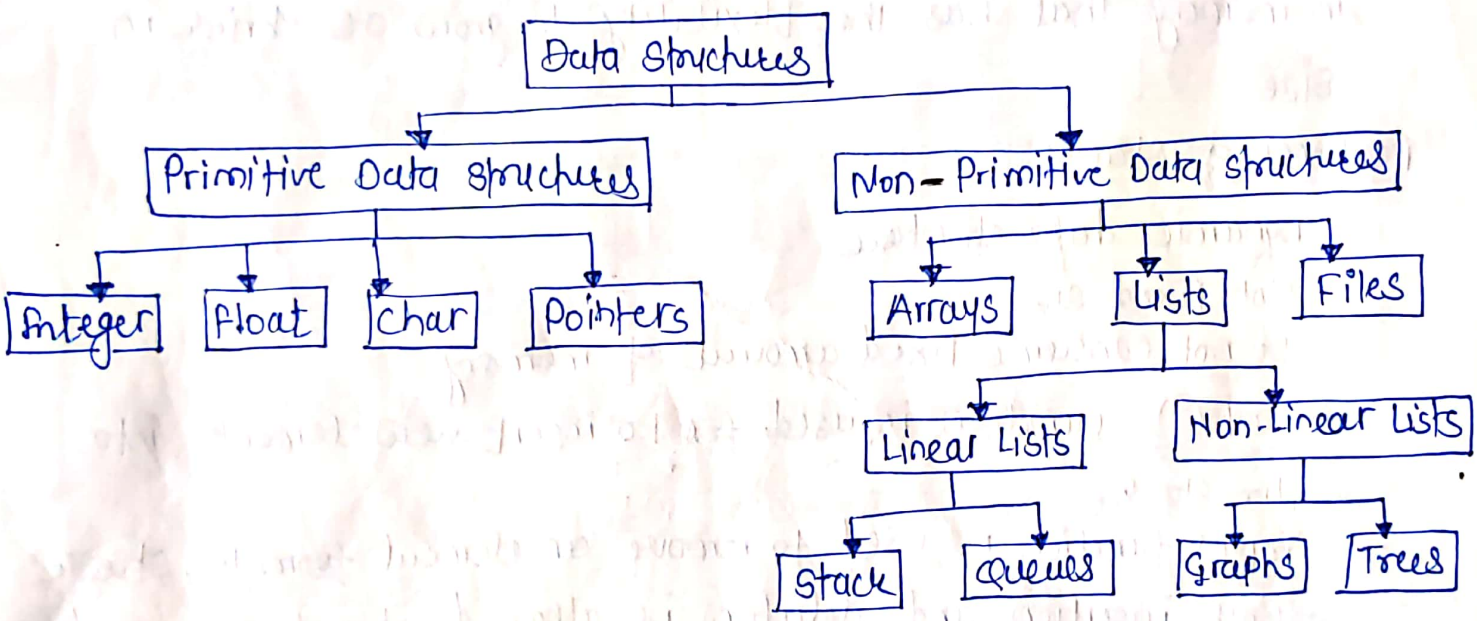
O - Worst case } (we are always intrested)
Ω - Best case → when both are same then we will go for Θ.
* Θ - Average case

5	8	10	13	15	6	1

$x = 5$ $\Omega(1)$ $O(n)$ ← worst case

$\Theta(n/2) = \Theta(n)$. ← Average case

* Data Structures :-



Types of Data structures.

- Linear Data Structure :- Is used to store similar types of data. An array is a finite collection of similar elements stored in adjacent memory locations.

eg: int a[20], b[3], c[7]; // one dimensional array.

```
int A[3][4] = { 10,13,24,3,
                41,5,6,17
                8,91,40,16 };
```

Two dimensional array.

$A[j][k] = base(A) + W[N(j - Row\_lowerbound) + (K - col\_lowbound)]$

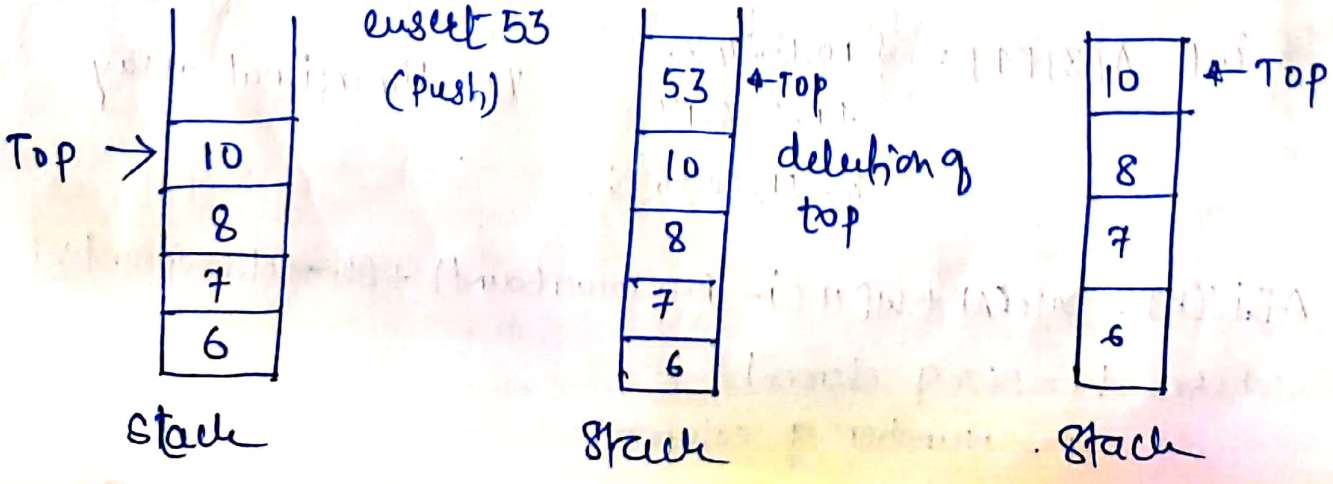
where W = Size of element
N = Number of columns

STACK

* Def^n:- Stack is an abstract data type with a bounded capacity.
  - It is a simple data structure that allows adding & removing elements in a particular order.
  - Stacks are dynamic data structure that follows Last in first out (LIFO) principle:-
    e.g:- stack of trays on a table.
  - A dynamic data structure refers to an collection of data in memory that has the flexibility to grow or shrink in size.

Features of Stacks:-

- Dynamic data structure
- Not fixed size
- Do not consume fixed amount of memory.
- push() function in used ~~for~~ to insert new elements into the stack.
- pop() function is used to remove an element from the stack
- Both insertion and deletion is allowed at only one end of stack called Top.
- Stack is said to be overflow state when it is completely full. & is said to be in underflow state when it is completely empty.



Operations on STACK:-

① PUSH(X) ② POP() ③ topElement() ④ IsEmpty() ⑤ Size()

① **PUSH(X)**:- Insert element x at the top of a stack

Algorithm:- i) Check if the stack is full or not
ii) If the stack is full, then print error of overflow & exit the program.
iii) If the stack is not full, then increment the top and add the element.

Code:-

```
void push ( int stack[], int x, int n)
{ if (top == n-1)
  { // If the top position is the last position of stack ← full
    cout << " Stack is full";
  }
else {
     top = top + 1;   // Incrementing top position
  stack[top] = x;  // Inserting element on incremented
                      Position
  }
}
```

② **POP()**:-

Algorithm:- i) Check if the stack is empty or not
ii) If the stack is empty, then print error of underflow & exit the program.
iii) If the stack is not empty then print the element at the top and decrement the top.

Position of stack	Status of stack
-1	Stack is Empty
0	only one element in stack
N-1	Stack is full
N	overflow state of stack

Code:-
```
void pop (int stack[], int n)
{
  if (isEmpty())
  {
    cout " stack is empty"; ← underflow
  }
  else
  {
    top = top - 1;
  }
}
```

③ topElement :-

Code:-
```
int topElement()
{
  return stack[top];
}
```

④ isEmpty() :-

Code:-
```
bool isEmpty()
{
  if (top == -1) // stack is empty
    return true;
  else
    return false;
}
```

⑤ size() :-

Code:-
```
int size()
{
  return top + 1;
}
```

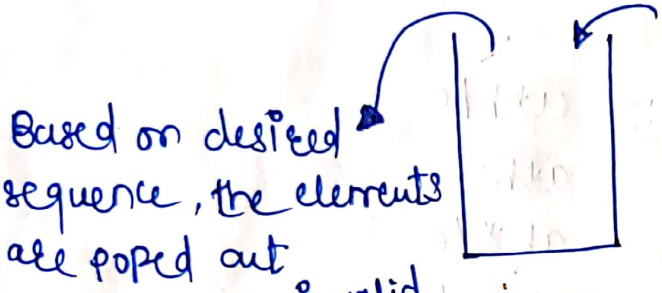
* Applications of stack :-

1) To reverse a word/string.
2) Parsing.
3) Expression conversion (Infix to Postfix, Postfix to Prefix) (Polish notation)
4) Permutations
5) Recursion

Analysis of Stack Operations:-

- Time complexities for various operations that can be performed on the stack data structure.
  - Push operation :- O(1) } we always have to insert or remove the data from the top.
  - Pop operation : O(1) }
  - Top operation :- O(1)
  - Search operation :- O(n)

1) Permutations:-



Q. Identify the invalid stack permutation for the input sequence 1,2,3,4

(a) 3,4,2,1 (b) 4,3,1,2 (c) 2,4,3,1 (d) 1,4,3,2.

Q. Identify valid / invalid stack operations:

a) IsEmpty() , b) GetTop()

i) IsEmpty(Pop(Push(Push(S,X),Y))) = True ← Invalid

ii) GetTop(Push(Pop(Push(Pop(Push(Pop(Push(S,a)),b)),c)) = a ← invalid

2) Polish Notations:-

- Devised by polish
- It avoids the repeated scanning of infix expression.
- compiler prefer → Prefix, Postfix } in one scan we get result

Infix → a+b
Prefix → +ab
Postfix → ab+

} names w.r.t. operator

a+b ≠ +ba This is called parsing error ; +ab = valid

Rule ① :- The relative position of operands : not disturbed.

Rule ② :- The relative position of operands : Distributed according to the precedence & associatively rules.

Ex :- 1) a+b*c (Precedence example)

Prefix :- a+b*c
a+*bc
+a*bc

Postfix :- a+b*c
a+bc*
abc*+

2) a*b/c (left associative)

Prefix :- a*b/c
(a*b)
*ab/c
/*abc

Postfix :- a*b/c
(a*b)
(ab*)/c
ab*c/

3) a↑b↑c. (Right Associative)

Prefix :- a↑b↑c
(b↑c)
a↑(↑bc)
↑a↑bc

Postfix :- a↑b↑c
(b↑c)
bc↑
a↑(bc↑)
abc↑↑

4) i) -b . ii) logx iii) x! iv) log x!

Sol :-

infix	-b	x!	logx
Prefix	-b	!x	logx
postfix	b-	x!	x.logx

$\log x_b$

Prefix:- $\log x_b$
$\log_b x$
$\log_b x$
$\log_b x$

Postfix :- $\log x_b$
$\log x_b$
$x_b \log$

- **Precedence Table:-**

High level
Low level

- unary
↑ : Right
*/ : Left associative
+- : Left associative
=

5) $a = -b + c * d / e - f \uparrow g \uparrow h + i * j$

Prefix:- $-b$ $(c*d)/e$ $f \uparrow g \uparrow h$ $i*j$
$-b$ $(*cd)/e$ $f \uparrow \uparrow gh$ $*ij$
$/*cde$ $\uparrow f \uparrow gh$ $*ij$

$(-b) + (/*cde) - (\uparrow f \uparrow gh) + (*ij)$

$= a + - + - b / * cde \uparrow f \uparrow gh * ij$

Postfix:- $-b$ $c*d/e$ $f \uparrow g \uparrow h$ $i*j$
$b-$ $cd*/e$ $f \uparrow gh \uparrow$ $ij*$
$cd*e/$ $fgh \uparrow \uparrow$

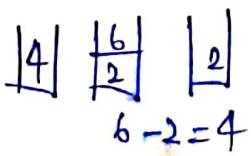
Ans:- $ab - cd*e/ + fgh \uparrow \uparrow - ij * + =$

$6-2=4$ (Evaluation)

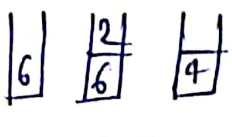
$(6-2)$
Prefix
$-6,2$

operand = push
operator =
i) Pop
ii) evaluation
iii) Put the result

$(6-2)$
$6,2,-$



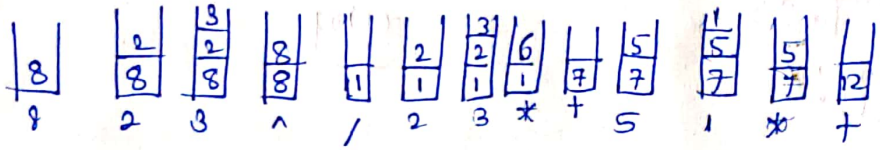
$6-2=4$



$6-2$

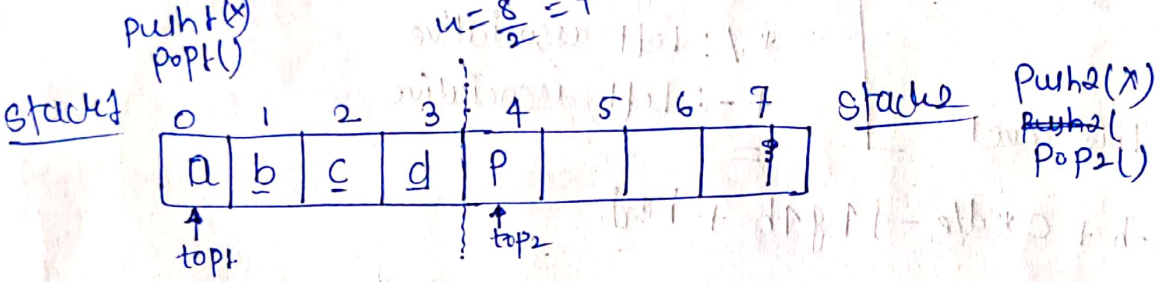
Q. 8, 2, 3, ^, /, 2, 3, *, +, 5, /, *, + (Already in postfix)

Postfix:-



* Implement two ~~Array~~ Stacks in an Array:- (Multiple Stack implementation)

- Approach 1:-



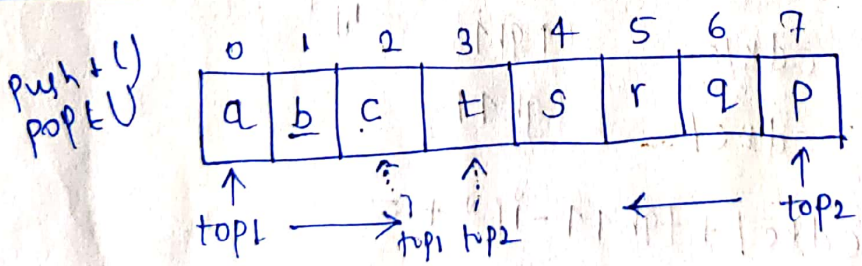
Size ÷ stack1 ⇒ 0 to n/2 - 1

stack2 ⇒ n/2 to (n-1)

- Push(e) - it cannot push because stack1 is full. But there is a space is available in array but it says the stack is overflow.

Disadvantage:- stack overflow even if there is space in array.

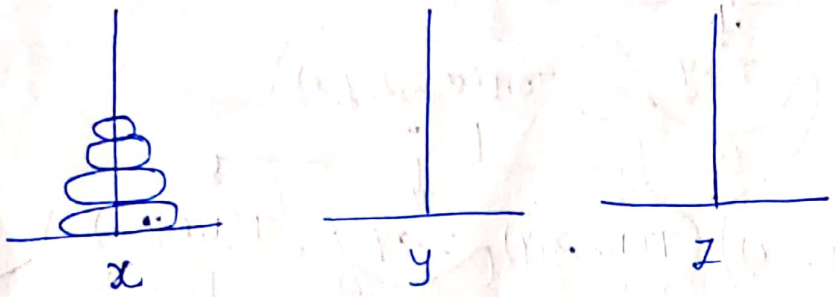
- Approach 2:-



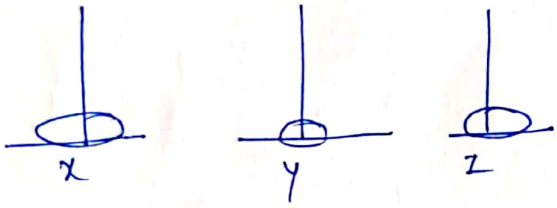
Stack overflow only when

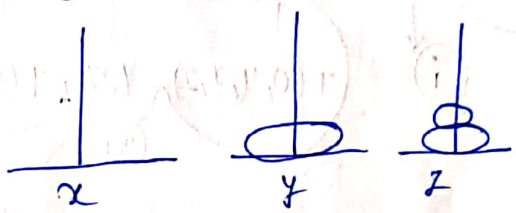
top1+1 == top2

Towers of Hanoi:-

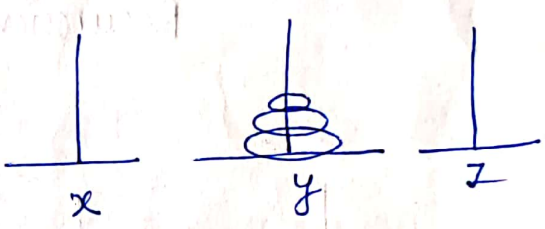


- This is a classical example of recursion.
- Move the disks from x to y by using z tower.

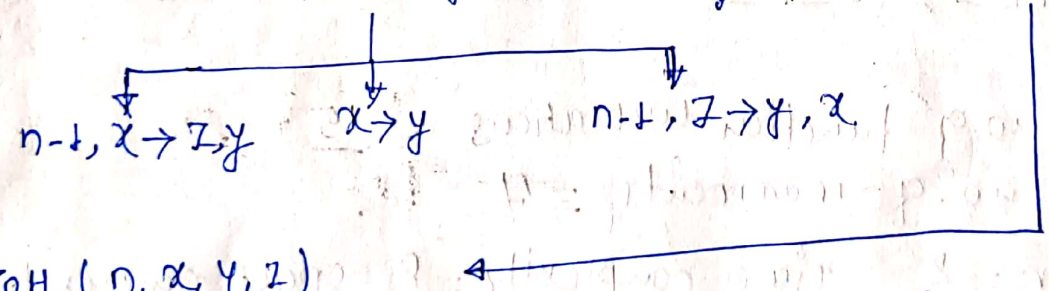
i)


ii)


iii)

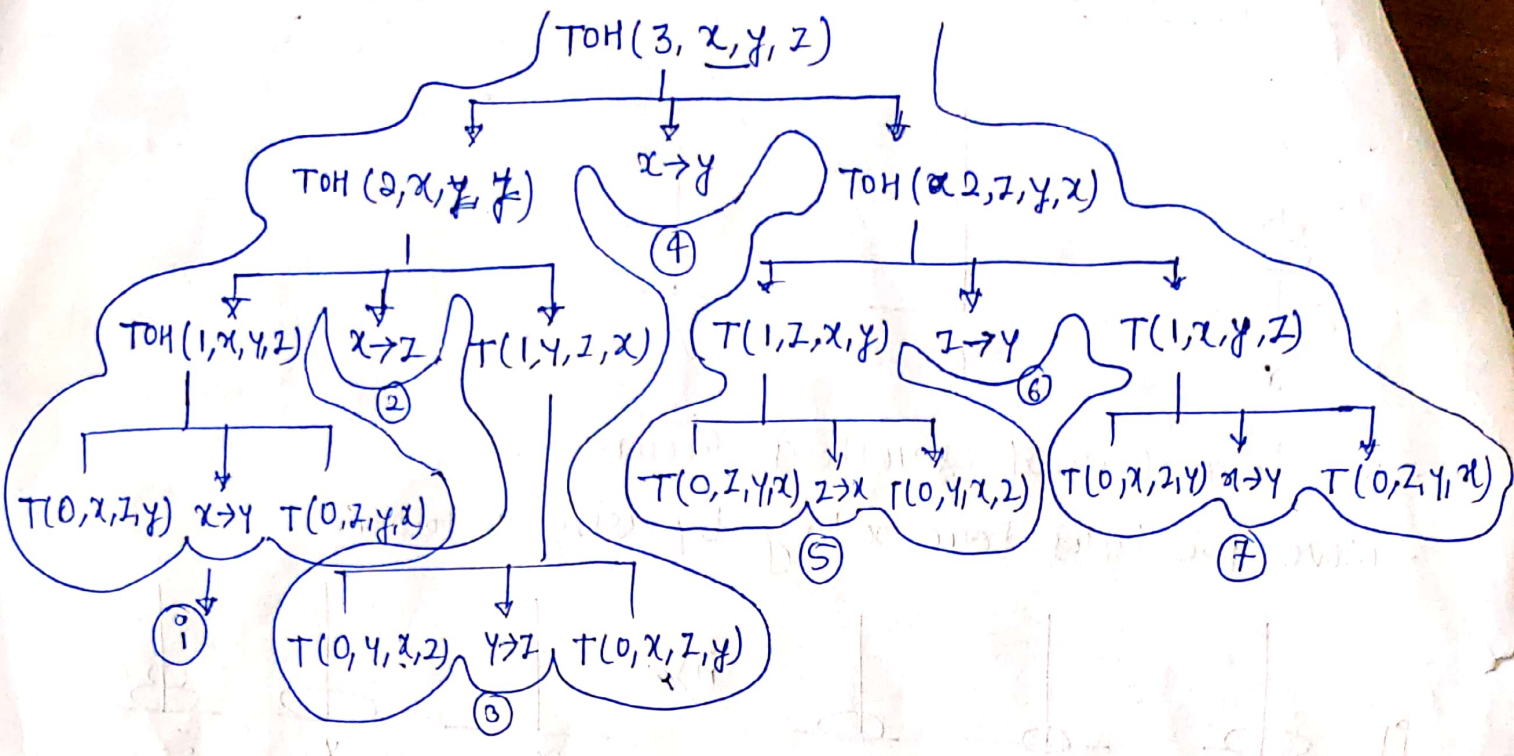

iv)


n, x → y, z ⇒ in order to n disks x to y using z i can do it in 3 steps.

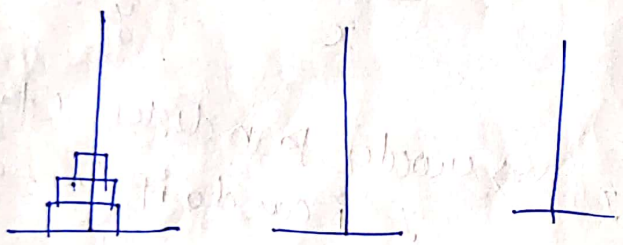


Algo

```
TOH (n, x, y, z)
{
  if (n >= 1)   - There is atleast one disk
  {
    TOH (n-1, x → z, y).
    move top disk from 'x' to 'y';
    TOH (n-1, z, y, x);
  }
}
```



Recursion tree



⇒ ⑦ moves for 3 disks

∴ Total no. of function invocations = 15

Total no. of movements = 7

for n = 3, time complexity = 3, space complexity = 3

∴ number of invocations required for n disks

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

$F(n) = 1, \; n = 0$ ← Base case

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

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

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

$$F(n) = 2^2 F(n-2) + 2 + 1$$

$$F(n) = 2^2(2F(n-3)+1) + 2+1$$

$$= 2^3 F(n-3) + 2^2 + 2 + 1$$

$$= 2^i F(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 1$$

$n - i = 0 \Rightarrow \underline{i = n}$

$$= 2^n F(0) + 2^{n-1} + 2^{n-2} + \cdots + 1$$

$$= 2^n + 2^{n-1} + 2^{n-2} + \cdots + 1 \quad \text{(Geometric series)}$$

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

$$\boxed{F(n) = 2^{n+1} - 1}$$

$\frac{ar^n - 1}{r-1}$

* Movements:-

$M(n) = 2M(n-1) + 1$ (Base conditions are different)

$M(n) = 0, 0$ ← no. of movements = 0, when $n = 0$

$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 1$

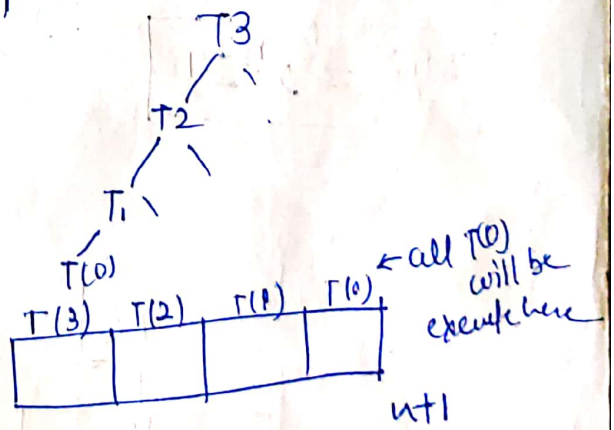
$n - i = 0 \Rightarrow i = n$

$$= 2^n M(0) + 2^{n-1} + 2^{n-2} + \cdots + 1$$

$$= 0 + 2^{n-1} + 2^{n-2} + \cdots + 1$$

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

$$\boxed{M(n) = 2^n - 1}$$



Time complexity : $T(n) = 2T(n-1) + 1$

$T(n) = 1, n = 0$

$T(n) = 2^n \cdot 2^1 - 1$

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

Space complexity

$$\boxed{n+1 = O(n)}$$

- How we can decrease the number of invocations.

```
TOH (n, x, y, z)
{
  if (n>1)
  {
    TOH (n-1, x, z, y);
    move top 'x' to 'y';
    TOH (n-1, z, y, x);
  }
  else if (n==1) mov x→y
}
```

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

$F(n) = 1, n=1$ ← Base case

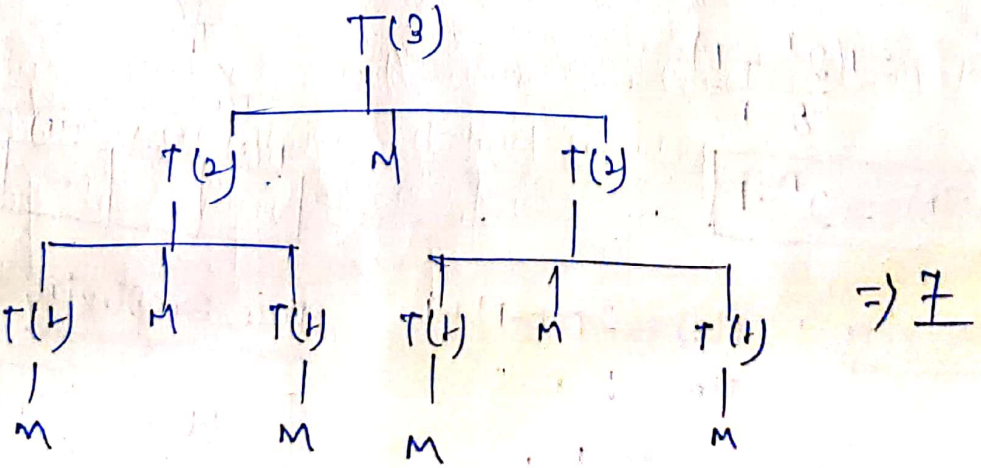
$F(n) = 2^i + F(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 1$

$n-i = 1$

$i = n-1$

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

$$\boxed{F(n) = 2^n - 1}$$



$\Rightarrow 7$