Chapter-4
Recursion

Introduction

Recursion is the powerful technique to write repeatable logic. Recursion is defined by implementing function. A function is said to be recursive function if it calls again and again with reduced input and has a base condition to stop the process. The basic idea behind recursion is to break a problem into smaller version of itself and then build up a solution for entire problem.

A recursive definition consists of two parts

i. Identifying Base / Ground /Anchor case:It is a terminating condition for the problem while designing a recursive function.

ii. Identifying recursive/inductive step: Each time the function call itself, it must be closer to the base case.

Example 1:

Consider F(n) which find the sum of first n natural number. Mathematically the function can be defined as

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

Now recursive definition of this function can be given as

Put n = 1, F(1) = 1
n = 2, F(2) = 1 + 2 = F(1) + 2
n = 3 , F(3) = 1 + 2 + 3 = F(2) +3

n = N, F(N) = F(N-1) +N

$$\text{i.e. } F(N)=\begin{cases} 1 & \text{if } n=1 \quad \text{(Base Case)} \\ F(N-1)+N & \text{if } n>1 \quad \text{(Inductive Step)} \end{cases}$$

So using this recursive function we can generate a sequence of numbers 1, 3, 6, 10, 15,……etc which includes the sum of first 1,2,3,4,5……natural numbers.

Example 2:

Consider F(n) which finds the factorial of number n.

Now recursive definition of this function can be given as

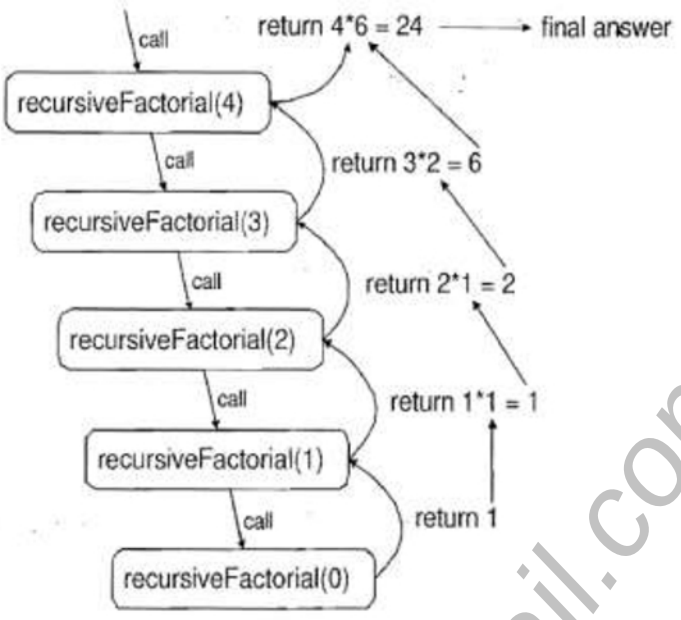
Put n = 0, F(0) = 1
n = 1, F(1) = 1 *1 = 1 * F(0)
n = 2, F(2) = 2 * 1 = 2 * F(1)
n = 3, F(3) = 3 * 2 * 1 = 3 * F(2)

n = N, F(N) = N * F(N-1)

$$\text{i.e. } F(N)=\begin{cases} 1 & \text{if } n=0 \text{ (Base Case)} \\ N * F(N-1) & \text{if } n>0 \text{ (Inductive Step)} \end{cases}$$

So using this recursive function we can generate a sequence of numbers 1, 1, 2, 6, 24…….. etc which includes the factorial of the numbers 0, 1, 2, 3, 4,……..

How it works?



Types of Recursion

1. Direct Recursion and Indirect Recursion
   - A recursion is said to be direct if a function itself.
     Example:

```
public void abc()
{
        abc();
}
```

   - A function is said to be indirect recursion if it contains a call to another function which ultimately call it.
     Example:

```
public void abc()
{
        xyz()
}
```

```
public int xyx()
{
        abc();
}
```

2. Tail Recursion and Non Tail Recursion
   - A recursion is said to be tail recursion if there are no pending operation to be performed on return from the recursive call, otherwise it is called non tail recursion.

Example:

```
public int fact(int n)
{
        if(n==0) return 1;
        return (n * fact ( n -1));
}
```

Algorithm 1

```
public int fact(int n , int res)
{
        if(n==0) return res;
        return fact(n – 1, n * res);
}
```

Algorithm 2

Algorithm 1 is non-tail recursion because it has pending operation i.e. multiplication to be performed on the return from each recursive call, but algorithm 2 is tail recursive since no such pending operation needs to be performed form each recursive call.

3. Nested Recursion
   - A recursion is said to be nested recursion if a function is not only defined in terms of itself, but also is used as one of the parameters.
     Example:

$$h(n)=\begin{cases}0 & \text{if } n=0\\ n & \text{if } n>4\\ h(2+h(2n)) & \text{if } n\le 4\end{cases}$$

```
int H( int n)
{
        if( n==0) return 0;
        else if (n > 4 ) return n;
        else return(H( 2 +H(2n)));
}
```

So in the above example, the recursive method check consists of parameter which again make recursive call so it is nested recursion. Here if n=2, then it will return 12.(how?)

If n =2, then H(2) = H(2 + H(4)) =H(2 + H(2 + H(8)))= H(2 + H(2 + 8) = H (2 + H(10))= H (2 +10)= H(12)=12

If n = 1 then H(1) = H(2 + H(2)) = H(2 + 12) = H(14) = 14

If n = 3 then H(3) = H(2 + H(6)) = H(3 + 6) = H(9) = 9

4. Excessive Recursion

A recursion is said to be excessive recursion if the price for using recursion is **slowing down execution time** and storing on the run-time stack **more things** than required in a non-recursive approach. If the recursion is too deep then we may run out of stack.

Example: A Recursive function for generating sequence of Fibonacci terms.

$$\text{Fib}(n)=\begin{cases}n & \text{if } n<2\\ \text{Fib}(n-2)+\text{Fib}(n-1) & \text{otherwise}\end{cases}$$

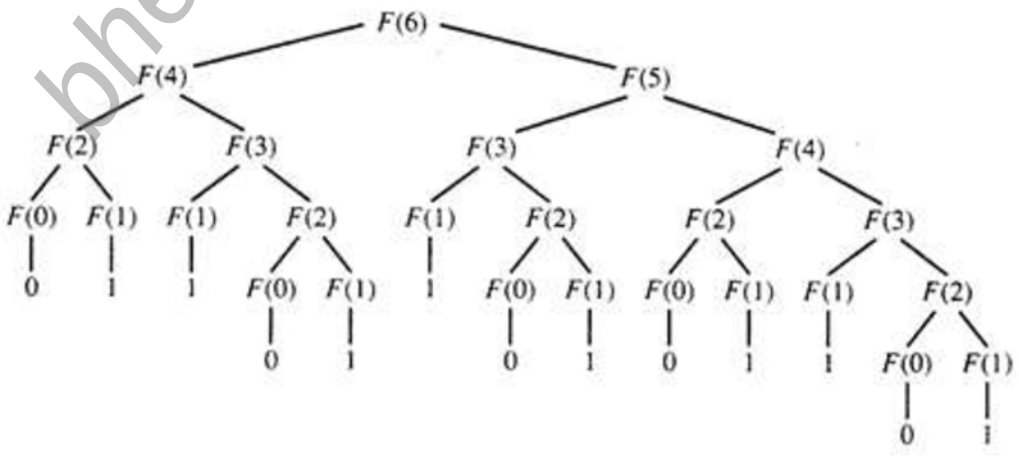
Recursive Definition

```
int Fib (int n) {
  if (n < 2)
    return n;
  else return Fib(n-2) + Fib(n-1);
}
```

C Implementation

So using the above recursive definition we can generate a sequence of numbers 0, 1, 1, 2, 3 , 5, 8……which includes the 0^{th}, 1^{th}, 2^{nd}, 3^{rd}, 4^{th}, 5^{th}, 6^{th}……Fibonacci terms

Now the tree for **Fib(6)** is given below



i.e Fib(6) = Fib(4) +Fib(5)
= Fib(2) + Fib (3) + Fib(5)
= Fib(0) + Fib(1) + Fib(3) + Fib(5)
= 0 + 1 +Fib(3) +Fib(5)
= 1 + Fib(1) + Fib(2) + Fib(5)
= 1 + 1 + Fib(0) + Fib(1) + Fib(5)
= 2 + 0 + 1 +Fib(5)
= 3 +Fib(3) + Fib(4)
= 3 + Fib(1) + Fib(2) + Fib(4)
= 3 + 1 + Fib(2) + Fib(4)
= 4 + Fib(0) + Fib(1) + Fib(4)
= 4 + 0 + 1 + Fib(4)
= 5 + Fib(4)
= 5 + Fib(2) + Fib(3)
= 5 + Fib(0) + Fib(1) + Fib(3)
= 5 + 0 + 1 +Fib(3)
= 6 + Fib(1) + Fib(2)
= 6 + 1 + Fib(0) + Fib(1)
= 7 + 0 + 1
= 8

So for computing 6^{th} term, we have to make call to Fib() for 25 times. For each call it must make use of some memory resources to make room for the stack frame. So if the recursion is deep, then say **Fib(1000) ,** then we may run out of memory. So it is usually best to develop iterative algorithm while working with large number.

Advantage of using Recursion

1. We can create simple and easy version of programs using recursion.
2. Some specific application are meant for recursion such as binary tree traversal, tower of Hanoi etc.

Disadvantages of using Recursion

1. It occupies more memory because of implementation of stack.
2. It consumes more time to get desired result because they uses calls.
3. The computer may run out of memory if proper precautions are not taken.

Application of Recurions

1. It is used to calculate factorial of a given number.
2. It is used to solve TOH problem.
3. It is used to translate infix expression into postfix expression.
4. It is used to check validity of expression.
5. It is used to calculate term in Fibonacci series.

Comparisons of Iteration and Recursion

Iteration	Recursion
1. It is the process of executing a group of statements repeatedly until some specified condition is satisfied.	1. It is the technique of defining something in term of itself.
2. It uses looping so the steps involved are initialization, condition, execution and updation.	2. The steps involved in recursive procedure are identifying base case and identifying recursive step.
3. Iteration executes very fast and consumes less memory and it can be easy.	3. Recursion consumes more time to execute and consumes a lots of memory.
4. There are some application in which iteration is not best suited such as TOH, tree traversal as iterative function are difficult to design and take more programming time.	4. There are some applications in which recursion is best suited for designing algorithms such as TOH, tree traversal as recursive function are more efficient and can be understood easily.
5. Any recursive problem can be solved iteratively.	5. Not all problems have recursive solution

6. Example:
iterative solution for finding factorial of n number
int fact(int n)
{
if(n==0) return 1;
	6. Example:
	Recursive solution for finding factorial of n number
	int fact(int n)
	{
	if(n==0) return 1;

```
        prod=1;
        for(int i=n; i>=1;i--)
        {
                prod=prod * i;
        }
        return prod;
}
```

```
        return(n * fact(n-1));

}
```

Method Calls and use of stack

When a method is called from main program or by other methods, the method called from has to be remembered by the system. This could be done by storing the return address in main memory in a place set aside for return addresses. For a method call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is the solution.

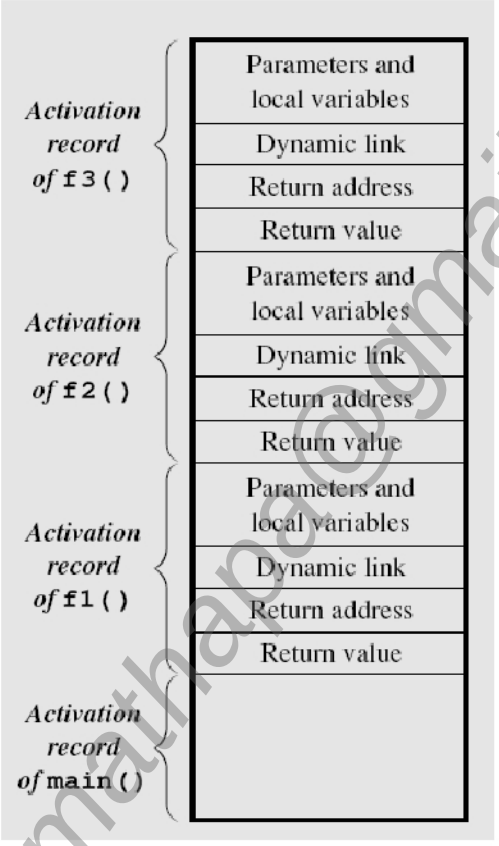


Fig: Content of stack when main call F1() , F1() call F2() and F2() call F3()

So data structure containing the information about the state of method i.e. content of automatic variable, values of methods parameter and the return address indicating where to restart its caller is called **activation record or stack frame.**

If a method is called either by main() or by another method, then its activation record is created on the run-time stack. The run-time stack always reflects the current state of the method. Once the return statement is encountered, control come back to previous call by using the return address present in the stack.

Example: Suppose that main() calls method f1(), f1() calls f2(), and f2() in turn calls f3(). If f3() is being executed, then the state of the run-time stack is as shown in Figure above. By the nature of the stack, if the activation record for f3() is popped by moving the stack pointer right below the return value of f3(), then f2() resumes execution and now has free access to the private pool of information necessary for reactivation of its execution. On the other hand, if f3() happens to call another method f4(), then the run-time stack increases its height because the activation record for f4() is created on the stack and the activity of f3() is suspended.

Tower Of Hanoi

Tower of Hanoi is a classical problem, which consists of N different sized disk and three towers over which these disk can be mounted. The problem of TOH is to move disk from one tower to another with the help of temporary tower. If we have three towers A, B and C. All the n disks are mounted on tower A in such a way that a larger disk is always below a smaller one then we have to move disk from tower A to tower C with the help of temporary pillar B.

The conditions for playing this game are

1. We can move only one disk from one tower to another at a time.

2. A larger disk can't be placed on the smaller one.
3. One and only one extra tower could be used for temporary storage of disks.

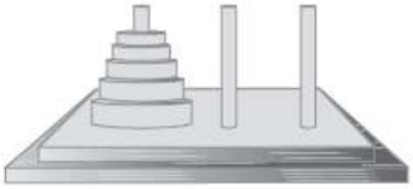
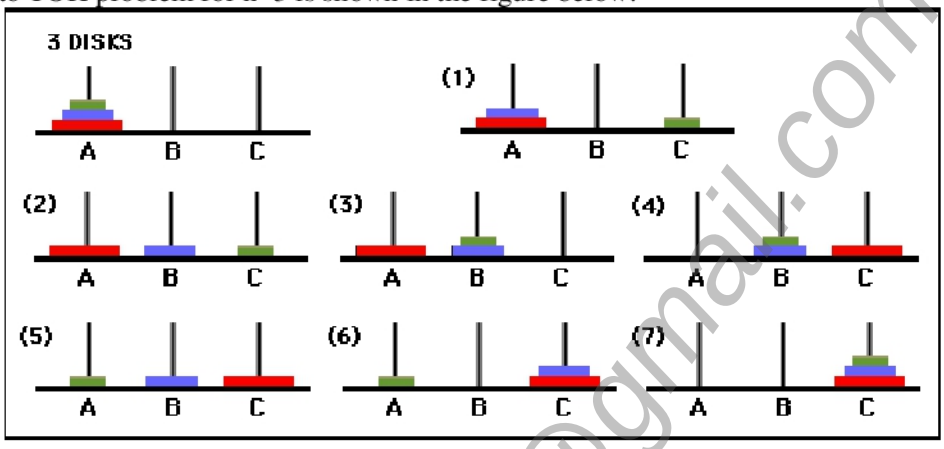


Fig: Initial setup for TOH

In general, the solution to TOH problem requires 2^N -1 moves of disc, where N is the number of disk.
Example: The solution to TOH problem for n=3 is shown in the figure below.



So for n=3, the steps to move disk involves

1. A to C
2. A to B
3. C to B
4. A to C
5. B to A
6. B to C
7. A to C

Algorithm

Let us assume that we have N disk and three towers named source, temp and destination. Now the algorithm for solution to TOH problem is

```
function TOH(N, S, T, D)
{

}
```

1. If (N==1)
   a. Move a disk from S to D
   b. Exit
2. Move upper N-1 Disk from source to temp using destination as temporary.
   TOH(N-1, S, D, T)
3. Move largest disk from source to destination. i.e. Move S To D
4. Move upper N-1 disk from temp to destination using source as temporary.
   TOH(N-1, T, S, D)
5. Exit

Simpler statement of iterative solution

Alternating between the smallest and the next-smallest disks, follow the steps for the appropriate case:

1. For an even number of disks:

- make the legal move between pegs A and B

- make the legal move between pegs A and C
- make the legal move between pegs B and C
- repeat until complete

2. For an odd number of disks:

- make the legal move between pegs A and C
- make the legal move between pegs A and B
- make the legal move between pegs C and B
- repeat until complete

Validity of Expression

A statement generally have the following delimiters: parentheses "(" and ")", square brackets "[" and "]", curly brackets "{" and "}", and comment delimiters "/*" and "*/".A statement is said to be valid for correct matching of delimiters if

i. The number of left delimiters is equal to the number of right delimiters.
ii. Each right delimiter is preceded by matching left delimiter.

Example: {(A + B) – (C * D}} is invalid
{(A + B) – (C * D)} is valid

Algorithm for validation of statement

1. Scan the elements from left to right until the end of statement is encountered.
2. If the scanned element is
   - I. (or { or [:- Push it onto the stack
   - II. / :- Scan the next character, if this character is * skip all characters until */ is found and report an error if the end of statement is reached before */ is reached.
   - III.) or } or]:- Pop the left delimiter from the top of stack
     - a. Check if popped left delimiter match the right delimiter, if it doesn't match then display "Invalid Statement" and Exit.
     - b. If during the attempt of pop operation, stack is found to be empty then display "InvalidStatement, No. of Right bracket > No. of left bracket" and Exit
   - iv. Other character: Ignore them
3. If the scanned element is the end of statement and stack doesn't get empty then display "Invalid Statement, No. of Left bracket > No. of Right bracket " otherwise display "Valid Statement and Exit"

Example: Check the validity of given statements.

1. {(A + B) – (C * D)}]

Scanned Symbol	Stack
{	{
(	{ (
A	{ (
+	{ (
B	{ (
)	{
-	{
(	{ (
C	{ (
*	{ (
D	{ (
)	{
}	**Empty**
]	

Since during attempt of pop operation, stack is found to be empty i.e. No of Right bracket> No of Left bracket, so it is invalid statement.

2. {(A + B) – (C * D})

Scanned Symbol	Stack
{	{
(	{ (
A	{ (
+	{ (
B	{ (
)	{
-	{
(	{ (
C	{ (
*	{ (
D	{ (
}	{ **(Mismatch)**
)	

Since the popped left bracket(( ) doesn't match the scanned right bracket (}), so it is invalid expression

3. [{(A + B) – (C + D)}]

Scanned Symbol	Stack
[	[
{	[{
(	[{ (
A	[{ (
+	[{ (
B	[{ (
)	[{
-	[{
(	[{ (
C	[{ (
+	[{ (
D	[{ (
)	[{
}	[
]	**Empty**

Since we have scanned the last element and stack is empty hence it is valid statement.

4. S =T[5] + U / (V * (W + Y));

Scanned Symbol	Stack
S	**Empty**
=	**Empty**
T	**Empty**
[	[
5	[
]	**Empty**
+	**Empty**

U	Empty				
/	Empty				
(	(				
V	(				
*	(				
(	(	(			
W	(	(			
+	(	(			
Y	(	(			
)	(				
)		Empty			
;		Empty			

Since we have scanned the last element and stack is empty hence it is valid statement.

Some programs Related to recursion

1. Write program to print the Fibonacci series where the user limits the number of terms in series.

```
#include<stdio.h>
int fibo(int);

int fibo(int n)
{
        if(n<2)
        {
                return n;
        }
        else
        {
                return (fibo(n-2)+fibo(n-1));
        }
}

int main()
{
        int n;
        printf("Enter the number of term you want to print the series up to");
        scanf("%d",&n);
        for(int i=0;i<n;i++)
        {
                int term=fibo(i);
                printf("%d\t",term);
        }

}
```

Output:

```
Enter the number of term you want to print the series up to15
0       1       1       2       3       5       8       13      21      34      55      89      144     233     377
```

2. WAP to solve the TOH problem

```
#include <stdio.h>
void towers(int,char,char,char);
```

```
void towers(int n,char frompeg,char topeg,char auxpeg)
{
        if(n==1) //If only 1 disk, make the move and return
        {
         printf("\nMove disk 1 from peg %c to peg %c",frompeg,topeg);
         return;
        }
        towers(n-1,frompeg,auxpeg,topeg); //Move top n-1 disks from A to B, using C as auxiliary
        printf("\nMove disk %d from peg %c to peg %c",n,frompeg,topeg); /* Move remaining disks from A to C
        towers(n-1,auxpeg,topeg,frompeg); //Move n-1 disks from B to C using A as auxiliary
}

void main()
{
        int n;
        printf("Enter the number of disks : ");
        scanf("%d",&n);
        printf("The Tower of Hanoi involves the moves :\n\n");
        towers(n,'A','C','B');
        return 0;
}
```

Output:

```
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```

3. WAP to find the factorial of given number using recursion.

```
#include<stdio.h>
int fact(int);

int fact(int n)
{
    if(n==0)
    {
                return 1;
    }
    else
    {
                return (n * fact(n-1));
    }
}

int main()
{
    int n;
    printf("Enter the number");
    scanf("%d",&n);
    int res=fact(n);
    printf("The sum is %d",res);
}
```

Output:

```
Enter the number5
The sum is 120
```

WAP to find the GCD of two numbers entered by user using recursion

```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
   int n1, n2;
   printf("Enter two positive integers: ");
   scanf("%d %d", &n1, &n2);
   printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
   return 0;
}

int hcf(int n1, int n2)
{
   if (n2 != 0)
      return hcf(n2, n1 % n2);
   else
      return n1;
}
```

. WAP to generate Fibonacci series using recursive function where user enters the limit of series

```
#include<stdio.h>
#include<conio.h>
intfibo(int n);
void  main()
{
        int n;
        int term;
        clrscr();
        printf("Enter the limit of fibonacciseries");
        scanf("%d",&n);
        for(int i=1;i<=n;i++)
        {
        term=fibo(i);
        printf("\t%d",term);
        }
        getch();
}
intfibo(int n)
{
        if(n==1 || n==2)
        {
        return 1;
        }
        else
        {
        return (fibo(n-2)+fibo(n-1));
        }
```

```
}
```

Output:

```
Enter the limit of fibonacciseries   10
1       1       2       3       5       8       13      21      34      55
```

WAP to find the sum on first n natural number using recursion.

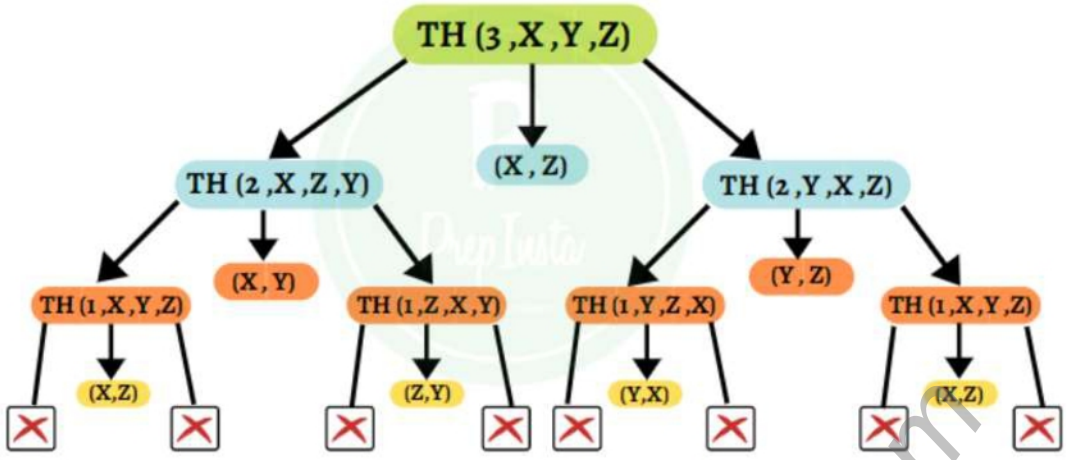
```
#include<stdio.h>
#include<conio.h>
int sum(int);
void main()
{
        int n;
        int res;
        printf("Enter the nth number");
        scanf("%d",&n);
        res=sum(n);
        printf("The sum  is %d",res);
        getch();
}
int sum(int n)
{
        if(n==1)
         {
                return 1;
         }
        else
         {
                return (sum(n-1)+n);
         }

 }
```

Output:

```
Enter the nth number 10
The sum is 55 _
```

Construct Recursion tree for TOH problem for N= 3 Disk. Assume x, y and z as source, auxiliary and destination pillar respectively.



So from the above tree, the necessary steps to solve TOH problem is as below

1. **X – Z.**
2. **X – Y.**
3. **Z – Y.**
4. **X – Z.**
5. **Y – X.**
6. **Y – Z.**
7. **X – Z.**

Construct Recursion tree for TOH problem for N= 4 Disk. Assume x, y and z as source, auxiliary and destination pillar respectively.

[Solve Yourself]