Recursion

Recursion is a process by which a function calls itself repeatedly until some condition has been satisfied. For problems to be solved recursively, two conditions must be satisfied.

a. The problem must be expressed in recursive form.
b. The problem statement must include a terminating or stopping condition.

⇒ A function that calls itself is called recursive function.

Note: At least one statement inside a function must be of type non recursive type

⇒ Each time a function is called, a number of words (such as variables, return address and other arguments and its data) are pushed onto the program stack.

⇒ When the function returns, this frame of data is popped off the stack.

⇒ Recursion of course is an elegant programming technique, but not the best way to solve a problem, even if it is recursive in nature. This is due to the following reasons:

- It requires stack implementation.
- It makes inefficient utilization of memory, as every time a new recursive call is made a new set of local variables is allocated to function.
- Moreover, it also slows down execution speed, as function calls require jumps, and saving the current state of program onto stack before jump.
- If proper precautions are not taken, recursion may result in non-terminating iterations.

⇒ Recursion is one of the most powerful programming tools.

⇒ Recursion provides a natural way to solve many problems.

⇒ Recursion makes algorithms and its implementation more compact and simpler

Recursion versus Iteration:

	Iteration	***Recursion***
1	It is a process of executing a statement or a set of statements repeatedly, until some specified condition is specified.	Recursion is the technique of defining anything in terms of itself.
2	Iteration involves four clear-cut Steps like initialization, condition, execution, and updating.	There must be an exclusive if statement inside the recursive function, specifying stopping condition.
3	Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
4	Iterative counterpart of a problem is more efficient in terms of memory utilization and execution speed.	Recursion is generally a worse option to go for simple problems, or problems not recursive in nature.

A recursive definition consists of two parts.

- **Anchor or Ground case**: The basic elements that are building blocks of the other elements of the set are listed in anchor case.
- **Recursive or Inductive case**: In the recursive case, rules are given that allow for the construction of new object.

The factorial of a number n can be defined recursively as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (anchor)} \\ n*(n-1)! & \text{if } n>0 \text{ (inductive step)} \end{cases}$$

Using this definition, we can compute the factorial of a given number, generate the sequence of numbers of Fibonacci series, calculate product of natural numbers, etc.

Types of Recursions:

1. Direct Recursion
2. Indirect Recursion
3. Tail Recursion
4. Non-tail Recursion

1. Direct Recursion

⇒ If the function call is within its body, then the recursion is direct.

Example:

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

2. Indirect Recursion:

⇒ If the function calls another function which in turn calls itself, then such recursion is indirect.

⇒ A recursive function need not call itself directly. Rather, it may call itself indirectly, as in the following example.

```
int fun1(int x)
{
        if(x <= 0)
                return 1;
        else
                return fun2(x);
}
int fun2(int y)
{
        return fun1(y-1);
}
```

⇒ In this example, method fun1 calls fun2, which may in turn call fun1, which may again call fun2. Thus, both fun1 and fun2 are recursive, since they indirectly call themselves.

3. Tail recursion:

⇒ Tail recursion has recursive call as the last statement in the method.

⇒ In other words, when the call is made, there are no statements left to be executed by the method.

⇒ Recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect.

⇒ For example, the following method ***tail()*** is a tail recursion:

```
void tail(int n)
{
        if(n < 0)
                return;
        else
                System.out.println(n+ " ");
                tail(n-1);
        }
}
```

4. non-tail recursion:

⇒ Recursive methods that are not tail recursive are called non-tail recursive.

Example:

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

Is the following program tail recursive?

```
void prog(int i)
{
        if (i>0)
        {
                prog(i-1);
                System.out.println (i+" ");
                prog(i-1);
        }
}
```

- No, because there is an earlier recursive call other than the last one.

Advantages of recursive function

i. Although at most of the times a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system cannot be solved without recursion.
ii. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
iii. Using recursion, the length of the program can be reduced.

Disadvantages

i. It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive calls separate memory is allocated to automatic variables with the same name.
ii. If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press Ctrl+ break to pause and stop the function.
iii. The recursion function is not efficient in execution speed and time.
iv. It is very difficult to trace and debug recursive function.

Fibonacci Sequence

The Fibonacci sequence is the sequence of integers :0,1,1,2,3,5,8,13,21,34…….

- Each element in this sequence is the sum of the two preceding elements.
- If we let fib(0)=0, fib(1)=1, and so on, then we may define the Fibonacci sequence by the following recursive definition:

fib(n)=n	if n==0 or n==1.
fib(n)=fib(n-1)+fib(n-2)	if n>=2.

A method in Java to compute the sequence of Fibonacci number up to nth term is as shown below:

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

Multiplication of Natural Numbers:

The multiplication of two numbers can also be performed recursively. The product a*b where a and b are positive integers, may be defined as a added itself b times. This is iterative definition. An equivalent recursive definition is:

a*b = a	if b==1
a*b = a + a*(b-1)	if b>1

The recursive method in Java for multiplication of two numbers is as shown below:

```
public static int product(int a, int b)
{
        if(b == 1)
        {
                return a;
        }
        else
        {
                return (a + product (a, b-1));
        }
}
```

Method Calls and Recursion Implementation

When method is called, the must know where to resume execution of the program after the method has finished. The information indicating where it has been called from has to be remembered by the system. For a method call, more information has to be stored than just a return a return address. Therefore, a dynamic memory allocation using the run time stack is a much better solution. The run time stack is maintained by a particular operating system.

What information should be preserved when a method is called?

First, local variables must be stored. If a method **f1()** declares variable **x**, calls method **f2()**, which locally declares the variable **x**. The system has to make a distinction between these two variables. This is more important when function call is recursive where **f1 ()** and **f2 ()** are same.

The state of the each method, including main(), is characterized by the contents of all local variables, the values of the method's parameters and the by the return address indicating where to restart its caller. The data area containing all this information is called activation record and is allocated on run time stack. It has short lifespan. The activation record usually contains the following information:

- Values for all the parameters to the method.
- The return address to resume control by the caller.
- The dynamic link, which is a pointer to the caller's activation record.
- The returned value for a method not declared as void.

Creating an activation record whenever a method is called allows the system to handle recursion properly. Recursion is calling a method that happens to have the same name as the caller. These invocations are represented by different activation records and are thus differentiated by the system.

Activation Record of f3()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of f2()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of f1()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of main ()	

Figure: *Contents of the run time stack when main () calls method f1 (), f1() calls f2(), and f2() calls f3()*.

Anatomy of a Recursive Call

The function that defines raising any number x to a nonnegative integer power n is a good example of a recursive function. The most natural definition of this function is given by:

$x^n = 1$, if n=0

$= x*x^{n-1}$, if n>0

A java method for computing x^n can be written directly from the definition of a power.

```
double power (double x, int n)
{
        if(n==0)
            return 1.0;
        else
            return x*power(x, n-1);
}
```

Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

h(n) = 0 if n = 0

= n if n>4

= h(2+h(2n) if n≤4

Function h has a solution for all n≥0. This fact is obvious for all n>4 and n=0, but it has to be proven for n=1, 2, 3 and 4. Thus, 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

Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on run time stack more things than required in a non-recursive approach. If recursion is too deep, then we can run out of space on the stack and our program terminates abnormally by raising an unrecoverable **StackOverflowError**. But usually, the number of recursive call is much smaller, so danger of overflowing the stack may not be imminent.
Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

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

The method is simple and easy to understand but extremely inefficient. Let us compute Fib(4).

```
                                        Fib(4)
=       Fib(2)                  +       Fib(3)
=       Fib(0)+Fib(1)                   +   Fib(3)
=       0   +   1                       +   Fib(1)  +   Fib(2)
=       0   +   1                       +     1     +   Fib(0) +  Fib(1)
=       0   +    1                      +     1     +    0     + 1

= 3
```

It takes almost a quarter of a million calls to find the twenty -sixth Fibonacci number, and nearly 3 million calls to determine the thirty first. This is too heavy a price for the simplicity of the recursive algorithm.
We can prove that the number of additions required to find Fib(n) using a recursive definition is equal to Fib(n+1)-1. Counting two calls per one addition plus the very first call means that Fib() is called 2.Fib(n+1)-1 times to compute Fib(n). This number can exceedingly large for fairly small ns, as in the following table.

N	Fin(N+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1973
20	10946	10945	21891
25	121393	121392	242785
30	1346269	1346268	2692537

Figure: Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

Tower of Hanoi Problem:

We can use recursive technique to produce a logical and elegant solution to Tower of Hanoi problem.

The initial setup of the problem is:

- Three pegs (or towers) Source (say X), Intermediate (say Y) and Destination (say Z) exists.
- There will be different sized disks. Each disk has a hole in the center so that it can be stacked on any of the pegs.
- At the beginning, the disks are stacked on the X peg, that is the largest sized disk on the bottom and the smallest sized disk on top.

Objective:

- We have to transfer all the disks from source peg X to the destination peg Z by using an intermediate peg Y.

Conditions:

- Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
- Only one disk may be moved at a time.
- Each disk must be stacked on any one of the pegs.

Example: Three disks with different sizes are transfer from tower A to tower C:

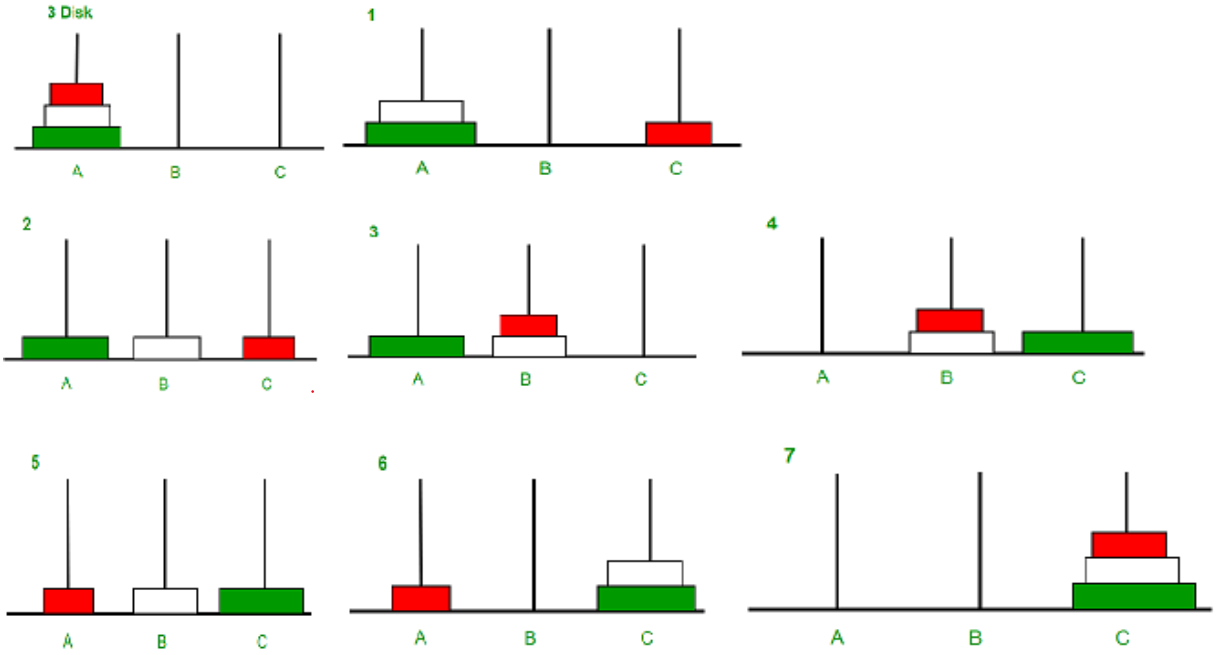
Number of steps required in TOH problem = $2^n - 1$, where n is number of disks

$= 2^3 - 1 = 8 - 1 = 7$

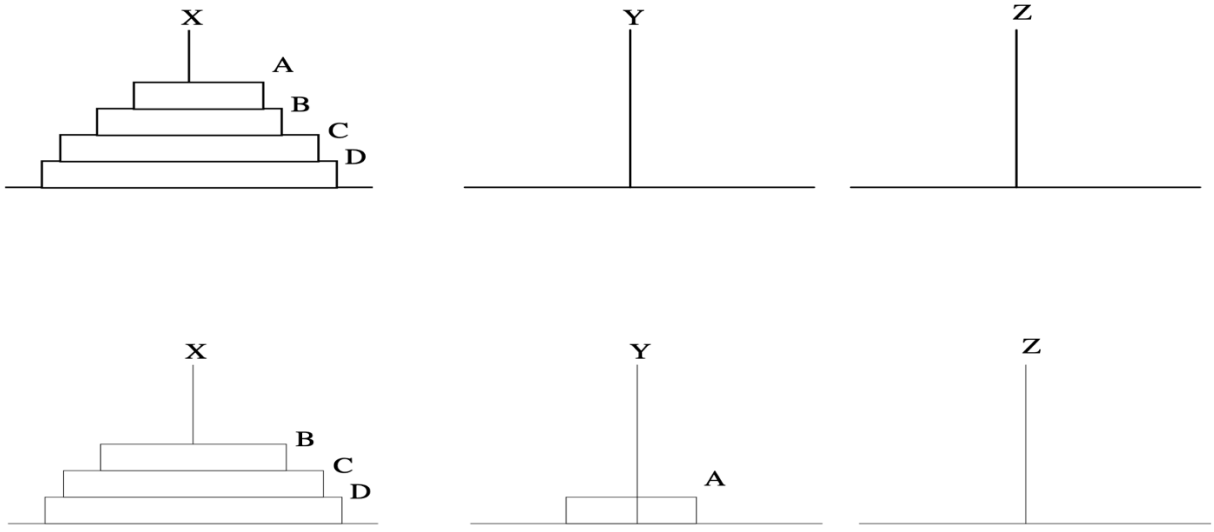
Method to implement Tower of Hanoi Problem:

```
void TOH(int n, char from_peg, char to_peg, char in_peg)
{
    if (n == 1)
    {
        System.out.println("Take disk 1 from peg " +  from_peg + " to peg " + to_peg);
        return;
    }
    TOH(n-1, from_peg, in_peg, to_peg);
    System.out.println("Take disk " + n + " from peg " +  from_peg + " to peg " + to_peg);
    TOH(n-1, in_peg, to_peg, from_peg);
}
```

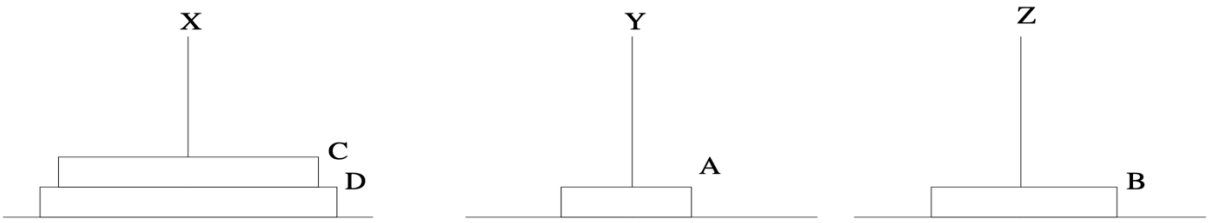
Example 1: with three disks



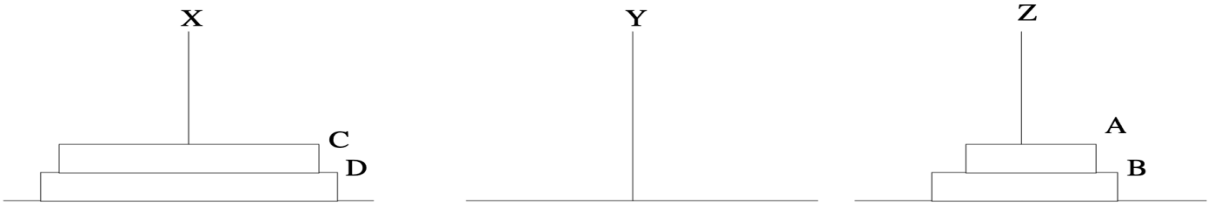
Example 2: with 4 disks



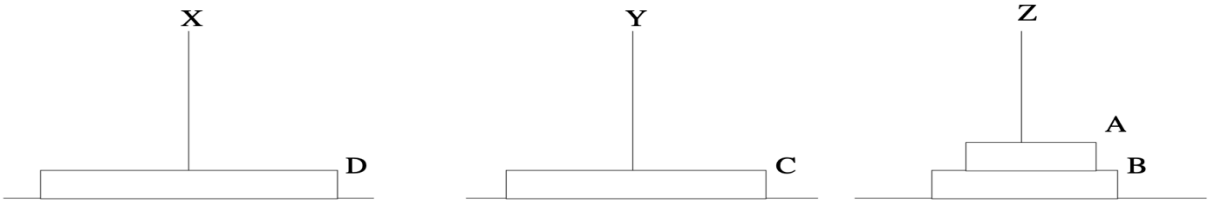
Move disk A from the peg X to peg Y



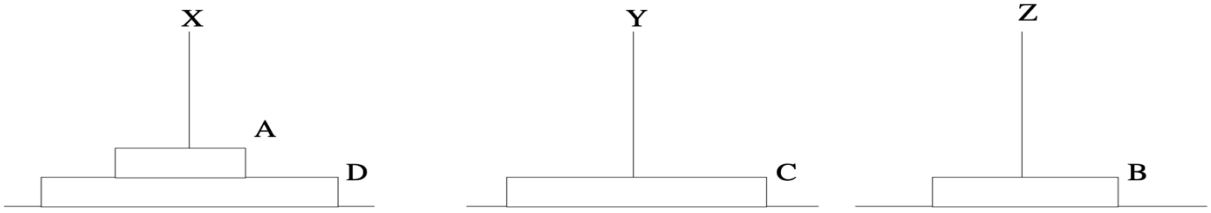
Move disk B from the peg X to peg Z



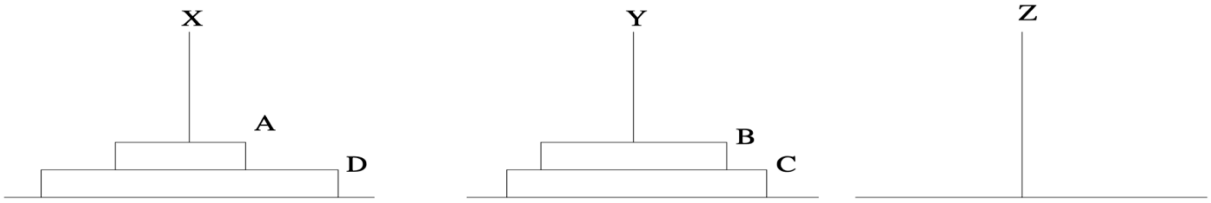
Move disk A from the peg Y to peg Z



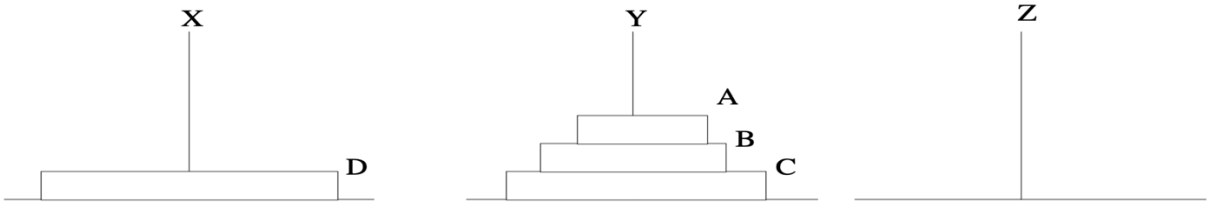
Move disk C from the peg X to peg Y



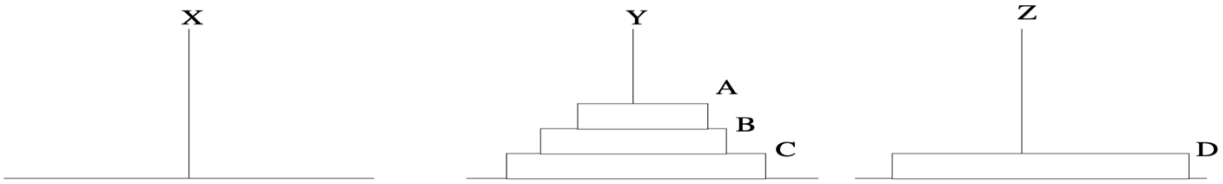
Move disk A from the peg Z to peg X



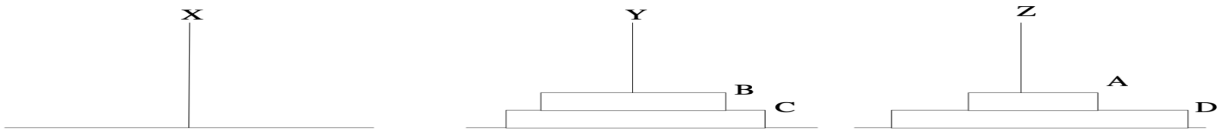
Move disk B from the peg Z to peg Y



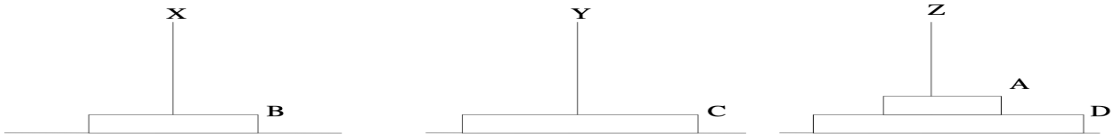
Move disk A from the peg X to peg Y



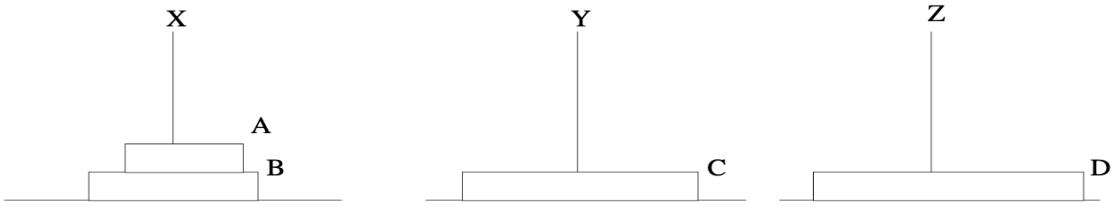
Move disk D from the peg X to peg Z



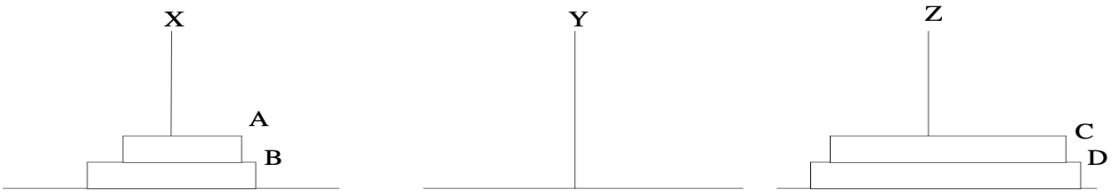
Move disk A from the peg Y to peg Z



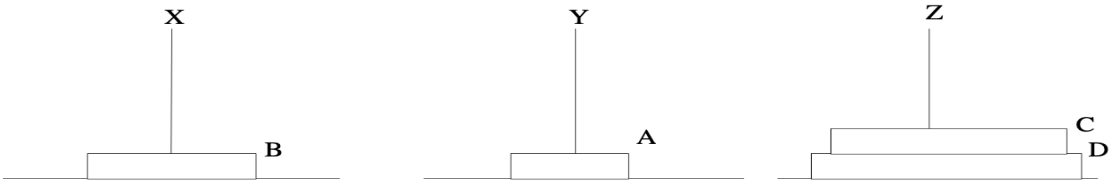
Move disk B from the peg Y to peg X



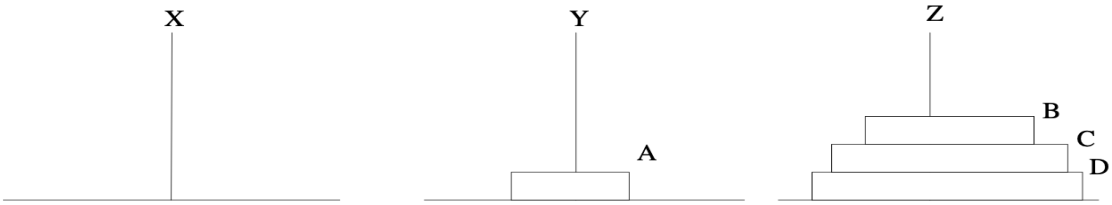
Move disk A from the peg Z to peg X



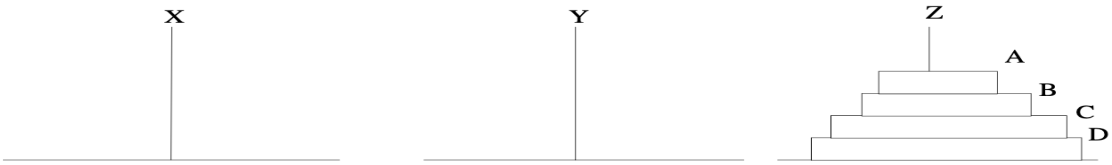
Move disk C from the peg Y to peg Z



Move disk A from the peg X to peg Y



Move disk B from the peg X to peg Z



Greatest Common Divisor (GCD)/ HCF:

- Greatest Common Divisor (GCD) of two numbers is the largest number that divides both of them.

Recursive function to calculate GCD or HCF is as below:

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