

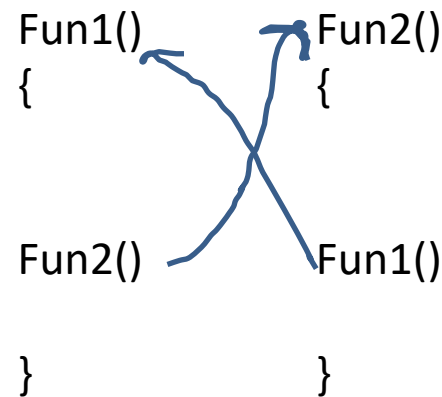
Recursion

- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied
- The process is used for repetitive computations in which each action is stated in terms of a previous result.
- To solve a problem recursively, two conditions must be satisfied.
 - First, the problem must be written in a recursive form
 - Second the problem statement must include a stopping condition

Types of Recursions:

Recursion are mainly of **two types** depending on whether a **function calls itself from within itself** or **more than one function call one another mutually**. The first one is called **direct recursion** and another one is called **indirect recursion**.

```
fun()  
{  
-----  
-----  
-----  
fun()  
}
```



Introduction to Stack

Head Recursion()

```
Fun()  
{  
  Fun();  
  .....  
  .....  
  .....  
}
```

Head & Tail Recursion()

```
Fun()  
{  
  Fun()  
  .....  
  .....  
  .....  
  Fun();  
  
}
```

Tail Recursion()

```
Fun()  
{  
  .....  
  .....  
  .....  
  Fun();  
  
}
```

Recursion

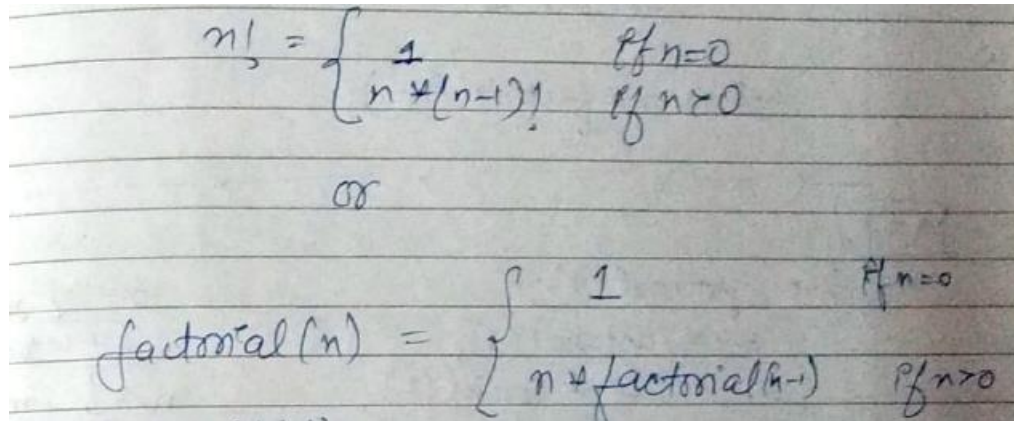
- There is base condition, that stops further calling of the function
- Function call itself directly or indirectly, it should reach towards base condition.

Introduction to Stack

Calculate Factorial of the number using Recursion.

factorial of $n \rightarrow (n!) = \underbrace{n * n-1 * n-2 * \dots * 2 * 1}_{(n-1)!}$

$n! = n * (n-1)! \quad \text{And } 0! = 1$



Handwritten recursive formula for factorial:

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

or

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

Factorial of an integer number using recursive function

```
def recur_factorial(n):  
    if n == 1:  
        return n  
    else:  
        return n*recur_factorial(n-1)  
  
num = int(input("Enter the number: "))  
  
# check if the number is negative  
if num < 0:  
    print("Sorry, factorial does not exist for  
negative numbers")  
elif num == 0:  
    print("The factorial of 0 is 1")  
else:  
    print("The factorial of", num, "is",  
recur_factorial(num))
```

Factorial(5)=

5*Factorial(4)=

5*(4*Factorial(3))=

5*(4*(3*Factorial(2))=

5*(4*(3*(2*Factorial(1))))=

5*(4*(3*(2*(1*Factorial(0))))))=

5*(4*(3*(2*(1*1))))=

5*(4*(3*(2*1)))=

5*(4*(3*2))= 5*(4*6)= 5*24=

120

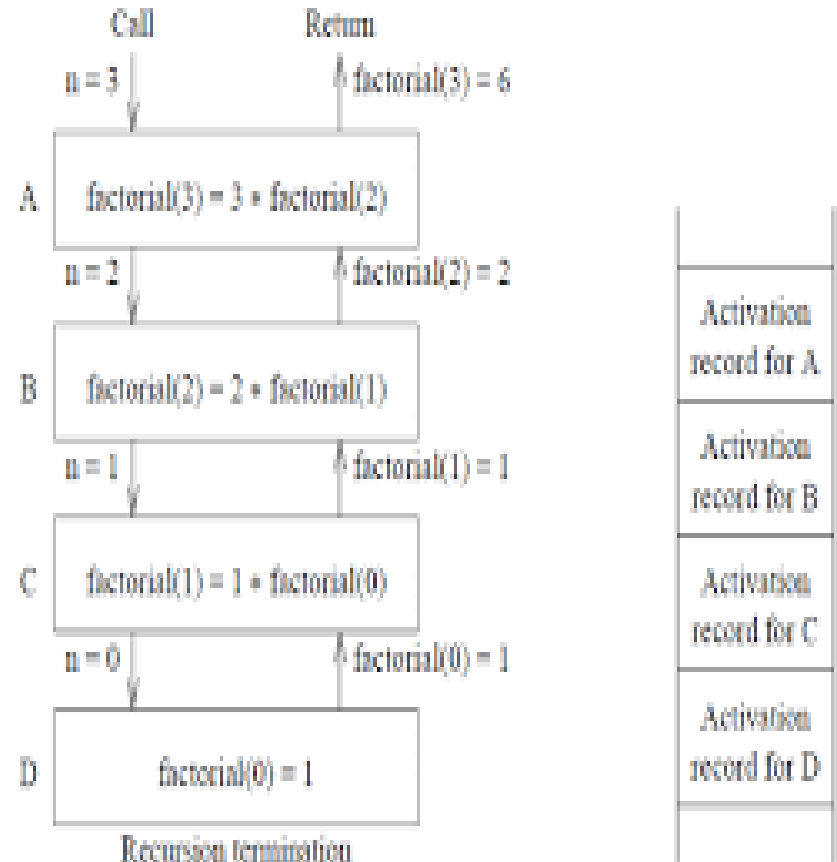
Introduction to Stack

Factorial of an integer number using recursive function

```
def recur_factorial(n):
    if n == 1:
        return n
    else:
        return n*recur_factorial(n-1)

num = int(input("Enter the number: "))

# check if the number is negative
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of", num, "is",
recur_factorial(num))
```

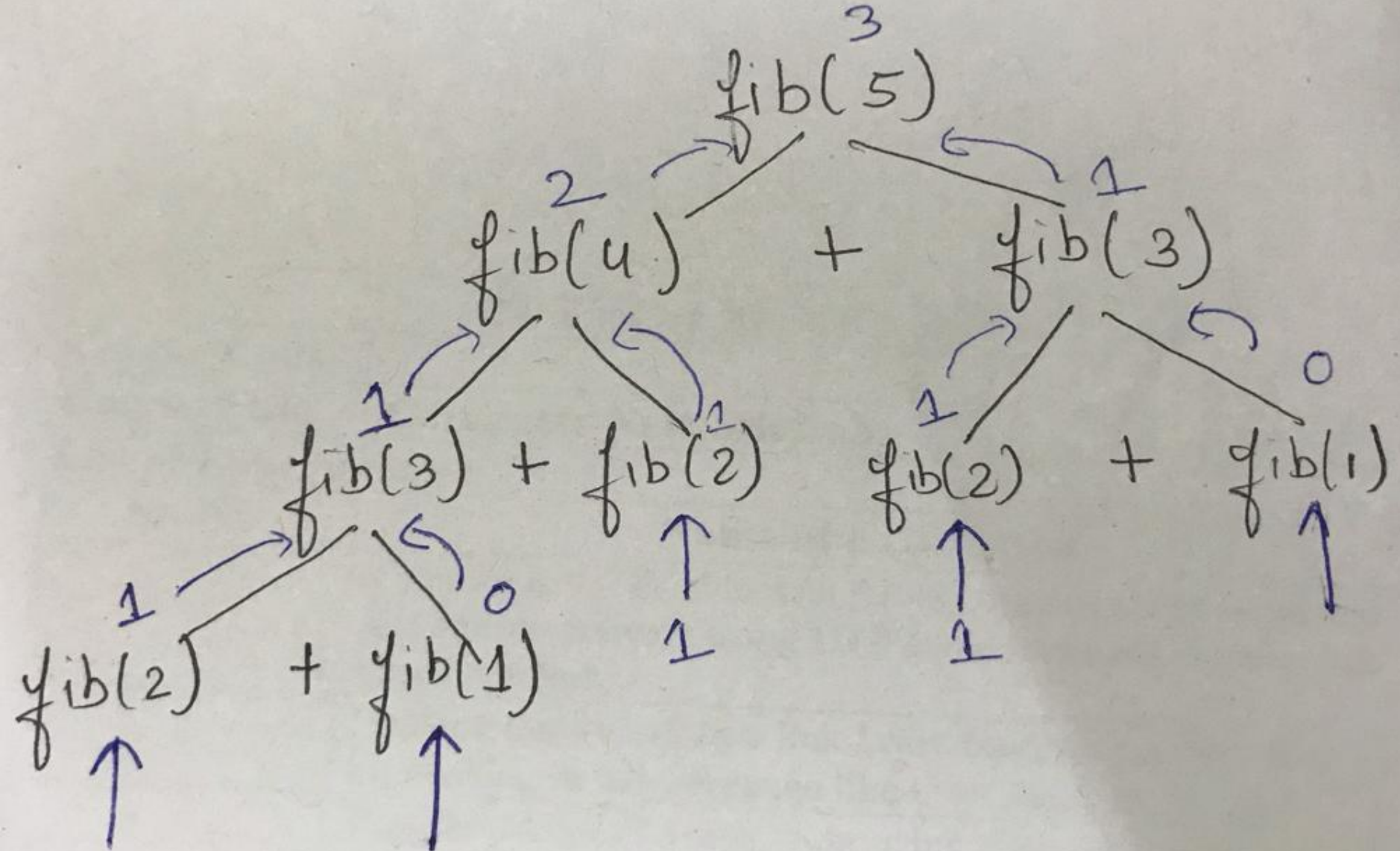


Program for Fibonacci numbers

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
- In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation
 $F_n = F_{n-1} + F_{n-2}$ with seed values $F_0 = 0$ and $F_1 = 1$.


```
def fib_num(n):  
    if n<=0:  
        print("Fibonacci can't be computed")  
    # First Fibonacci number  
    elif n==1:  
        return 0  
    # Second Fibonacci number  
    elif n==2:  
        return 1  
    else:  
        return fib_num(n-1)+fib_num(n-2)  
#input  
n=int(input("Enter n: "))  
print("{}th Fibonacci number is {}".format(n),fib_num(n))
```

Recursion



Recursion Pros and Cons

- **Pros**

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as tower of Hanoi.

- **Cons**

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

Recursion

Binary search

steps: 0



Sequential search

steps: 0



Binary Search Using Recursion

```
def binary_search(my_list, low, high, elem):  
    if high >= low:  
        mid = (high + low) // 2  
        if my_list[mid] == elem:  
            return mid  
        elif my_list[mid] > elem:  
            return binary_search(my_list, low, mid - 1, elem)  
        else:  
            return binary_search(my_list, mid + 1, high, elem)  
    else:  
        return -1
```

```
my_list = [ 1, 9, 11, 21, 34, 54, 67, 90 ]
```

```
elem_to_search = 1
```

```
print("The list is")
```

```
print(my_list)
```

```
my_result = binary_search(my_list,0,len(my_list)-1,elem_to_search)
```

```
if my_result != -1:
```

```
    print("Element found at index ", str(my_result))
```

```
else:
```

```
    print("Element not found!")
```

Tower of Hanoi Problem



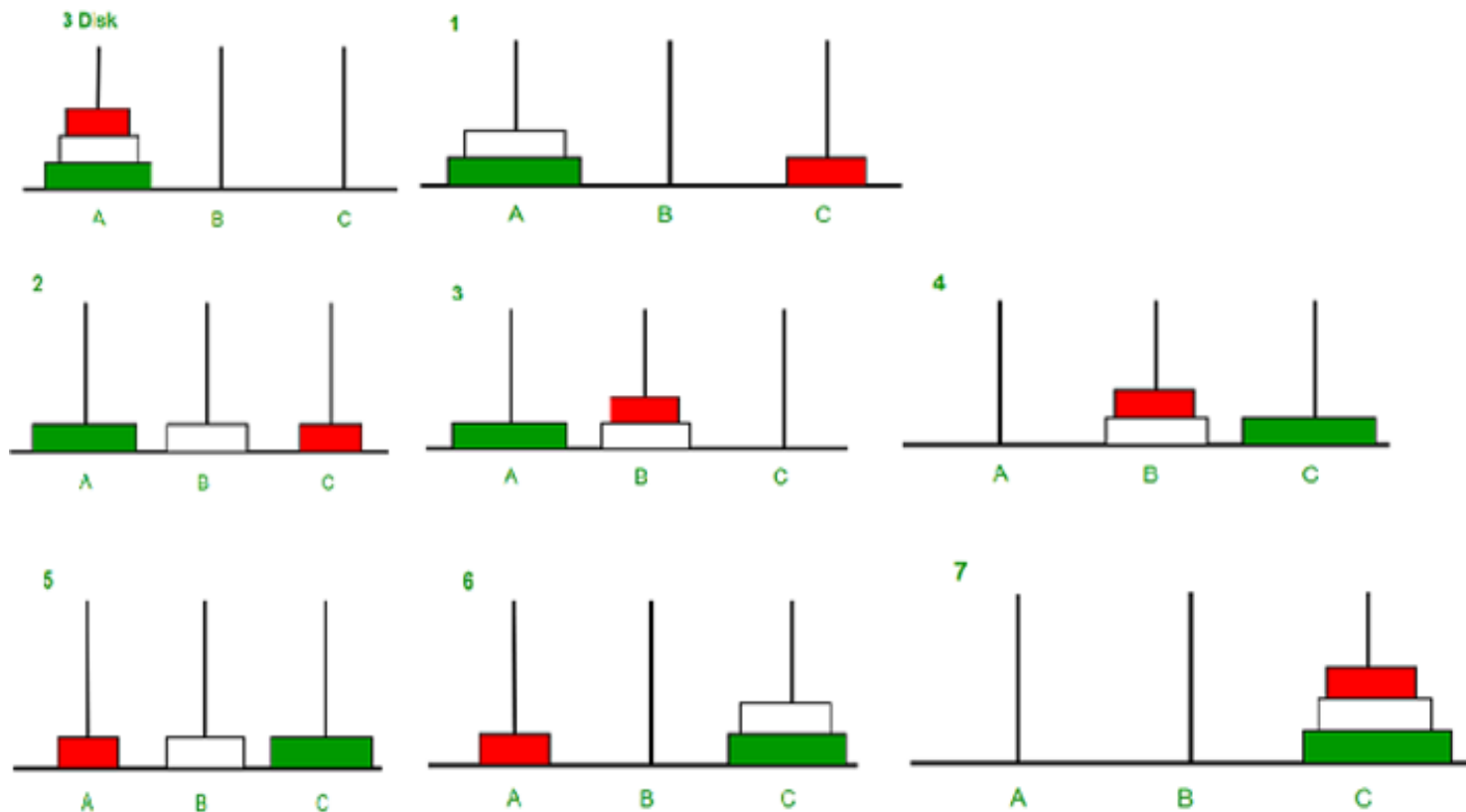
Tower of Hanoi is a mathematical puzzle where we have three rods and n disks.

Tower of Hanoi Problem

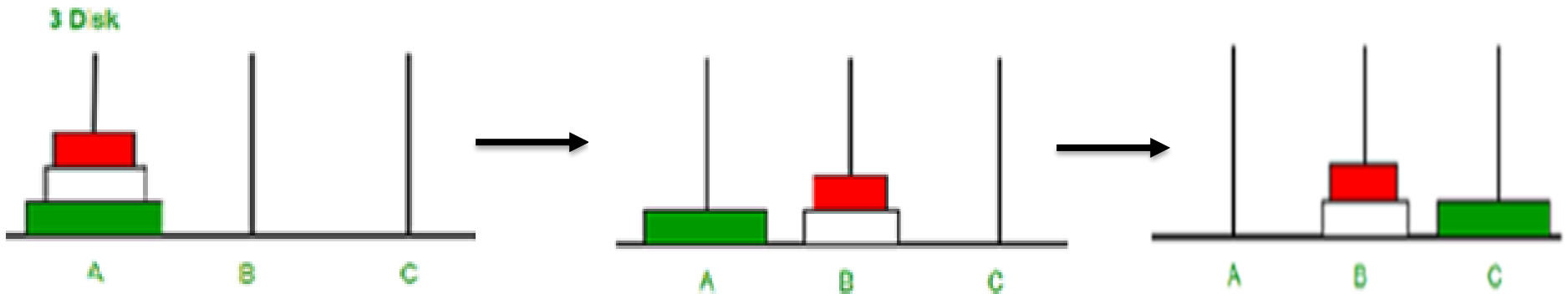
The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- 1) Only one disk can be moved at a time.
- 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- 3) No disk may be placed on top of a smaller disk.

Tower of Hanoi Problem Illustration



Tower of Hanoi Algorithm Concept



Introduction to Stack

A recursive algorithm for Tower of Hanoi can be driven as follows

```
def TowerOfHanoi(n , source, destination, auxiliary):  
    if n==1:  
        print ("Move disk 1 from source",source,"to destination",destination)  
        return  
    TowerOfHanoi(n-1, source, auxiliary, destination)  
    print ("Move disk",n,"from source",source,"to destination",destination)  
    TowerOfHanoi(n-1, auxiliary, destination, source)
```

Driver code

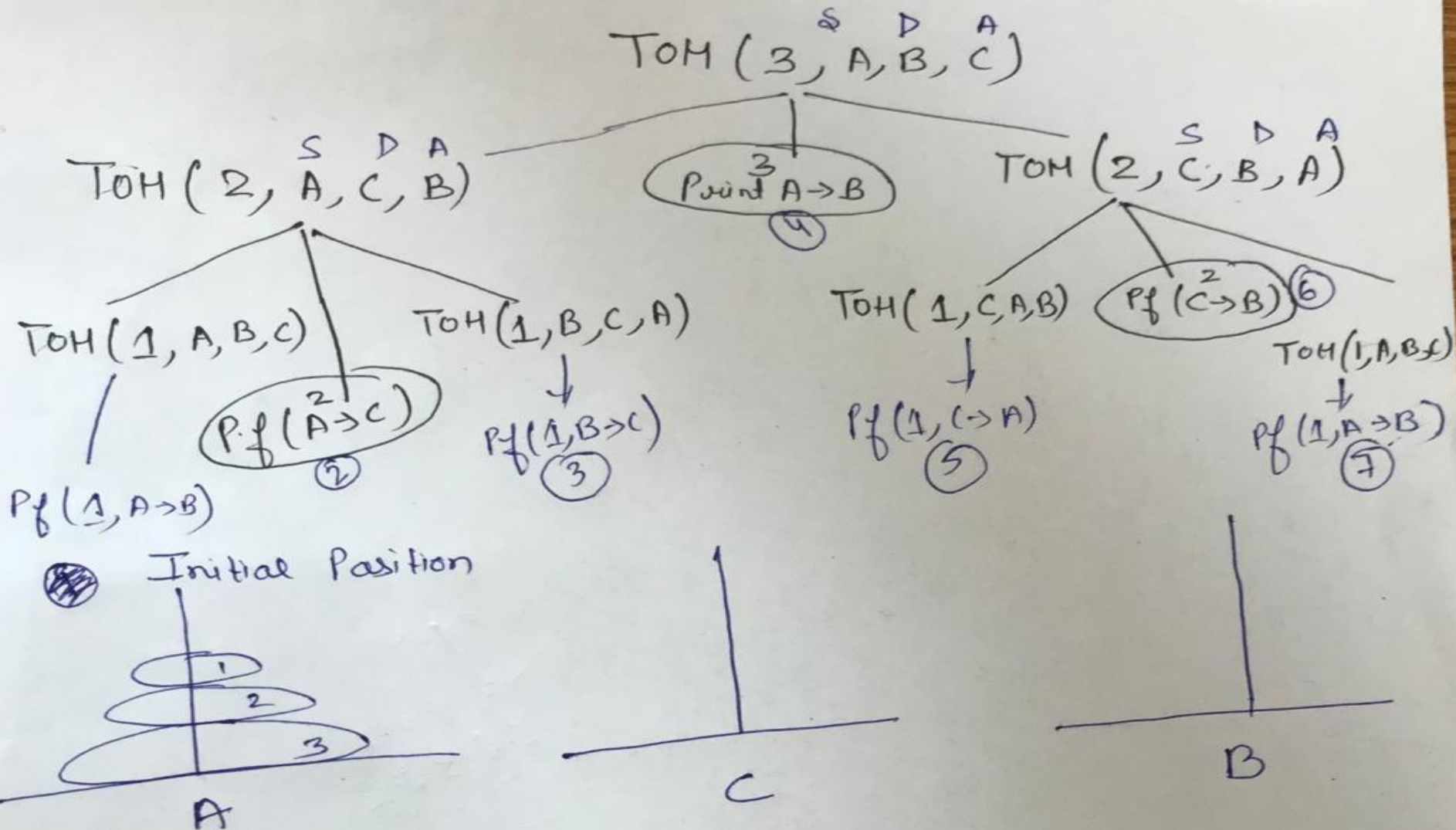
n = 3

TowerOfHanoi(n,'A','B','C')

A, C, B are the name of rods

Introduction to Stack

Recursive Tree

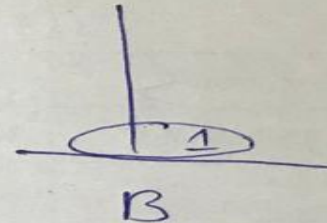
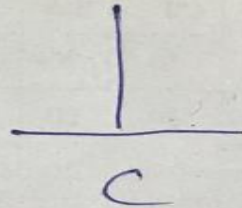
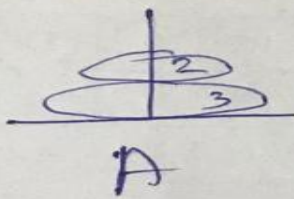


Introduction to Stack

Moves

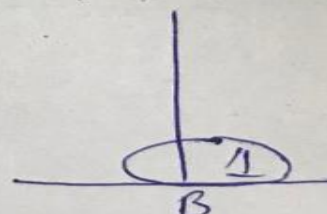
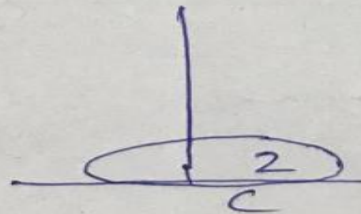
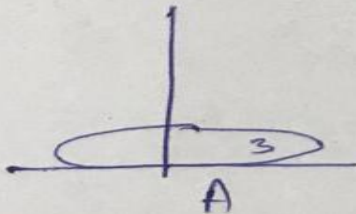
- 1) 1, $A \rightarrow B$
- 2) 2, $A \rightarrow C$
- 3) 1, $B \rightarrow C$
- 4) 3, $A \rightarrow B$
- 5) 1, $C \rightarrow A$
- 6) 2, $C \rightarrow B$
- 7) 1, $A \rightarrow B$

1)



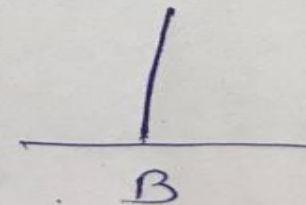
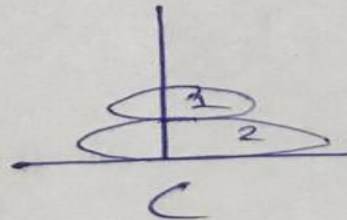
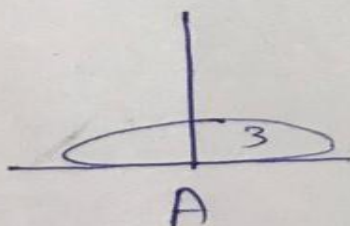
1, $A \rightarrow B$

2)



2, $A \rightarrow C$

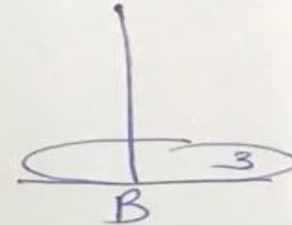
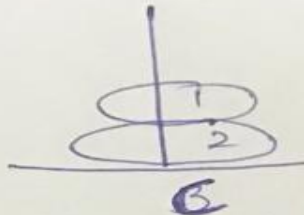
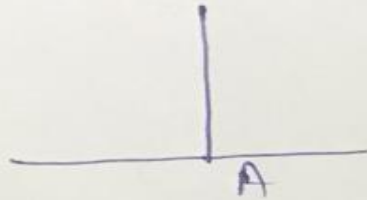
3)



1, $B \rightarrow C$

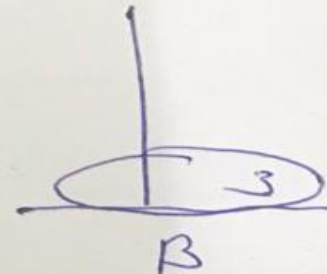
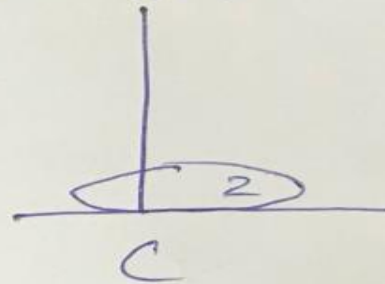
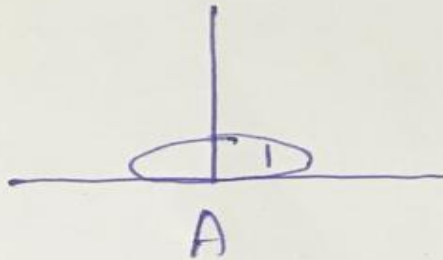
Introduction to Stack

4)



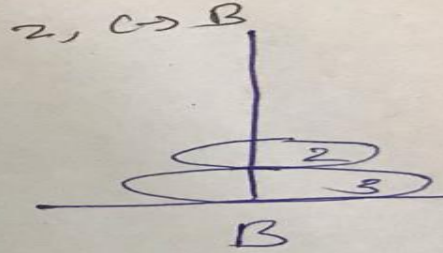
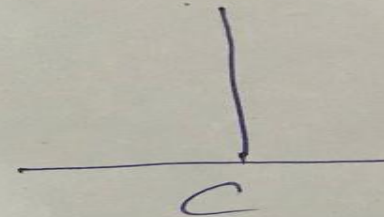
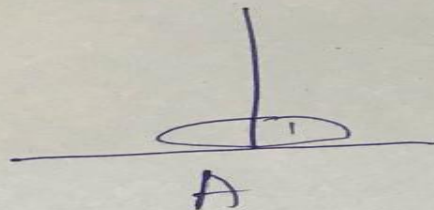
3, $A \rightarrow B$

5)



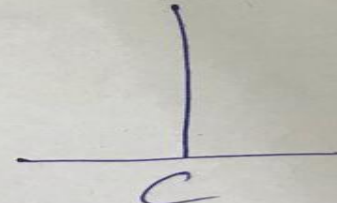
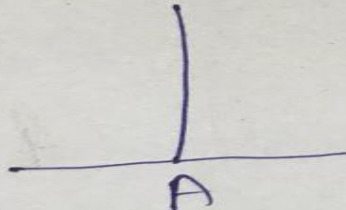
1, $C \rightarrow A$

6)

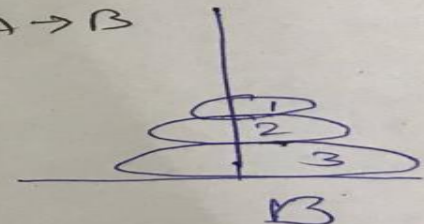


2, $C \rightarrow B$

7)



1, $A \rightarrow B$



- Difference Between Recursion and Iteration**

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.

- Difference Between Recursion and Iteration**

BASIS FOR COMPARISON	RECURSION	ITERATION
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.

- Difference Between Recursion and Iteration**

BASIS FOR COMPARISON	RECURSION	ITERATION
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.