General Guideline

ABES
Estd. 2000

Introduction to Recursion

Introduction to Recursion

- Let us consider the given scenario:

A child couldn't sleep, so his mother told him a story about a little frog,

Little frog couldn't sleep, so frog's mother told him a story about a little bear,

Little bear couldn't sleep, so the bear's mother told him a story about a little weasel...

who fell asleep.

… and the little bear fell asleep

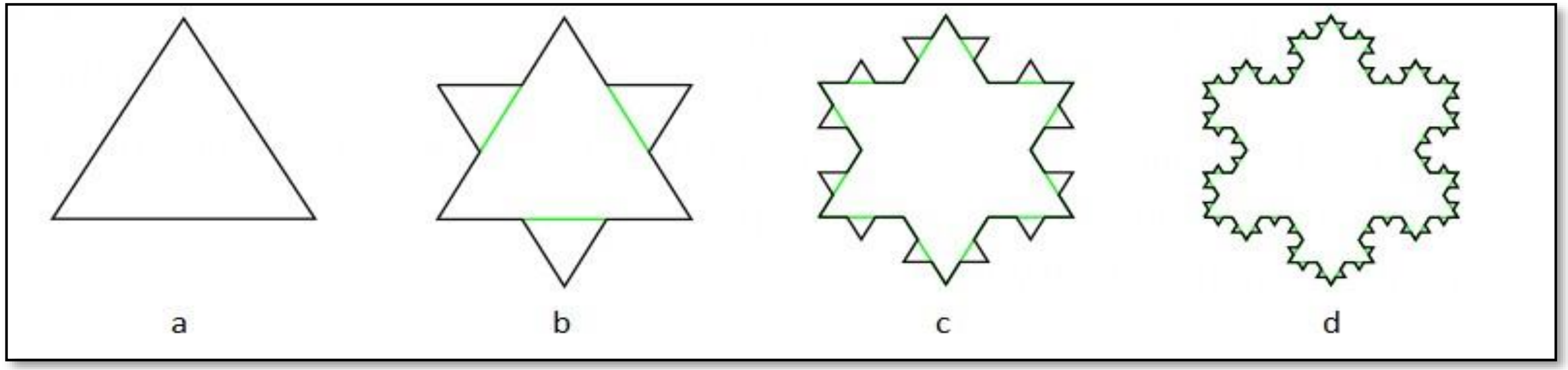
... and the little frog fell asleep

... and the child fell asleep.

Continue..

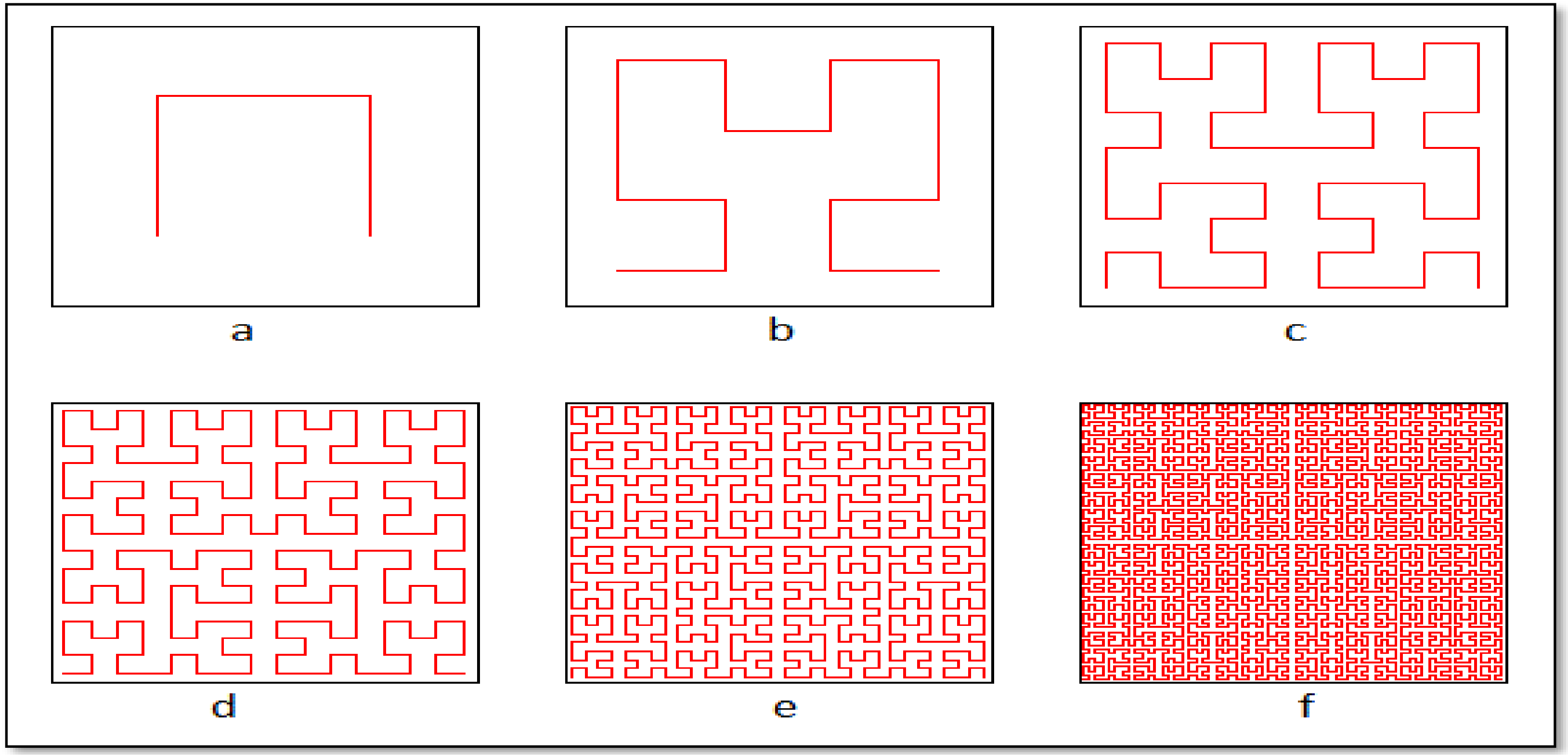
- Recursion occurs when a thing is defined in terms of itself or of its type.

  Refer to the diagrams given below showing the growth of the triangle recursively. A triangle is added on each edge of the triangle in the next iteration.



Continue..

The Hilbert space finding curve grows recursively by increasing the number of elements by 4 times in each step.



Applications of Recursion

- The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its definition.

In mathematics and computer science, a method exhibits recursive behavior when two properties can define it:

- A simple base case (or cases) — a terminating scenario that does not use recursion to produce an answer
- A recursive step — a set of rules that reduces all successive cases toward the base case.

Functional Recursion

- With respect to a programming language, recursion is defined as: “When a function calls itself (either directly or indirectly) then it is called recursive function and process is called recursion.”

Points to remember: Recursive function performs some part of the task and delegates the rest of it to subsequent recursive calls.

- Recursion is a problem solving technique where the solution of larger problems is defined in smaller instances of itself.
- Recursion always has a terminating condition (base condition); otherwise, it will fall in infinite loop.
- Recursive function performs some part of the task and delegates the rest of it to subsequent recursive calls.

Example of Functional Recursion

- A familiar example is a **Factorial number**: $F(n)$=n*$F(n-1)$. For such a definition to be useful, it must be reducible to non-recursively defined values (Base case): in this case $F(0) = 1$.
- Another example is the generation of **power (an):** $F(a, n)$=a*$F(a, n-1)$. The function must be reducible to non-recursively defined values (Base case): in this case $F(a, 0)$=1
- Another example is the **Fibonacci number** sequence: $F(n) = F(n-1) + F(n-2)$. For such a definition to be useful, it must be reducible to non-recursively defined values (Base case): in this case $F(0) = 0$ and $F(1) = 1$.

How to write Recursive Functions?

- We should first write the mathematical representation of the solution along with the base case.

  Example: Sum of N natural numbers

  If we have to find the sum of natural numbers up to N terms, this can be written mathematically as

$$
\mathrm{Sum}(N) = \begin{cases} N + \mathrm{Sum}(N-1) & \text{if } N>1 \\ 1 & \text{if } N=1 \end{cases}
$$

ALGORITHM Sum(N)

Input: Any positive number N

Output: Sum of first N natural Number

BEGIN:

```
    IF N==1 THEN
        RETURN 1                 } Base case
    ELSE
        RETURN N+Sum(N-1)        } Recursive Call
END;
```

Another Example of sum of N Numbers.

$$Sum(N) = \begin{cases} N + Sum(N-1) & \text{if } N>1 \\ 0 & \text{if } N=0 \\ 1 & \text{if } N=1 \end{cases}$$

- In recursion, always check boundary conditions.

ALGORITHM Sum(N)

Input: Any positive number N

Output: Sum of first N natural Number

BEGIN:

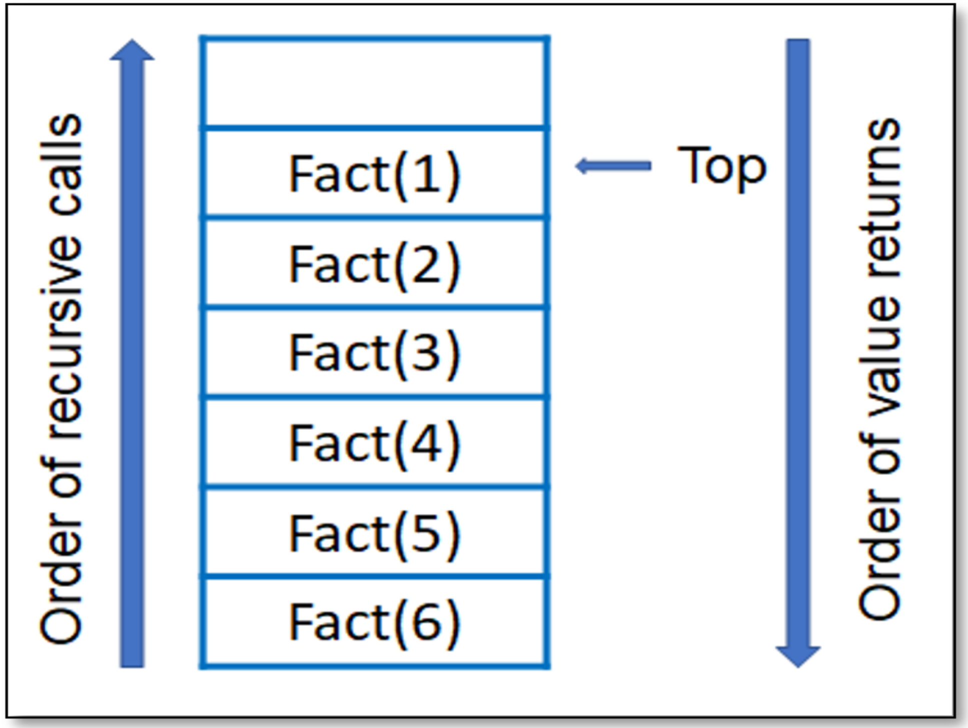
```
    IF N==0 THEN          } Base case
        RETURN 0
    IF N==1 THEN          } Base case
        RETURN 1
    ELSE
        RETURN N+Sum(N-1) } Recursive Call
```

END;

Factorial of a given Number

Mathematically, factorial N is defined as N*Factorial of N-1.

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



Example

6! = 6*5!

5! = 5*4!

4! = 4*3!

3! = 3*2!

2! = 2*1!

1! = 1*0!

0! = 1 (base case)

If we compute the factorial in reverse order, 1! = 1*1 = 1

2! = 2*1 = 2

3! = 3*2 = 6

4! = 4*6 = 24

5! = 5*24 = 120

6! = 6*120 = 720

Algorithm of Factorial with Recursion

ALGORITHM Fact (N)

Input: Any positive number N

Output: factorial of N

```
BEGIN:
    IF N==0 || N==1 THEN          }  Base case
        RETURN 1
    ELSE
        RETURN N* Fact (N-1)      }  Recursive Call
END;
```

Example 2 - Computation of Power (a^N)

- In the compiler, there is no arithmetic operator defined as power. Power is simulated with repeated multiplications.

$$\text{Power}(a, N) = \begin{cases} a * \text{Power}(N-1) & \text{if } N>0 \\ 1 & \text{if } N=1 \end{cases}$$

Example

- $3^4 = 3 * 3^3$
- $3^3 = 3 * 3^2$
- $3^2 = 3 * 3^1$
- $3^1 = 3 * 3^0$
- $3^0 = 1$

Computing the power in reverse approach

- $3^1 = 3 * 1 = 3$
- $3^2 = 3 * 3 = 9$
- $3^3 = 3 * 9 = 27$
- $3^4 = 3 * 27 = 81$

Algorithm for computation of power

ALGORITHM Power (a, N)

Input: Base a and Power N

Output: a raised to power N

BEGIN:

```
        IF N==0 THEN
                RETURN 1
        ELSE
                RETURN a*Power(a, N-1)
```

END;

Example 3- Computation of greatest common divisor (GCD)

Greatest common divisor of two numbers can be generated with two approaches:

- subtraction method
- modulus method.

1. Subtraction method

$$\text{Gcd}(a, b) = \begin{cases} \text{Gcd}(a-b, b) & \text{if } a>b \\ \text{Gcd}(a, b-a) & \text{if } b>a \\ a & \text{if } a=b \end{cases}$$

Algorithm of Subtraction Method of GCD Calculation

```
ALGORITHM Gcd(a, b)
Input: Two positive numbers a and b
Output: Greatest common divisor of a and b
BEGIN:
    IF a==b THEN                  Base case
        RETURN a
    ELSE
        IF a>b THEN
            RETURN Gcd(a-b, b)    Recursive Call
        ELSE
            RETURN Gcd(a, b-a)    Recursive Call
END;
```

a	b	Remarks	Action
54	16	a>b	a=a-b
38	16	a>b	a=a-b
22	16	a>b	a=a-b
6	16	b>a	b=b-a
6	10	b>a	b=b-a
6	4	a>b	a=a-b
2	4	b>a	b=b-a
2	2	a=b	Gcd is 2

Continue

2. **Modulus Method**

$$Gcd(a, b) = \begin{cases} Gcd(a\%b, b) & \text{if } a>b \\ Gcd(a, b\%a) & \text{if } b>a \\ a & \text{if } b=0 \\ b & \text{if } a=0 \end{cases}$$

a	b	Remarks	Action
if 54	16	a>b	a=a%b
6	16	b>a	b=b%a
6	4	a>b	a=a%b
2	4	b>a	b=b%a
2	0	b=0	Gcd is 2

Algorithm of Modulus Method of GCD Calculation

```
ALGORITHM Gcd(a, b)
Input: Two positive numbers a and b
Output: Greatest common divisor of a and b
BEGIN:
        IF a==0 THEN
                RETURN b                        } Base case
        ELSE
                IF b==0 THEN
                        RETURN a                } Base case
                ELSE
                        IF a>b THEN
                                RETURN Gcd(a%b, b)  } Recursive Call
                        ELSE
                                RETURN Gcd(a, b%a)  } Recursive Call
END;
```

Types of Recursion

- Head Recursion
- Tail Recursion
- Tree Recursion
- Mixed Recursion

Head & Tail Recursion

- In this type of recursion, the Recursive function performs some task and calls itself. If the call is made before function performs its task, then it is called head recursion.
- On the other hand, if the function performs its task first and is followed by a recursive call, then it is tail recursion.

Example of Head Recursion

- **ALGORITHM** Fun(N)

Input: Any positive number N

Output: print number N to 1

BEGIN:

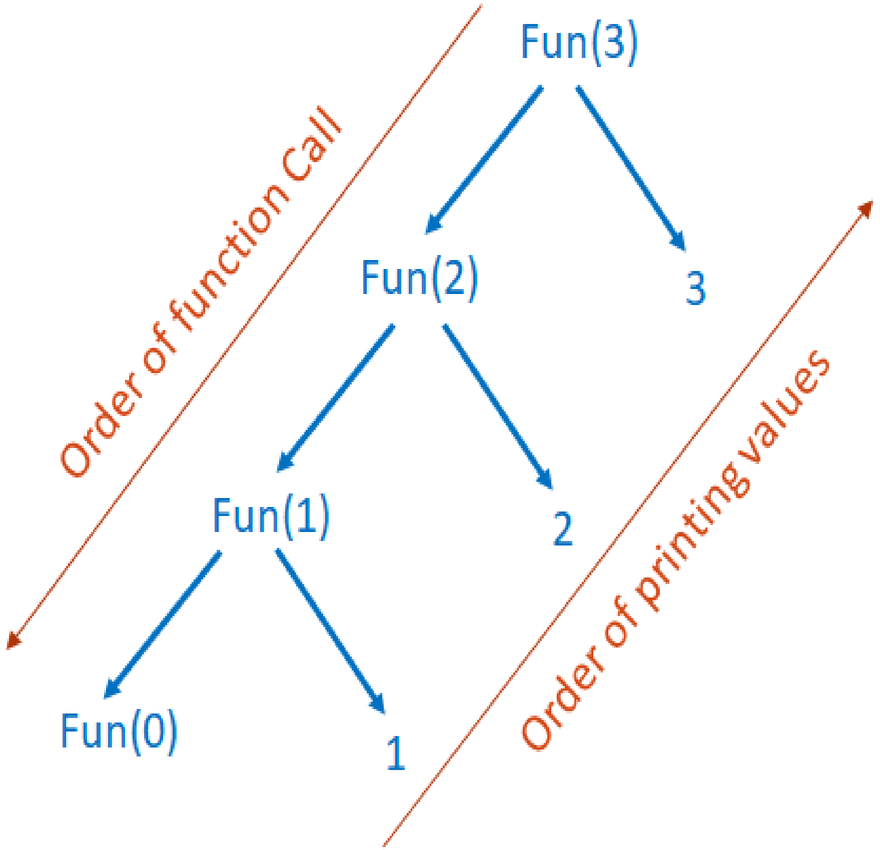
IF N>0 THEN

Fun(N-1)

WRITE("N")

Recursive Call

END



Example Tail recursion

ALGORITHM Fun(N)

Input: Any positive number N

Output: print number N to 1

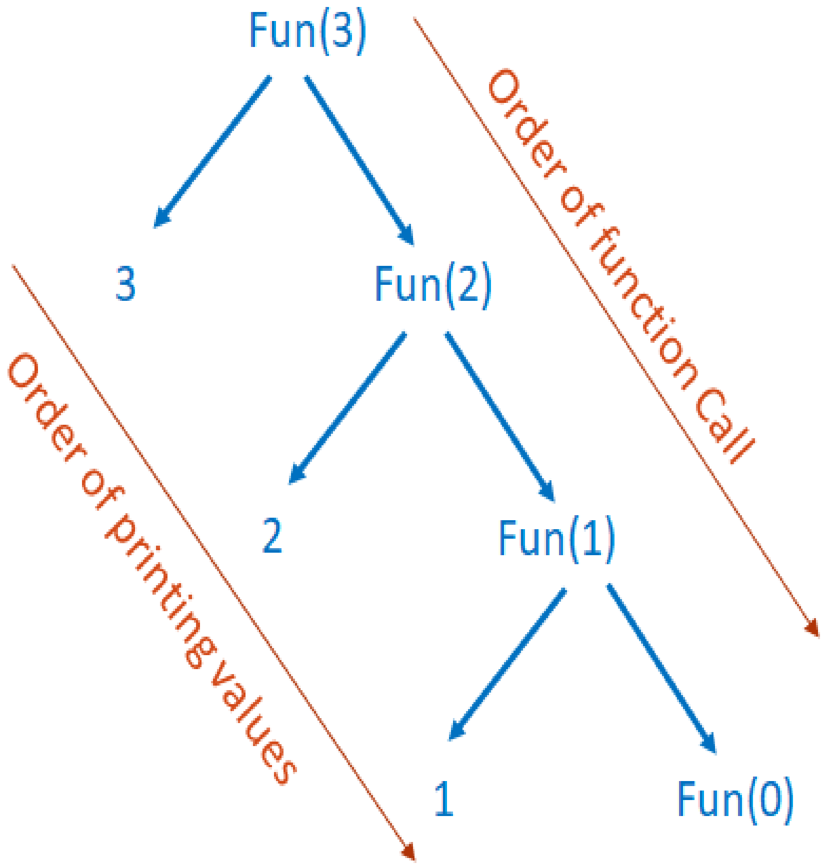
BEGIN:

IF N>0 THEN

WRITE(“N”)

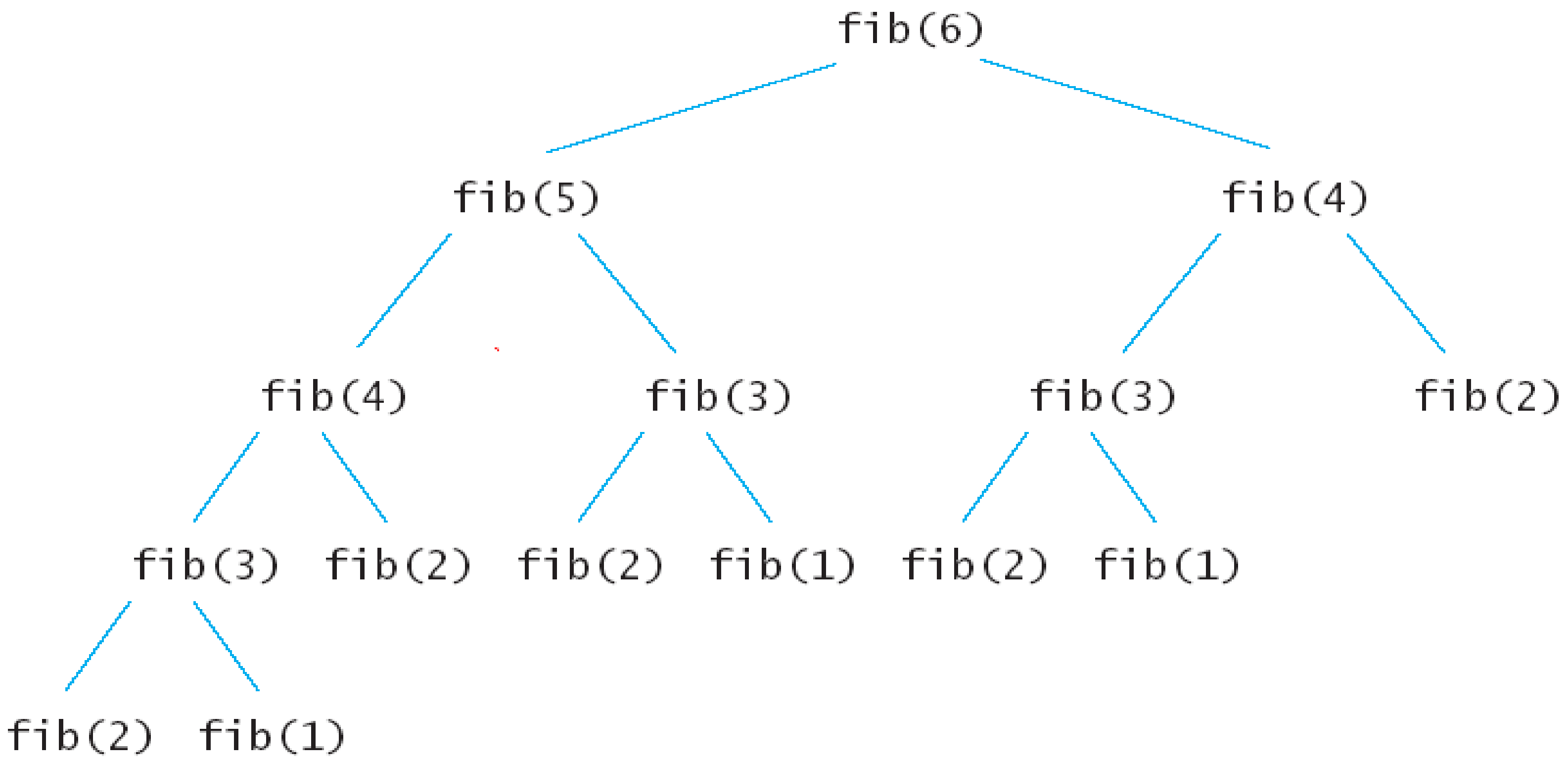
Fun(N-1) — Recursive Call

END;



Tree Recursion

- The tree recursion is a type of recursion in which recursive calls grow in the form of a tree. Let us recap the generation of nth Fibonacci term.



Continue..

ALGORITHM Fib(N)

Input: Any positive number N

Output: Nth Fibonacci term

BEGIN:

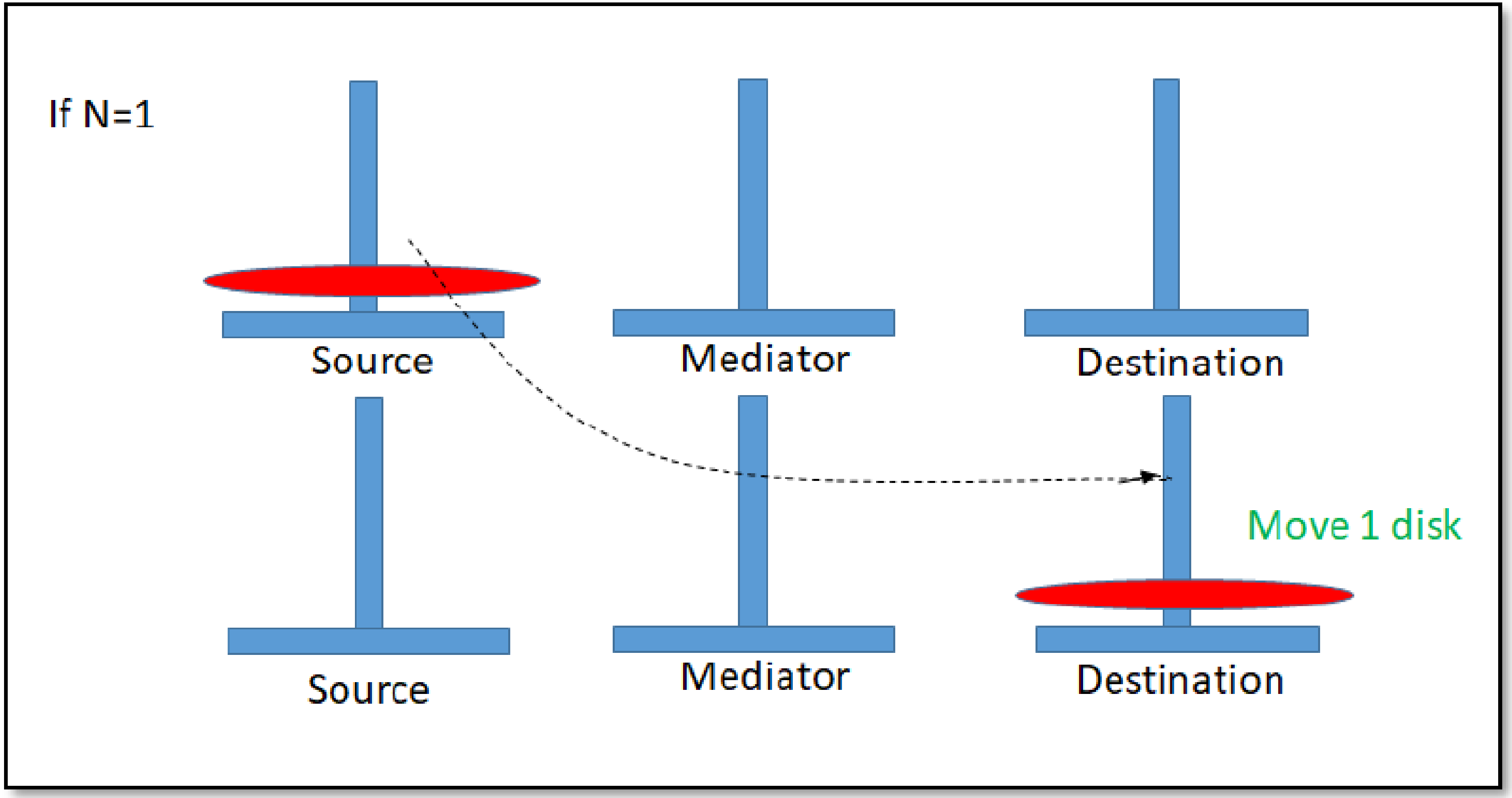
```
        IF N == 1 THEN
                RETURN 0        } Base case
        IF N == 2 THEN
                RETURN 1        } Base case
        ELSE
                RETURN Fib(N-1) + Fib(N-2)   } Recursive Call
END;
```

Mixed Recursion

- Mixed recursion is the recursion that utilizes the concept of Head, Tail and Tree recursion. To understand the concept of Mixed Recursion, let us take an interesting concept of Towers of Hanoi.
- A arrangement is made in the Buddhist temple in Hanoi (Vietnam). The Monk makes one move per day according to the rule that a bigger disk cannot be kept above the smaller one. The puzzle is named under the Hanoi and hence known as towers of Hanoi.
  - ✓ There are three Pegs: Source, Destination and Mediator and there are n disks (of different sizes). All disks are initially at source (in the order of bigger to smaller).
  - ✓ Our aim is to move all disks from Source to Destination using Mediator but never place a larger disk on the top of a smaller one.
  - ✓ At a time, only one move is allowed.

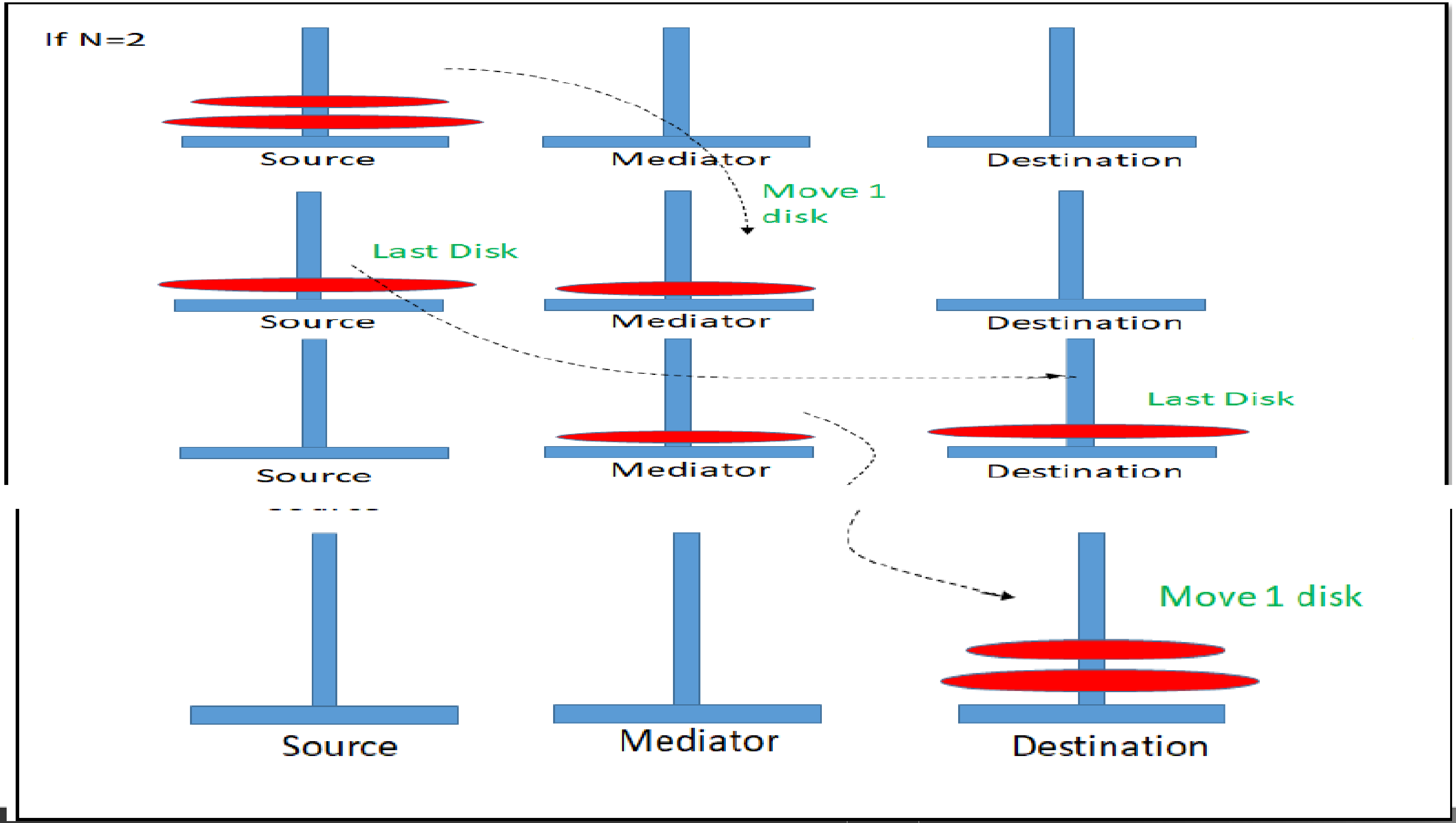
Tower of Hanoi with 1 Disk

If there is only 1 disk at source Peg



Tower of Hanoi with 2 Disks

If there are 2 disks at source Peg



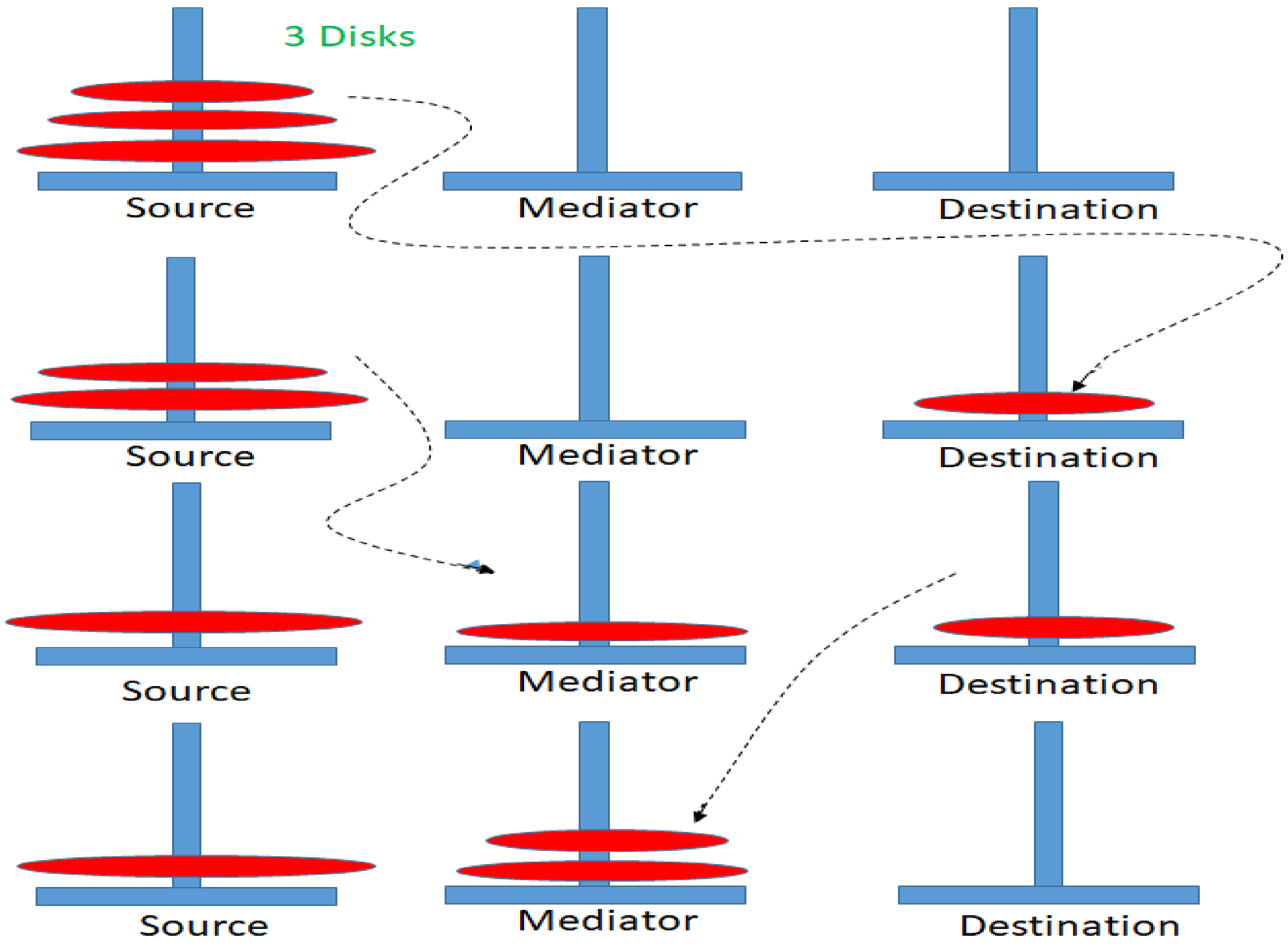
Can you solve ??

- Tower of Hanoi with 3 disks.

Solution of Tower of Hanoi with 3 Disks

If N=3

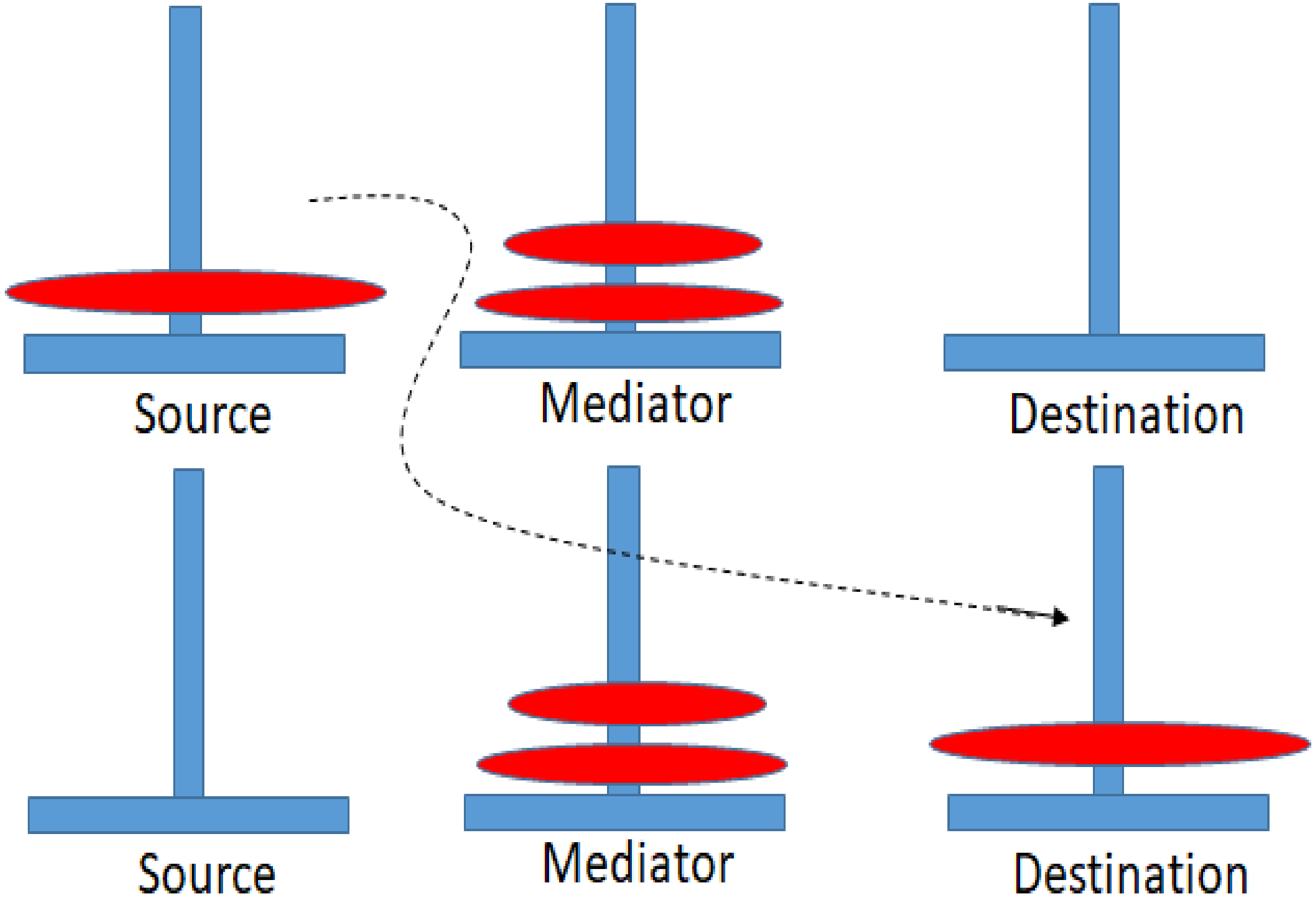
Step-1
Move 2 disk from source to Mediator recursively



Continue..

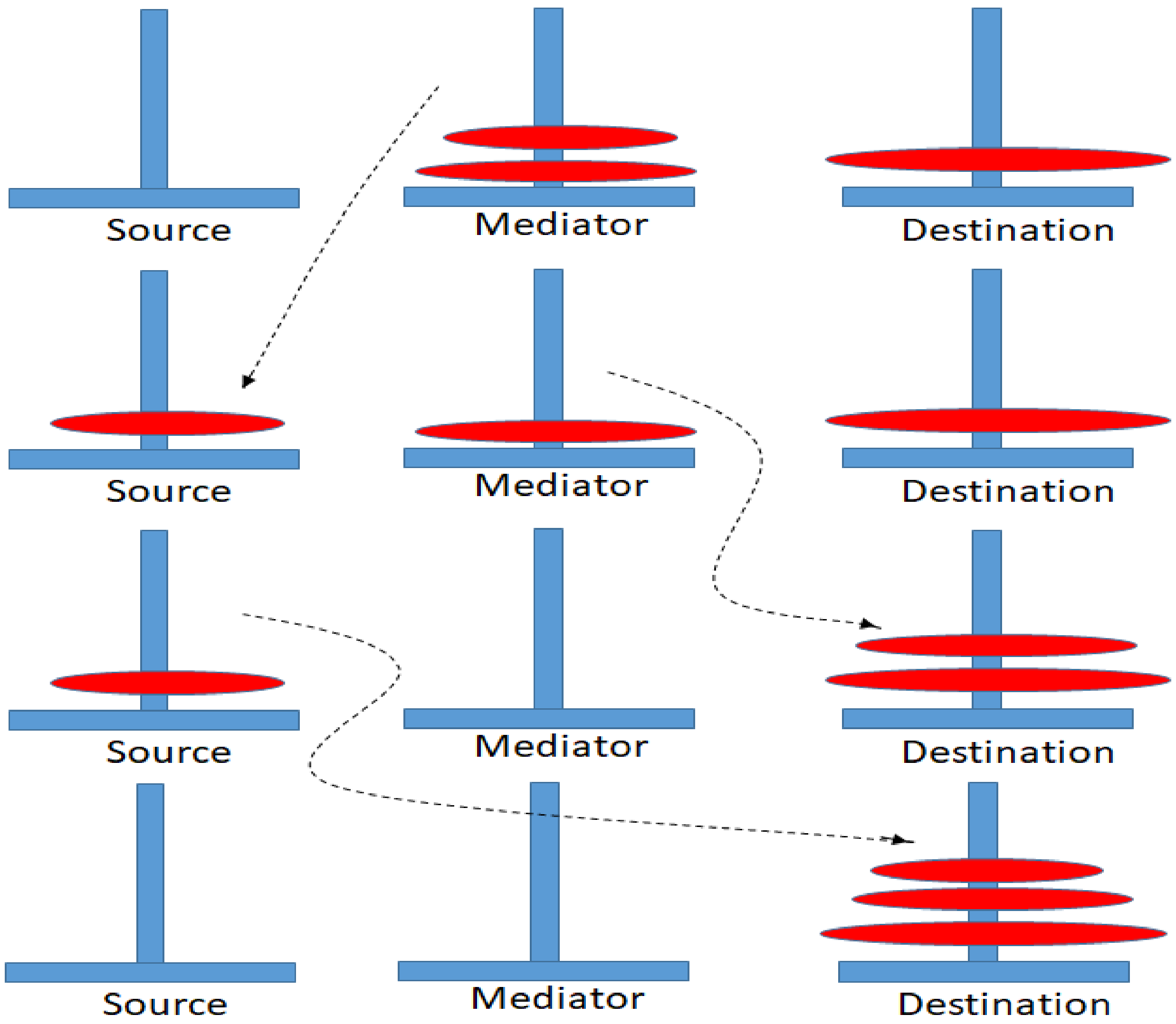
Step-2

Move last disk from source to destination



Continue..

Step-3
Move 2 disk from Mediator to destination



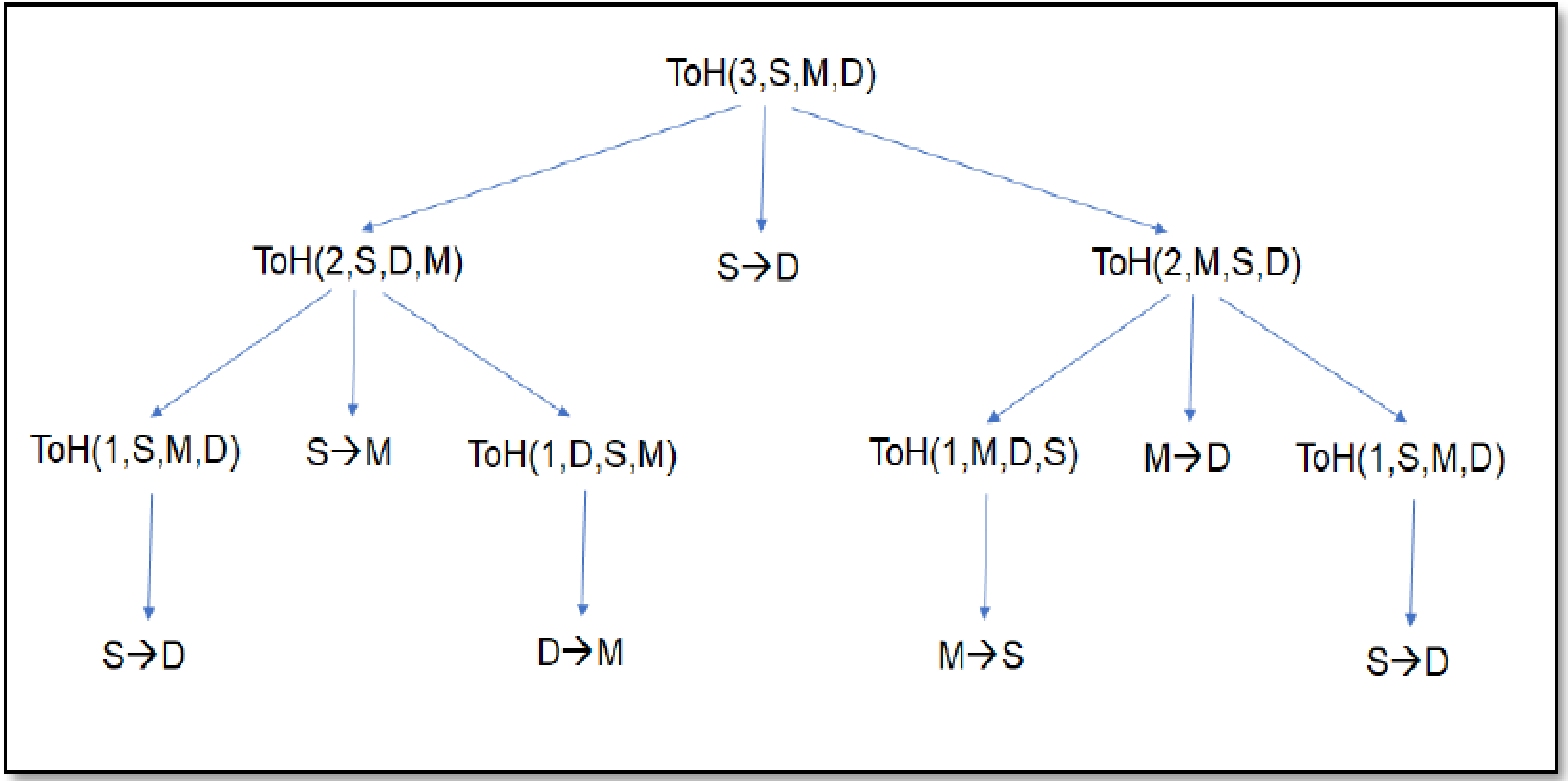
Algorithm of Tower of Hanoi

ALGORITHM ToH (S, M, D, n)
Input: A number n and three characters representing Source, Mediator, Destination Peg
Output: Disk movement from peg to peg
BEGIN:

```
    IF n == 1 THEN
        WRITE("Move disk from Source (S) to Destination (D)")
    ELSE
        ToH(S, D, M, n-1)
        WRITE("Move disk from Source (S) to Destination (D)")
        ToH(M, S, D, n-1)
```

END;

Time Complexity of Tower of Hanoi



Sequence of steps performed:

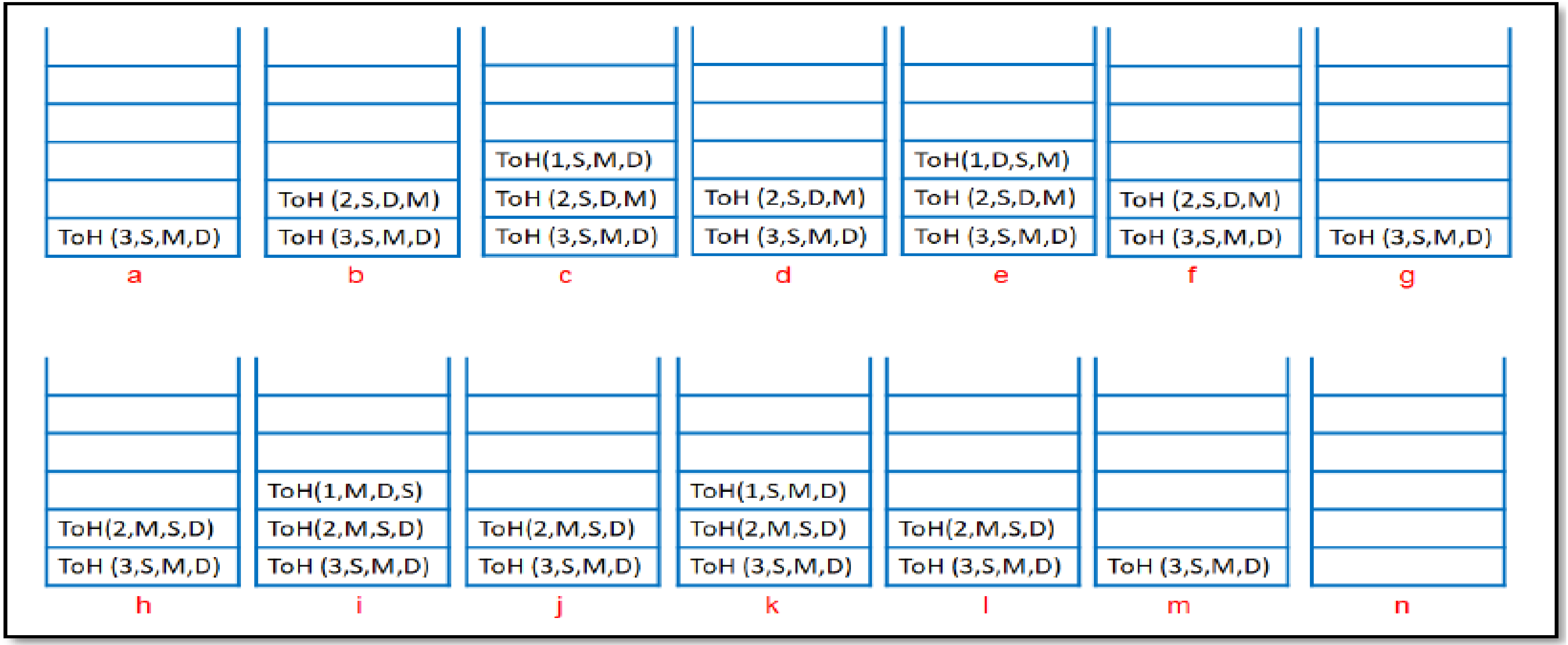
1. S→D
2. S→M
3. D→M
4. S→D
5. M→S
6. M→D
7. S→D

Continue..

- Total Function Calls : 7
- Total Push in Call Stack : 7
- Total Pop in Call Stack : 7
- For N disks on source peg, there will be 2N-1 function calls.
- Therefore, total Push and Pop operations will be 2*(2N-1) i.e., 4N – 2.
- Since Push and pop operations take constant time, the Time complexity will be Θ(N).

Space Complexity of Tower of Hanoi

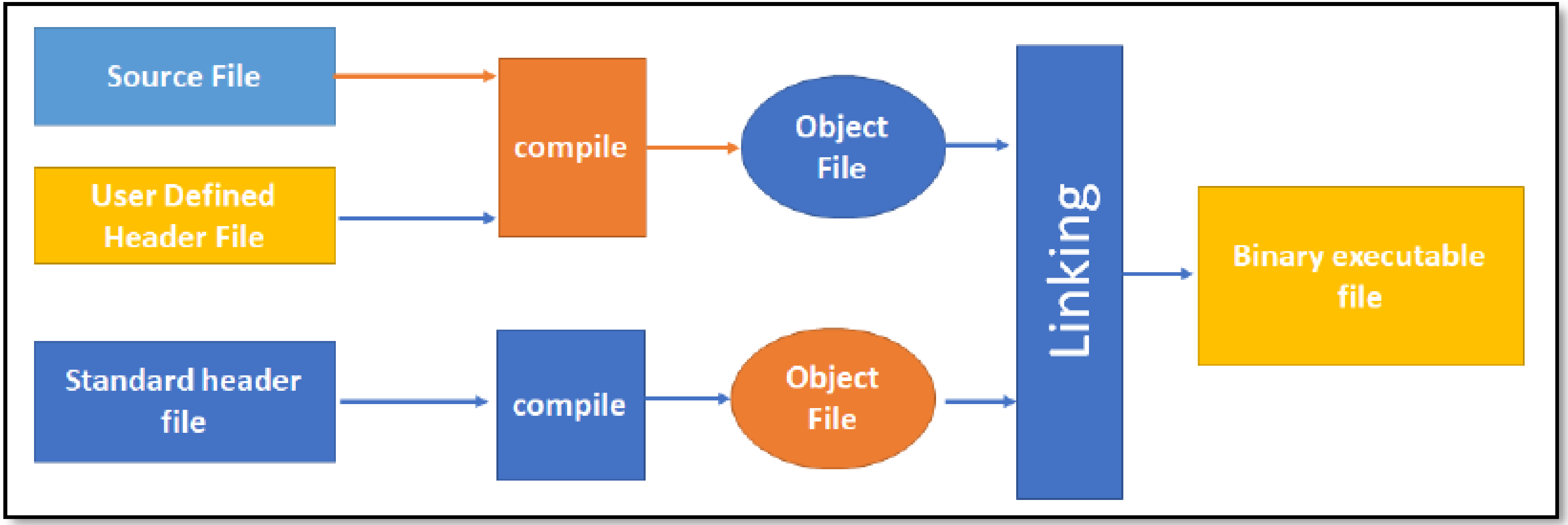
To understand the space complexity, let us see the maximum number of pending activation records at any moment.



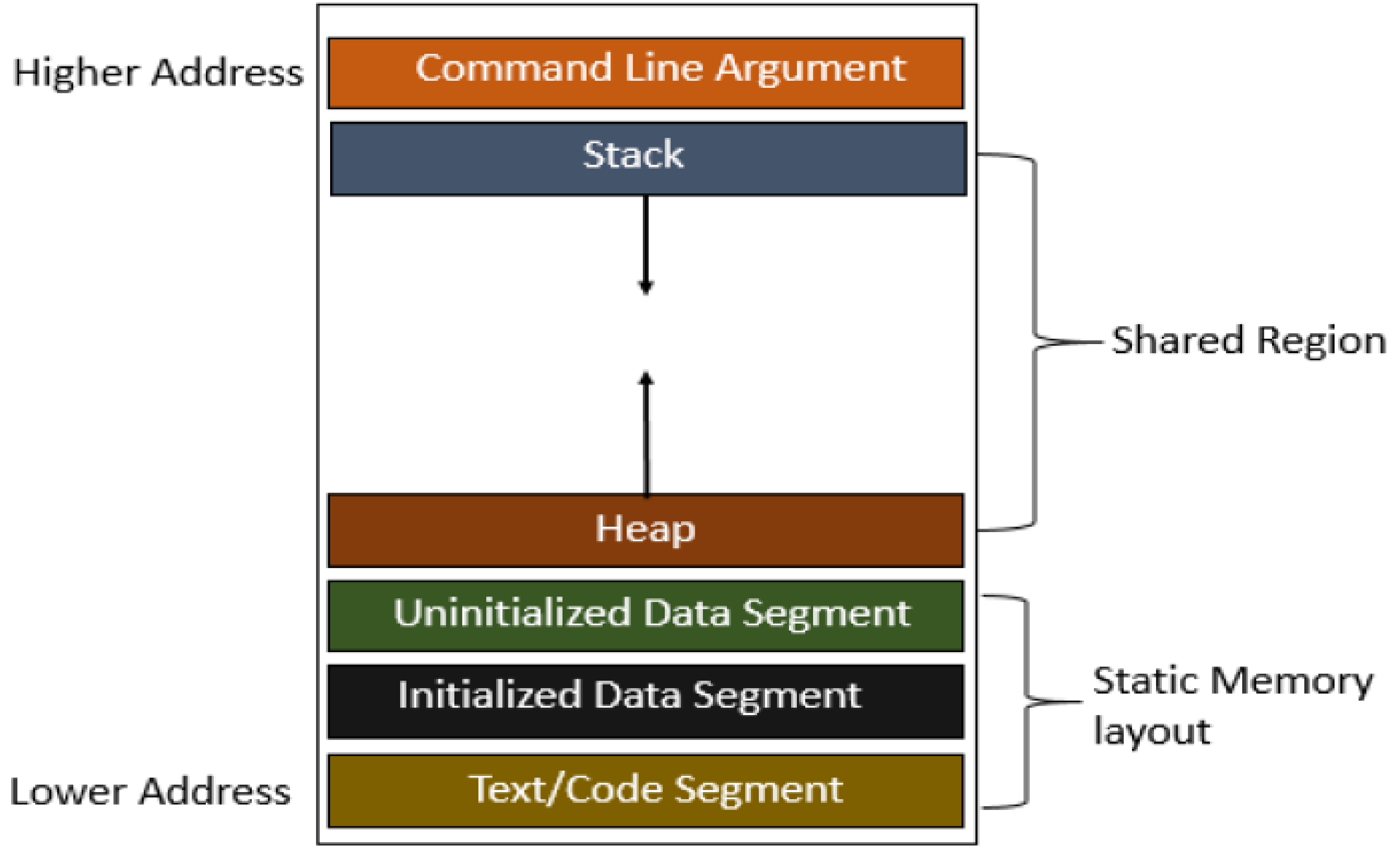
Continue..

- It can be seen from the diagram above that the maximum pending activation records at any moment are 3 (if the number of disks initially on the source peg is 3).
- If the disks are N, the figure would be N.
- Considering constant space for each activation record, the space complexity would be Θ(N).

➢ Life Cycle of Program



Process Address Space



Memory allocation for different segments through example (C Language)

```
int SquareOfSum(int x,int y);
int total;

int main()
{
        int a=4, b=2;
        total=squareofsum(a,b);
        printf("square of sum=%d\n", total);
        return 0;
}
```

Process Address Space	
Total, Count	Data Segment
	Heap Segment
AR of SquareOfSum() AR of main()	Stack Segment
Exe file	Code Segment
	Unmapped

Continue...

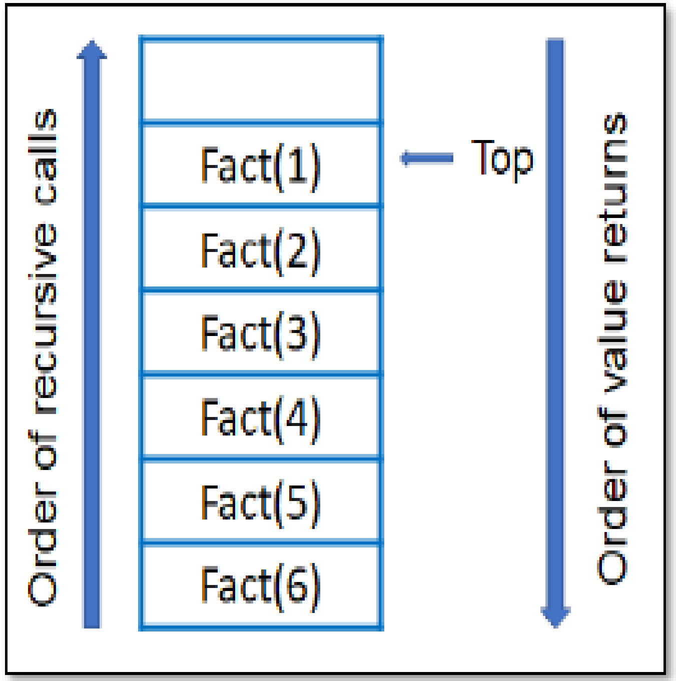
```
int squareofsum(int x,int y)
{
        int count=0;
        printf("fun called %d times",++count);
        return (x*y);
}
```

Activation Record	
Return Value	8
Return Address	1000
Local Variables	count
Actual argument	a & b
Formal argument	x & y

Complexity in Recursion:

1. Factorial

```
ALGORITHM Factorial(N)
Input: Any positive number N
Output: factorial of N
BEGIN:
        IF N==0 || N==1 THEN
                RETURN  1
        ELSE
                RETURN N*fact(N-1)
END;
```



Continue…

- **Time Complexity:** Total Number of function calls is n; hence total push and pop operations would be 2*n. Time complexity of this function would be Ɵ(n).
- **Space Complexity:** Space complexity will be equal to number of maximum pending activation records, n for this function i.e., Ɵ(n).

Can you find complexity of these problems:

- Computation of Power
- Fibonacci Series

Nested Recursion

- When a function calls itself with call of itself as a parameter, this is called nested recursion. See the function given below:

ALGORITHM A()

BEGIN:

A(A())

END;

- Function A() calls itself with the parameter in the function as A() itself.

Ackermann function

- Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a not primitive recursive.
- n+1 if m=0
- A(m, n)= A(m-1,1) if n=0
- A(m-1, A(m,n-1)) m>0, n>0
- **Example: Calculate A(1, 2)**

A(1,2) = A(0, A(1, 1))
= A(0, A(0, A(1, 0)))
= A(0, A(0, A(0,1)))
= A(0, A(0, 2))
= A(0, 3)
= 4

Indirect Recursion

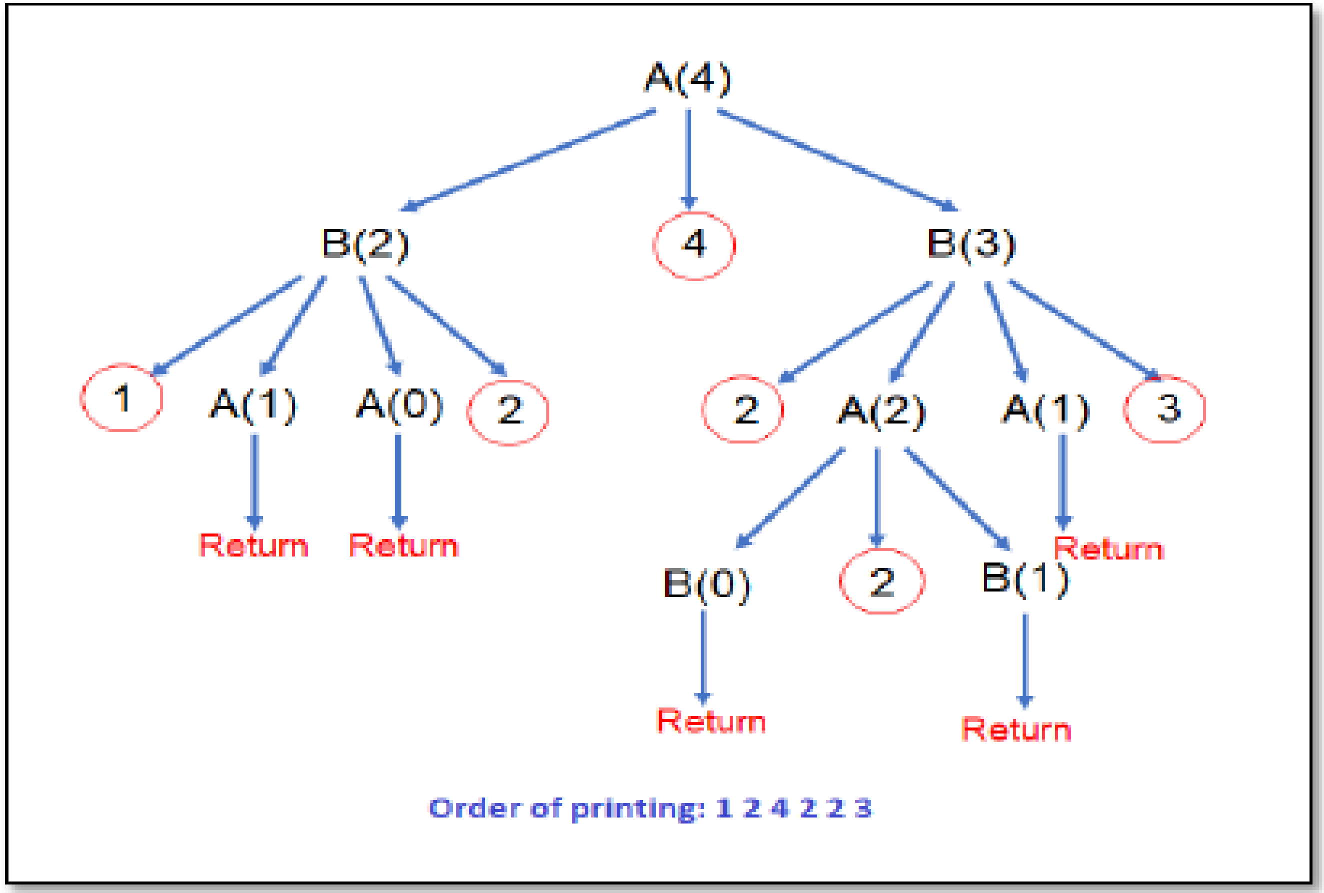
- When any function calls itself indirectly through another function, it is termed as indirect recursion. In the example given below, function A() indirectly is called from function B() and function B() is indirectly called from function A().

```
ALGORITHM A(n)
Input: Any positive number n
Output: input integer
BEGIN:
        IF n<=1 THEN
                RETURN
        ELSE
                B(n-2)
                WRITE("n")
                B(n-1)
END;
```

```
ALGORITHM B(n)
Input: Any positive number n
Output: input integer

BEGIN:
        IF n<=1 THEN
                RETURN
        ELSE
                WRITE(n-1)
                A(n-1)
                A(n-2)
                WRITE(n)
END;
```

Continue...


A(4)
B(2)
4
B(3)
1
A(1)
A(0)
2
2
A(2)
A(1)
3
Return
Return
B(0)
2
B(1)
Return
Return
Return
Order of printing: 1 2 4 2 2 3

- **Problem 1: Permutation Problem (Recursion within loop)**
- A permutation also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.
- Below are the permutations of string ABC.
- ABC ACB BAC BCA CBA CAB
- **ALGORITHM Permutation(A[ ], n, l)**

```
BEGIN:
        IF l == n-1 THEN
                display(A, n)
                RETURN
        FOR i=l TO n DO
                Swap(A[i], A[l])
                Permutation(A,n,l+1)
                Swap(A[i], A[l])
END;
```

Continue..

Problem 2: Given an array of integers, write a recursive code that add sum of all the previous numbers to each index of array.

Input: 1,2,3,4,5,6,7

Output: 1,3,6,10,15,21,28

```
ALGORITHM CumulativeSum(A[ ], N)
BEGIN:
        If N==1 THEN
                RETURN A[0]
        ELSE
                A[N-1] = CumulativeSum(A, N-1) + A[N-1]
                RETURN A[N-1]
END;
```

Exercise.....

```
Count(x,y) {
   if (y != 1){
      if (x != 1) {
         printf("*");
         Count(x/2, y);
      }
      else {
         y = y-1;
         Count(1024, y);
      }
   }
}
```

The number of times that the print statement is executed by the call Count(1024,1024) is _____.

A. 10240

B. 10250

C. 10230

D. 10220

Exercise.....

```
Count(x,y) {
   if (y != 1){
      if (x != 1) {
         printf("*");
         Count(x/2, y);
      }
      else {
         y = y-1;
         Count(1024, y);
      }
   }
}
```

The number of times that the print statement is executed by the call Count(1024,1024) is _____.

A. 10240

B. 10250

C. 10230

D. 10220

EXERCISE...

```c
#include<stdio.h>
int f(int n, int k)
{ if(n==0) return 0;
else if(n%2) return f(n/2, 2*k)+k;
else return f(n/2, 2*k)-k; }
int main()
{
printf("%d",f(20,1)); return 0;
}
```

What is the output printed by the following program?

A. 5

B. 8

C. 9

D. 20

EXERCISE..

```c
#include<stdio.h>
int f(int n, int k)
{ if(n==0) return 0;
else if(n%2) return f(n/2, 2*k)+k;
else return f(n/2, 2*k)-k; }
int main()
{
printf("%d",f(20,1)); return 0;
}
```

What is the output printed by the following program?

A. 5

B. 8

C. 9

D. 20

EXERCISE...

```
int f(int n)
 { static int r=0;
if(n<=0) return 1;
if(n>3)
{
r=n;
 return f(n-2)+2;
}
return f(n-1)+r;
}
```

What is the value of f(5)?

A. 5

B. 7

C. 9

D. 18

EXERCISE..

```
int f(int n)
 { static int r=0;
if(n<=0) return 1;
if(n>3)
{
r=n;
 return f(n-2)+2;
}
return f(n-1)+r;
}
```

What is the value of f(5)?

A. 5

B. 7

C. 9

D. 18

EXERCISE..

```c
void f(int n)
{
if(n<=1)
{
printf("%d",n);
}
else
{
f(n/2);
printf("%d",n%2);
}}
```

What does f(173) print?

A. 010110101

B. 010101101

C. 10110101

D. 10101101

EXERCISE..

```c
void f(int n)
{
if(n<=1)
{
printf("%d",n);
}
else
{
f(n/2);
printf("%d",n%2);
}}
```

What does f(173) print?

A. 010110101

B. 010101101

C. 10110101

D. 10101101

EXERCISE..

```
unsigned int foo(unsigned int n, unsigned int r)
{
if(n>0) return((n%r)+foo(n/r,r));
else return 0;
}
```

Consider the following recursive C function that takes two arguments. What is the return value of the function foo when it is called as foo(345,10)?

A. 345

B. 12

C. 5

D. 3

Summary

- Recursion and its Types
- Space and time complexity of recursion
- Nested Recursion
- Indirect Recursion
- Competitive questions.
- Exercise

Thank You