Data structures and Algorithms

Recursion

Course Objective Outline

- Understand the concept of Recursion
- Difference between Recursion Vs Iteration
- Implementation of recursion
- Give Recursion Examples used in data structures
- Analysis of recursion
- Memoization

Introduction

- Recursion is a very powerful programming tool. For a ‘C’ programmer it is always advantageous to use the recursion, for shortening the code.
- Recursion s actually a procedure of repeating some part of the code again and again for specific input.

- **Recursive Definition and Processes**
  - **Recursion is** a process of doing the same task again and again for specific input.
  - Many objects in mathematics are defined by presenting a process to produce that object.
  - For example if I wish to calculate the area of a square. If I write a function in which I can pass the parameters for the length and the breadth as a result I will get a square of some area. If I double the length and breadth I will get a big square.

Recursive programming

- Recursion is a programming technique in which a method calls itself, that is , a method is defined in terms of itself.
- Each call to a method creates a new environment in which all local variables and parameters are newly defined.
- Each time a method terminates, processing returns to the method that called it (which for recursive calls, is an earlier invocation of the same method).
- To ensure program termination, the method must define both :
  - A base case and
  - A recursive case
- A problem that can be solved in pieces is a good candidate for recursion.

Properties

- A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have:
  - **Base criteria/case** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
  - **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Subproblem

- The key is defining a subproblem.
- The recursive step MUST reduce the problem size (or you will end up with infinite recursion).
- You must be able to split the problem to make a recursive call.
- When the subproblem is “obvious”, you have the base case.
- Every problem must end with a base case.

Finding nth Term

Using Loop

```
int triangle(int n)
{
   int total = 0;
   while (n > 0)
   {
       total = total + n;
       --n;
    }
    return total;
}
```

Using Recursion

```
int triangle(int n)
{
   if(n == 1)
   {
      return 1;
    }
    else
       return(n + triangle(n-1));
}
```

Our observation (between loop vs. recursion)

- If a loop is used, the method cycles around the loop n times, adding n to the total the first time, n-1 the second time and so on, down to 1, quitting the loop when n becomes 0.

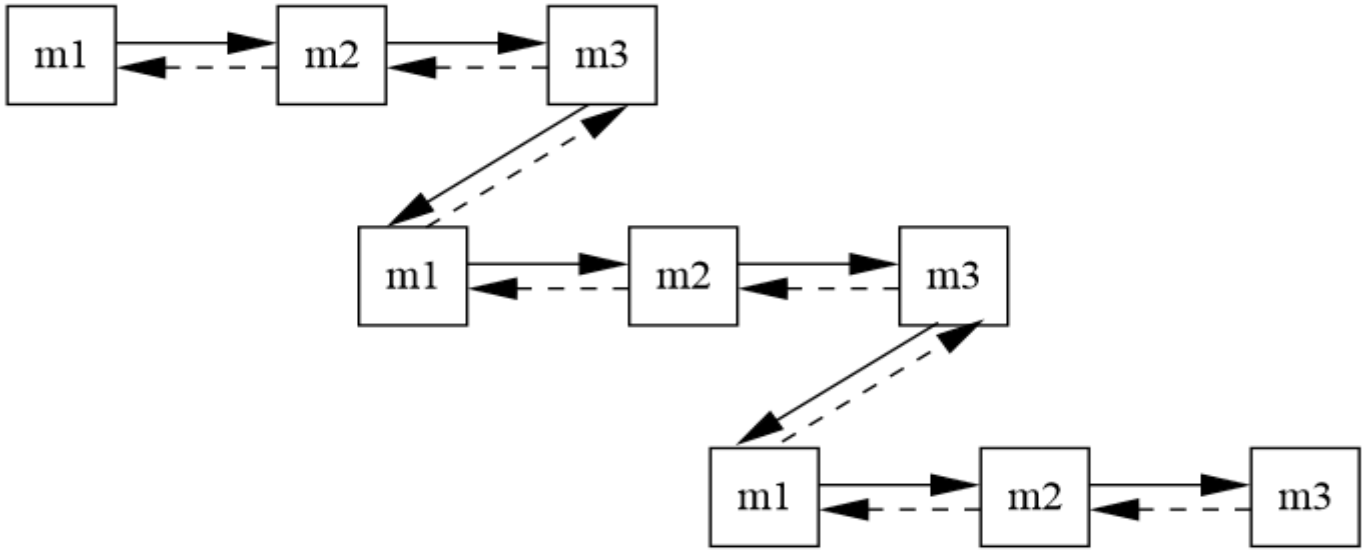
- If recursion is used, then a base case is used that determines when the recursion ends.

Characteristics of Recursive Methods

- The recursive method calls itself to solve a smaller problem.
- The base case is the smallest problem that the routine solves and the value is returned to the calling method. **(Terminal condition)**
- Calling a method involves certain overhead in transferring the control to the beginning of the method and in storing the information of the return point.
- Memory is used to store all the intermediate arguments and return values on the internal stack.
- The most important advantage is that it simplifies the problem conceptually.

Direct vs. Indirect Recursion

- Direct recursion occurs when a method invokes itself.
- Indirect recursion occurs when a method invokes another method, eventually resulting in the original method being invoked again



Recursive Algorithm

- Recursive Algorithm
  - Algorithm that uses itself as part of the solution
- Recursive Call
  - A method call in which the method being called is the same as the one making the call
- Direct Recursion
  - Recursion in which a method directly calls itself
- Indirect Recursion
  - Recursion in which a chain of two or more method calls returns to the method that originated the chain, e.g. A calls B, B calls C, and C calls A

Mathematical Induction

- Recursion is programming equivalent of mathematical induction, which is a way of defining something in terms of itself.
- It is valid provided there is a base case.

Recursive Sum of Numbers

- Consider the sum of numbers from 1 to 10 :
- What is the base case?
- What is the recursive case?

```
public int sum (int num) {
int result;
if (num == 1)
    result = 1;
else
    result = num + sum (num);
return result;
}
```

Recursion vs. Iteration

- The non-recursive solution to the summation problem compute the numbers between 1 and num in an iterative manner.

```
sum = 0;
for (int i = 1; i <= num; i++)
sum += i;
```

- This solution is more straight forward. The recursive solution however, demonstrates the concept of recursion using a simple problem.
- In many problems (though not all!), recursion is the shorter and more elegant solution.

Iterative Algorithm

- In general, an iterative algorithm is a non-recursive one
- A typical iterative algorithm contains one or several, possibly nested loops (repetitions of instructions):
  - For ... Do
  - While ... Do
  - Repeat ... Until
- Iteration also stands for the style of programming used in imperative programming languages (e.g. Assembler, Basic, Pascal, C++, Java, . . .)
- This contrasts with recursion, which is more declarative
  - Functional programming languages (e.g. Scheme, Haskell, . . .)
  - Logical programming languages (e.g. Prolog)

Factorials

- Factorials are similar to triangle, addition being replaced by multiplication and the base case is when input is 0.
- The factorial of n is found by multiplying n with the factorial of n-1.
- E.g. factorial of 3 = 3 * 2 * 1.
- Factorial of 0 is defined as 1.

Iterative definition

- In general, we can define the factorial function in the following way:

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120 \end{aligned}$$

Recursive Definition

- We can also define the factorial function in the following way:

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

Difference between Iterative and Recursive

- Iterative

Function does NOT calls itself

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

- Recursive

Function calls itself

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

Iterative algorithm

```
factorial(n) {
    i = 1
    factN = 1
    loop (i <= n)
        factN = factN * i
        i = i + 1
    end loop
    return factN
}
```

The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.

Recursive Algorithm

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

Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version ➔

we have eliminated the loop and implemented the algorithm with 1 'if' statement.

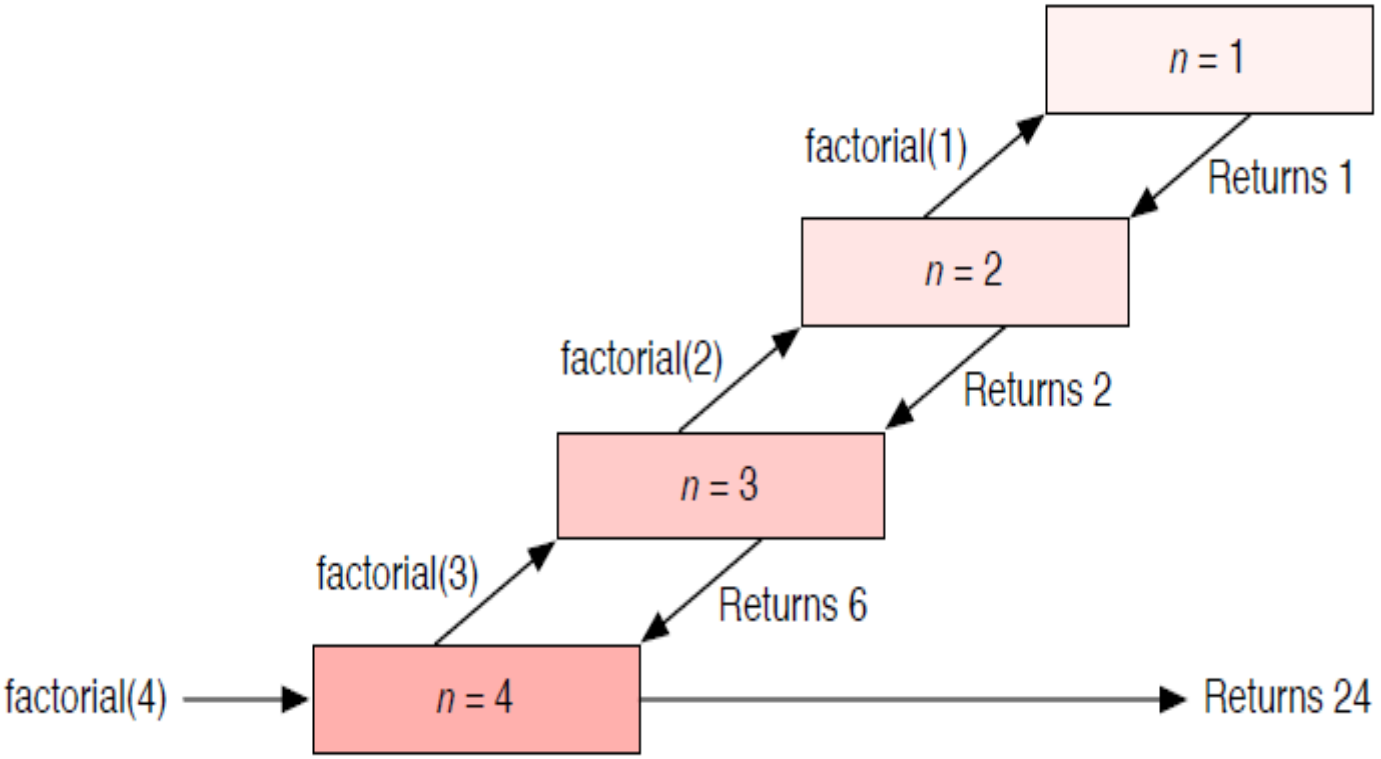
Example 1: Factorial (Recursion)

- Compute the factorial of integer n:
  - $n! = 1 * 2 * \cdots * n = \begin{cases} 1 & n=1 \\ n*(n-1)! & n>1 \end{cases}$

```
algorithm factorial(n)
  if n = 1 then
    return 1 // base case of the recursion
  else
    return n*factorial(n − 1) // recursive call
```

Example 1: Factorial (Recursion)

- The execution of a recursive algorithm uses a call stack to keep track of previous method call.



Recursion Vs Iteration

- Some things (e.g. reading from a file) are easier to implement iteratively.
- Other things (e.g. merge sort, quick sort) are easier to implement recursively.
- Others are just as easy both ways.
- When there is no real benefit to the programmer to choose recursion, iteration is the more efficient choice.
- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version are equivalent.

Fibonacci Numbers(Recursion)

```
def fib(n):
    If  n==0:
      return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Fibonacci Numbers(Iteration)

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a,b = b, a+b
    return a
```

Fibonacci Numbers(Iteration)

```
If (n<=1)
      return (n);
       a=0;
       b=1;
      for (i=2;i<=n; i++)
       {
          x=a;
          a=b;
          b=x+a;
       }

     return (b);
```

Recursive Binary Search

- Binary search can also be implemented using recursion.
- The method can call itself with new starting and ending values.
- The base case is when the starting value is greater than the end value.

Algorithm for Binary search using Recursive Definition

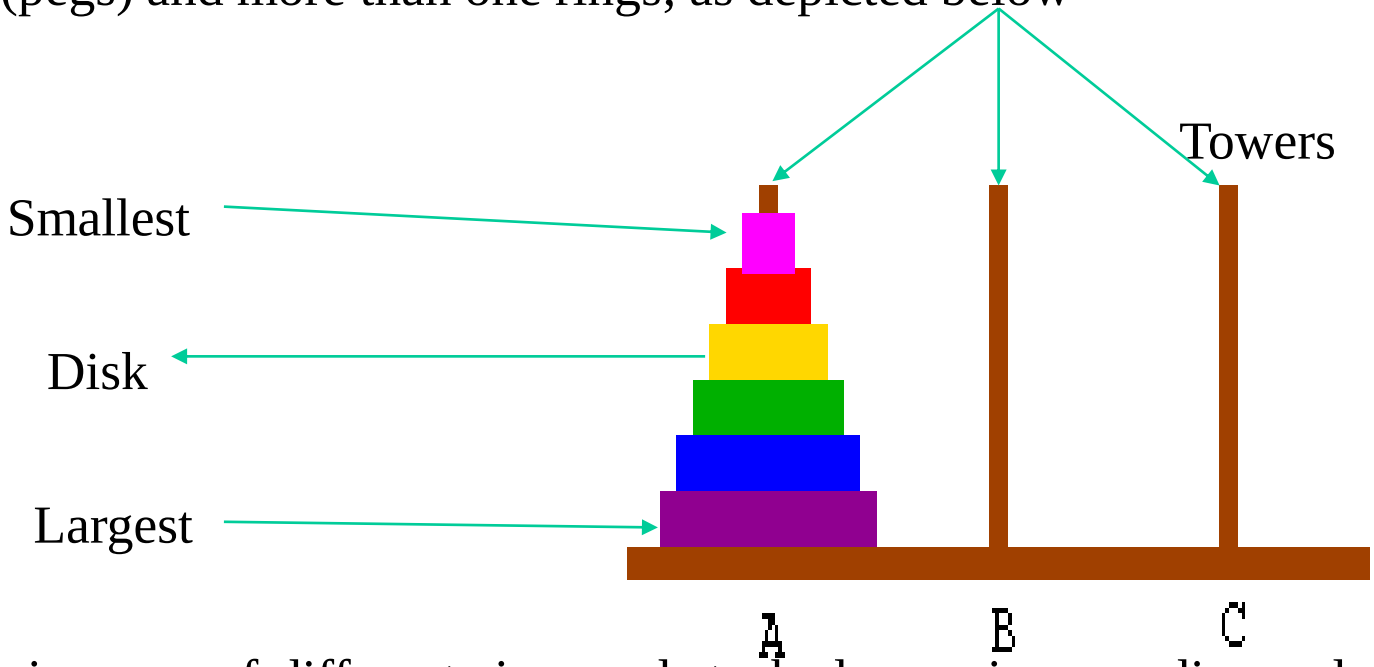
1. if(low>high)
2. return;
3. mid=(low+high)/2;
4. if(x==a[mid]);
5. return (mid);
6. if (x<a[mid])
7. search for x in a[low] to a[mid-1];
8. else
9. search for x in a[mid+1] to a[high];

Divide-and-Conquer

- Recursive binary search is an example of divide – and – conquer.
- The idea is to divide the bigger problem into smaller problems and solve each one separately.
- The solution to each smaller problem is to divide it into even smaller problems and solve them.
- The process continues till the base case is reached.
- It can be used with recursion as well as non recursion.
- Benefits of divide and conquer
  - Powerful tool for solving conceptually difficult problems
  - Often provides a natural way to design efficient algorithms
  - Sub-problems be executed in parallel
- Decrease and conquer is a variation of divide and conquer, where the problem is simplified into a single sub-problem

Tower of Hanoi

- Tower of Hanoi, is a mathematical puzzle which consists of three tower (pegs) and more than one rings; as depicted below −



- These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one.
- There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

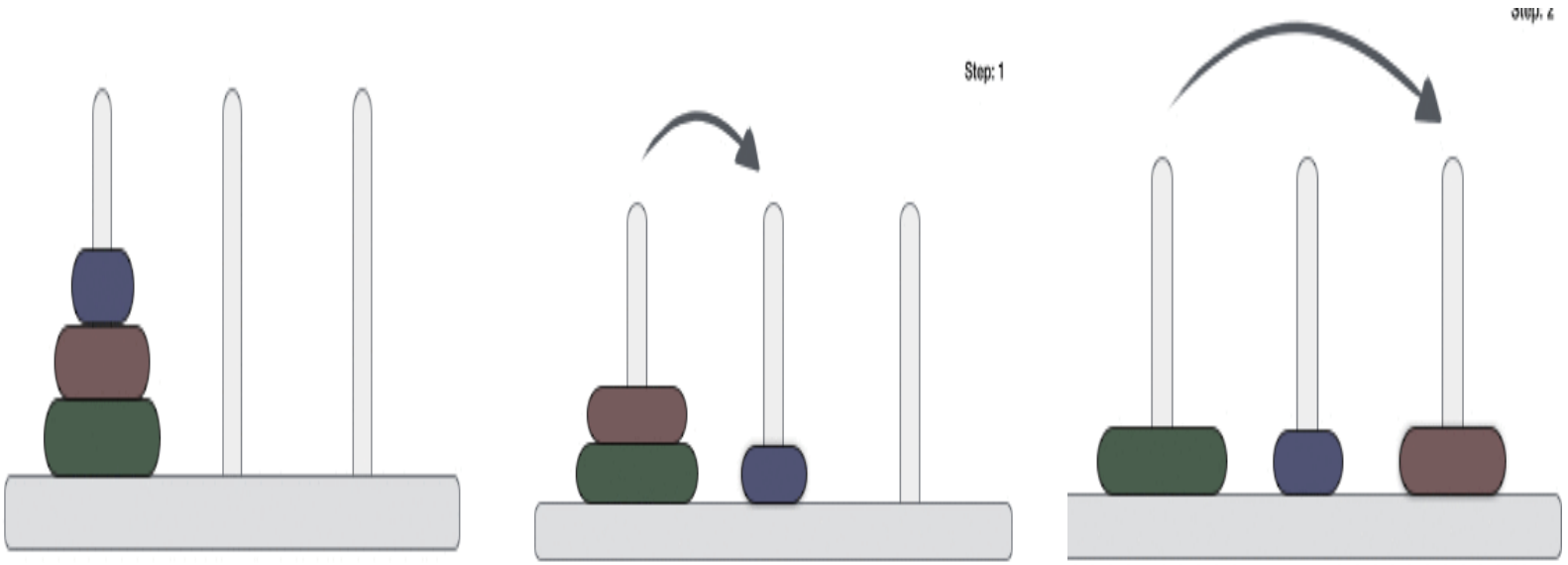
Tower of Hanoi :Rules

- The mission is to move all the disks to some other tower without violating the sequence of arrangement. Below mentioned are few rules which are to be followed for tower of Hanoi −
  - Only one disk can be moved among the towers at any given time.
  - Only the "top" disk can be removed.
  - No large disk can sit over a small disk.

Tower of Hanoi :Example

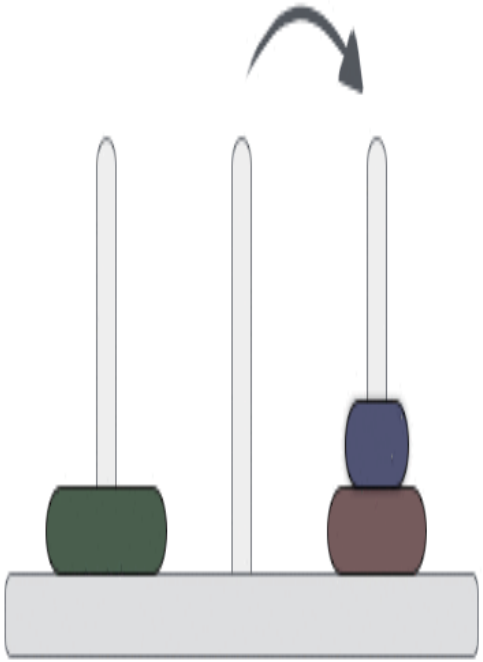
- Tower of hanoi puzzle with n disks can be solved in minimum 2^n-1 steps. This presentation shows that a puzzle with 3 disks has taken $2^3-1 = 7$ steps.

Step 0 Step 1 Step 2

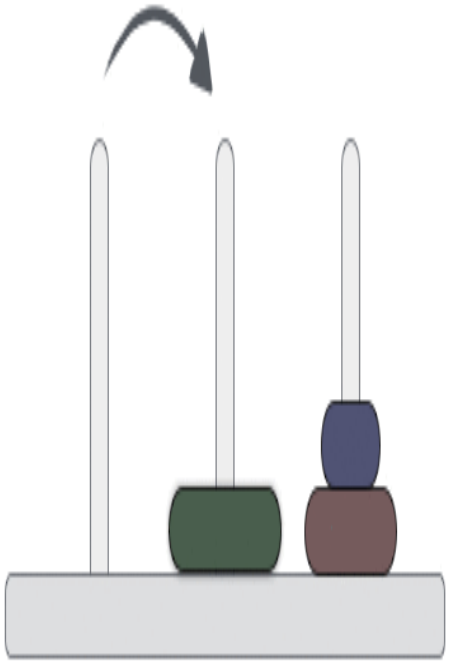


Tower of Hanoi : Example- Continued

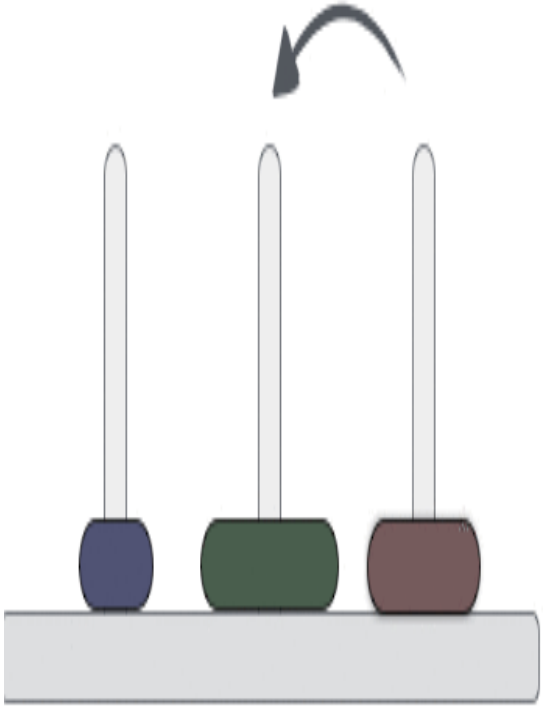
Step 3



Step 4

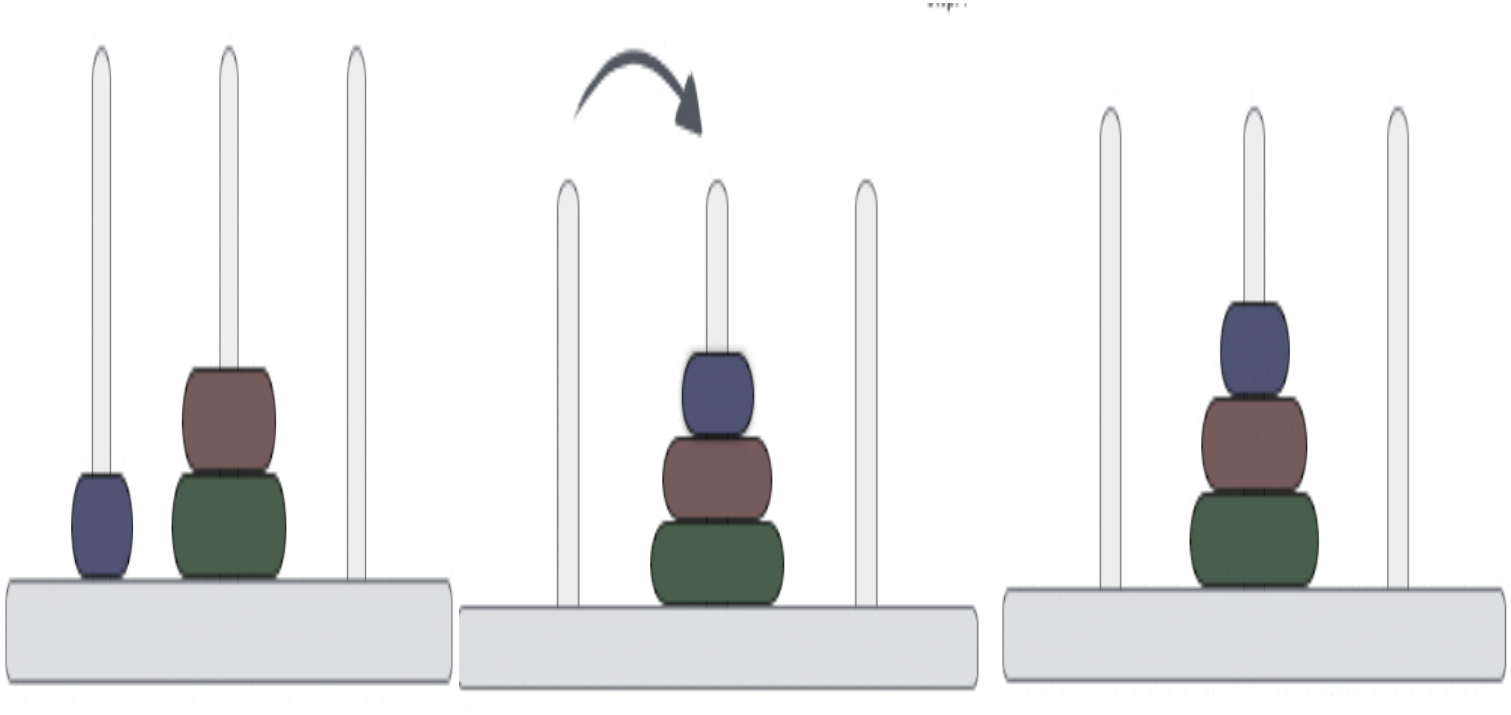


Step 5



Tower of Hanoi : Example- Continued

Step 6 Step 7 Done



Tower of Hanoi : Algorithm

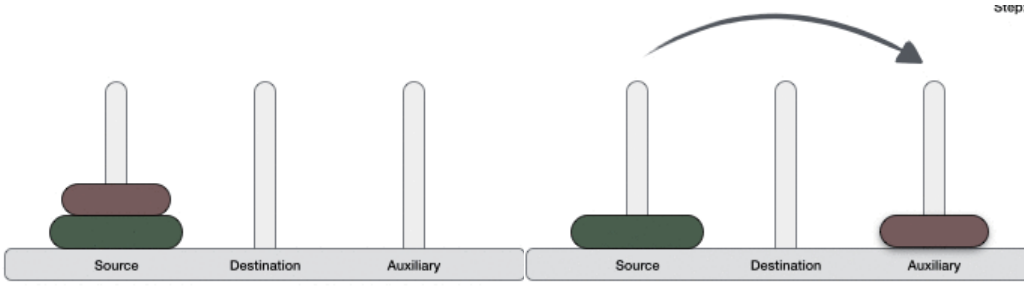
- To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2.
- We mark three towers with name, source, destination and aux (only to help moving disks).
- If we have only one disk, then it can easily be moved from source to destination peg.
- If we have 2 disks −
  - First we move the smaller one (top) disk to aux peg
  - Then we move the larger one (bottom) disk to destination peg
  - And finally, we move the smaller one from aux to destination peg.

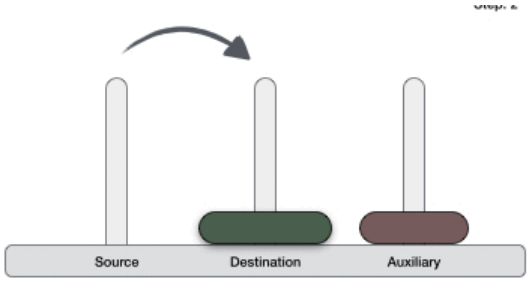
Tower of Hanoi : Algorithm

- Originally

Step 1

Step 2

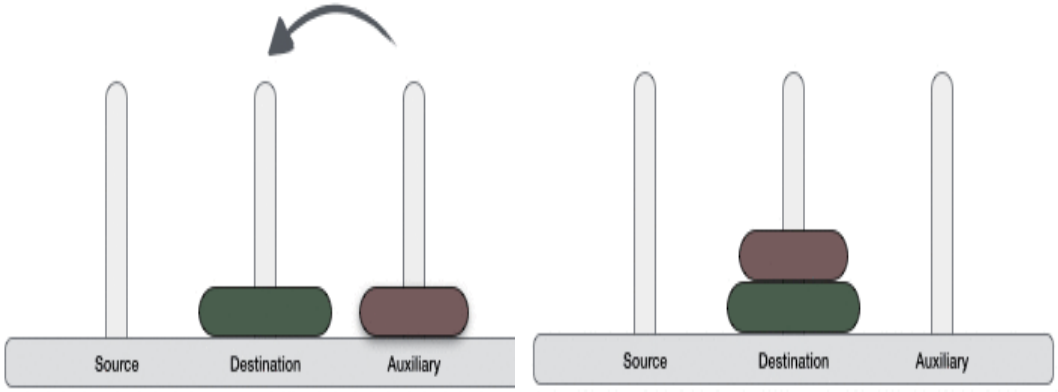




Tower of Hanoi : Algorithm

- Step 3

  Done



Tower of Hanoi : Pseudocode

```
FUNCTION MoveTower(disk, source, dest, spare):
IF disk == 1, THEN:
    move disk from source to dest
ELSE:
    MoveTower(disk - 1, source, spare, dest)   // Step 1 above
    move disk from source to dest              // Step 2 above
    MoveTower(disk - 1, spare, dest, source)   // Step 3 above
END IF
```

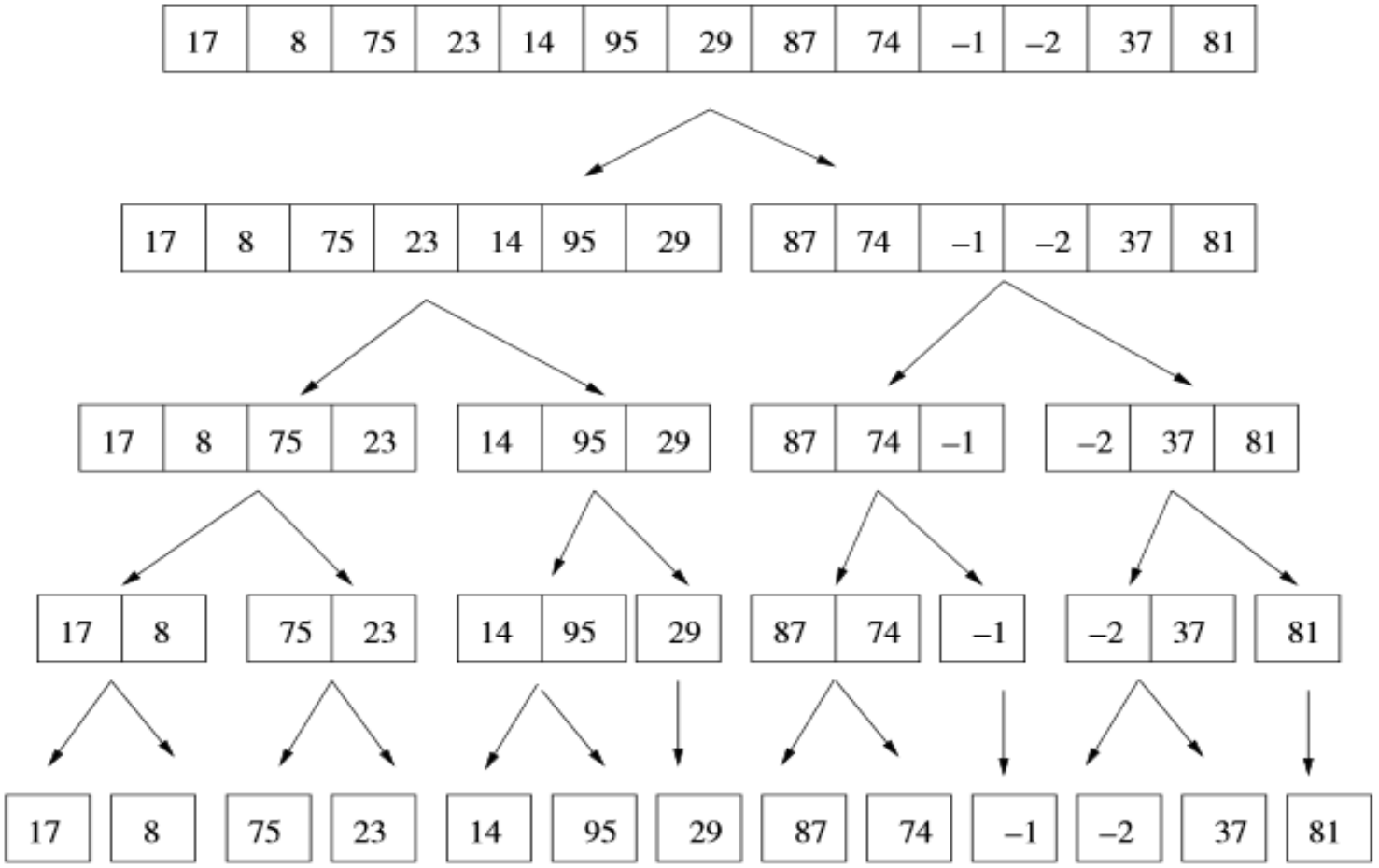
Merge Sort

- Now that we've seen recursion, we can look at another sorting algorithm called merge sort, which is more efficient than insertion and selection sort.
- It is usually defined recursively, where the recursive calls work on smaller and smaller parts of the list.
- Idea:
  - Split the unsorted list into two sub-list of about half the size
  - Recursively sort each half
  - "Merge" the two sorted halves into a single sorted list

Merge Sort Example

- Based on a divide and conquer strategy:
  - list is divided into two halves (divide)
  - each half is sorted independently (conquer).
  - two halves are merged into a sorted sequence
- Split original list recursively until base case is reached

Merge Sort Example



Efficiency

- Mergesort runs in O(N*logN) time.
- Assuming that the number of items is a power of 2, for each individual merging operation, the maximum number of comparisons is always less than the number of items being merged, minimum is half the number of items being merged.

Eliminating Recursion

- Some algorithms use recursive methods, some don’t.
- Often an algorithm is easy to conceptualize as a recursive method, but in practice the recursive method might prove to be inefficient.
- In such cases, it is useful to transform the recursive approach into a non-recursive approach.
- Such transformation makes use of stack.

Recursion and Stacks

- Most compilers implement recursion using stacks.
- When a method is called the compiler pushes the arguments to the method and the return address on the stack and then transfers the control to the method
- When the method returns, it pops these values off the stack
- The arguments disappear and the control returns to the return address.

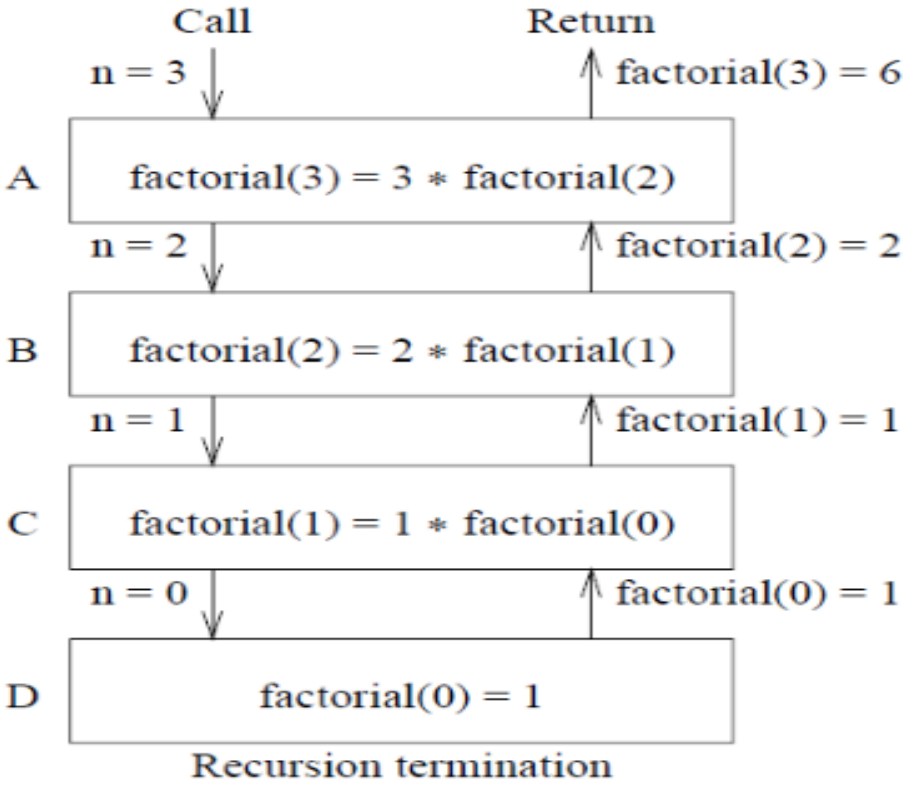
How Recursion Works

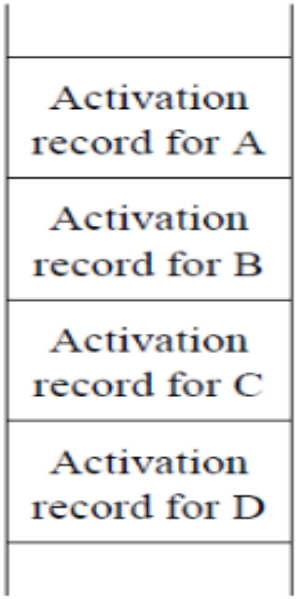
- To truly understand how recursion works we need to first explore how any function call works.
- When a program calls a subroutine (function) the current function must suspend its processing.
- The called function then takes over control of the program.
- When the function is finished, it needs to return to the function that called it.
- The calling function then 'wakes up' and continues processing.
- One important point in this interaction is that, unless changed through call-by- reference, all local data in the calling module must remain unchanged.
- Therefore, when a function is called, some information needs to be saved in order to return the calling module back to its original state (i.e., the state it was in before the call).
- We need to save information such as the local variables and the spot in the code to return to after the called function is finished.

How Recursion Works

- To do this we use a stack.
- Before a function is called, all relevant data is stored in a ***stackframe***.
- This stackframe is then pushed onto the system stack.
- After the called function is finished, it simply pops the system stack to return to the original state.
- By using a stack, we can have functions call other functions which can call other functions, etc.
- Because the stack is a first-in, last-out data structure, as the stackframes are popped, the data comes out in the correct order.

How recursion works





- As you can see from the figure, each call to the factorial creates an activation record until the base case is reached and starting from there we accumulate the result in the form of product.

Advantages of recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Analysis of recursion

- One may argue that why to use recursion as the same task can be done with iteration.
  - The first reason is recursion makes a program more readable and because of today's enhanced CPU systems, recursion is more efficient than iterations.
  - **Time complexity**
    - In case of iterations, we take number of iterations to count the time complexity.
    - Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of time recursive call is being made. A call made to a function is O(1), hence the (n) number of time a recursive call is made makes the recursive function O(n).

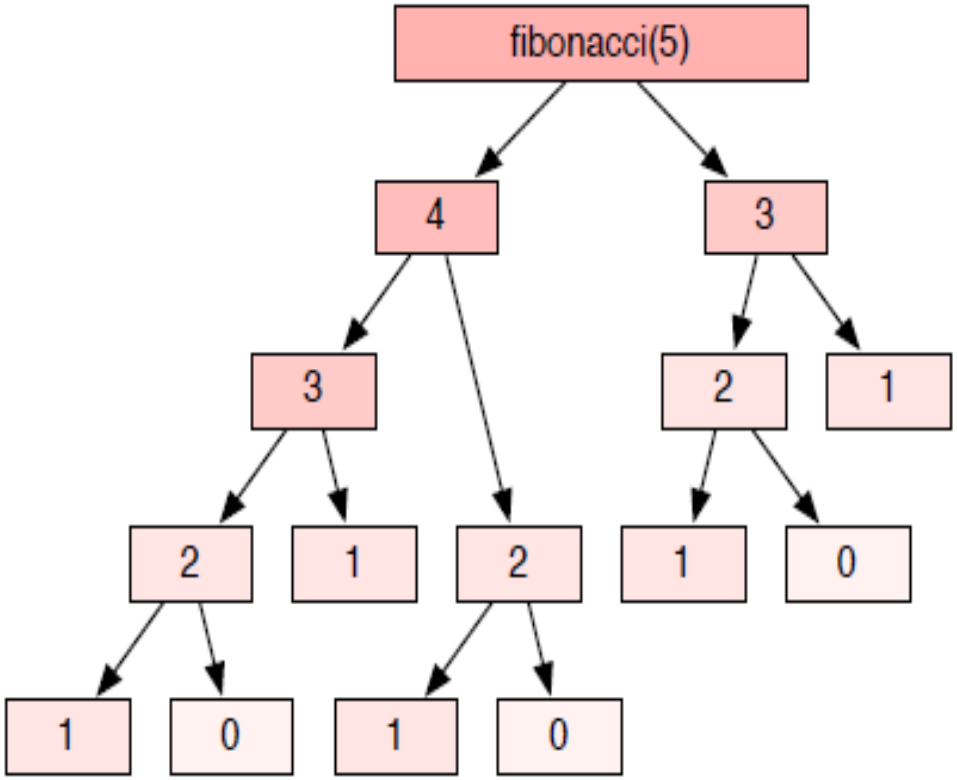
Analysis of recursion

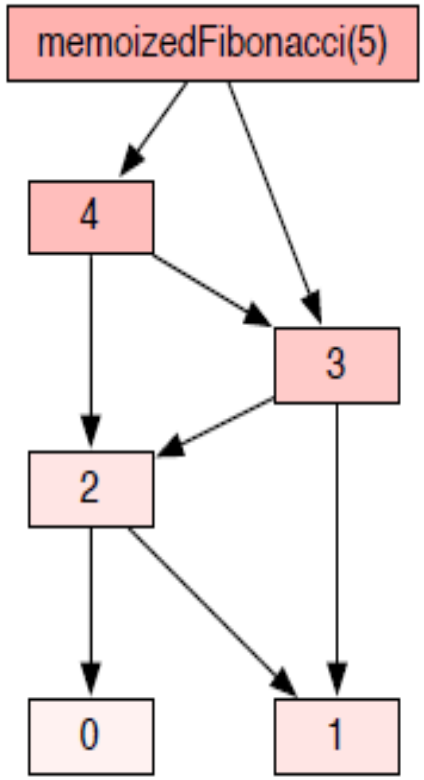
- **Space complexity**
  - Space complexity is counted as what amount of extra space is required for a module to execute.
  - In case of iterations, the compiler hardly requires any extra space. Compiler keeps updating the values of variables used in the iterations.
  - But in case of recursion, the system needs to store activation record each time a recursive call is made. So it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Memoization

- A common technique to speed up recursive algorithms is called memoization
  - A memoized function stores (caches) results of previous calls.
  - When the function is later called with the same parameters, it returns the stored result rather than recalculating it.
  - Memoized functions are not allowed to have side-effects.
    - No screen output
    - No global variables assignments
    - No change of object attributes
  - Memoization is usually implemented with hash tables

Memoization- Example





$\Rightarrow$ Complexity is reduced from $O(2^n)$ to $O(n)$

Question

- Write a recursive function f(n)= 4 * n, i.e. that prints out multiples of 4 for n<=50.

Comparison of Iteration and Recursion

Iteration

- Iteration is a process of executing certain set of instructions repeatedly, without calling the self function
- The iterative functions are implemented with the help of for, while, do-while programming constructs.
- The iterative methods are more efficient because of better execution speed

Recursion

- Recursion is a process of executing certain set of instructions repeatedly by calling the self function repeatedly.
- Instead of making use of for, while, do-while the repetition in code execution is obtained by calling the same function again and again over some condition
- The recursive methods are less efficient

Comparison between Iteration and Recursion

Iteration

- Memory utilization by iteration is less.
- It is simple to implement.
- The lines of code is more when we use iteration

Recursion

- Memory utilization is more in recursive functions.
- Recursive methods are complex to implement
- Recursive methods bring compactness in the program