CHAPTER 2

RECURSION AND BACKTRACKING

2.1 Introduction

In this chapter, we will look at one of the important topics, *“recursion”*, which will be used in almost every chapter, and also its relative *“backtracking”*.

2.2 What is Recursion?

Any function which calls itself is called *recursive*. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls.

It is important to ensure that the recursion terminates. Each time the function calls itself with a slightly simpler version of the original problem. The sequence of smaller problems must eventually converge on the base case.

2.3 Why Recursion?

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

2.4 Format of a Recursive Function

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the *base case*. The former, where the function calls itself to perform a subtask, is referred to as the *ecursive case*. We can write all recursive functions using the format:

```
if(test for the base case)
   return some base case value
else if(test for another base case)
   return some other base case value
// the recursive case
else
   return (some work and then a recursive call)
```

As an example consider the factorial function: $n!$ is the product of all integers between n and 1. The definition of recursive factorial looks like:

$$n! = 1, \qquad \text{if } n = 0$$
$$n! = n * (n-1)! \text{ if } n > 0$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of $n!$, and the subproblem is determining the value of $(n - l)!$. In the recursive case, when n is greater than 1, the function calls itself to determine the value of $(n - l)!$ and multiplies that with n.

In the base case, when n is 0 or 1, the function simply returns 1. This looks like the following:

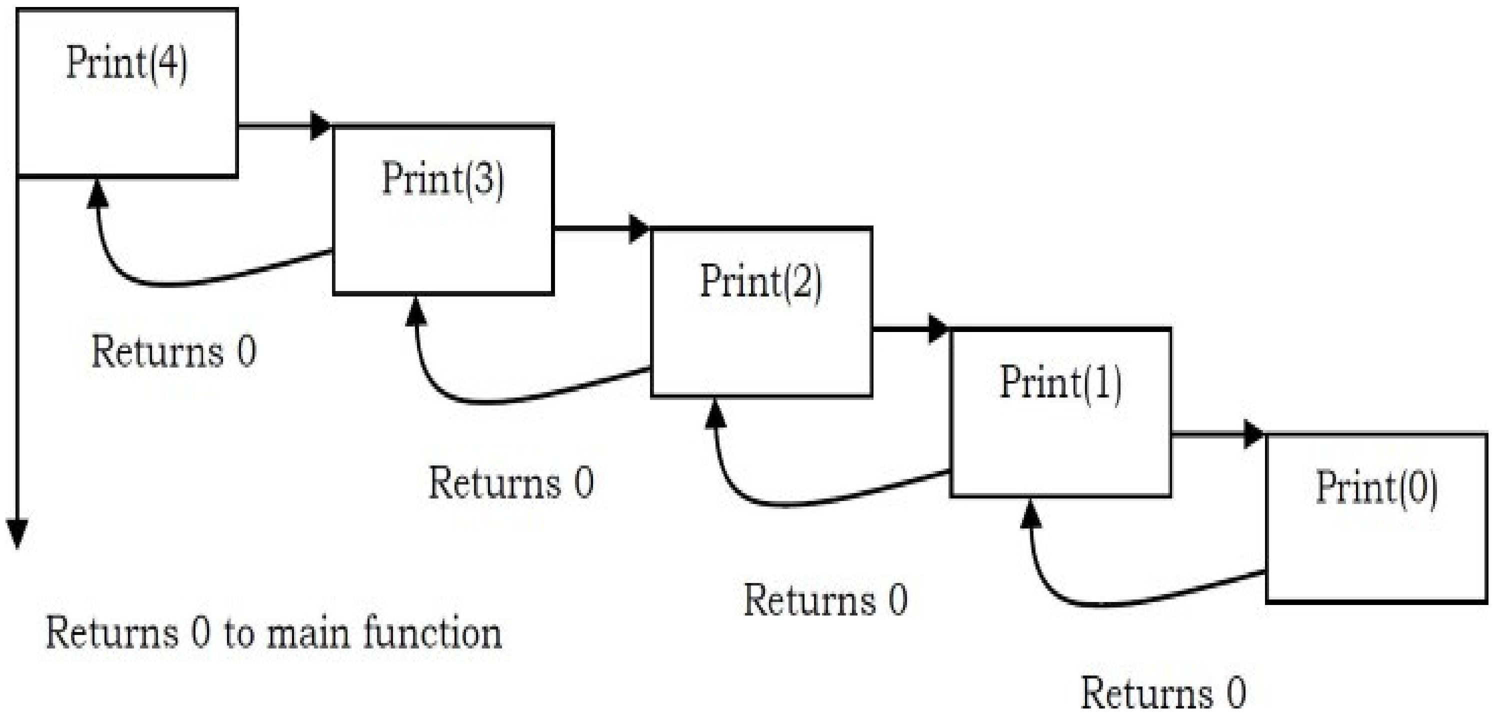
```
// calculates factorial of a positive integer
int Fact(int n) {
   if(n == 1)    // base cases: fact of 0 or 1 is 1
      return 1;
   else if(n == 0)
      return 1;
   else     // recursive case: multiply n by (n – 1) factorial
      return n*Fact(n-1);
}
```

2.5 Recursion and Memory (Visualization)

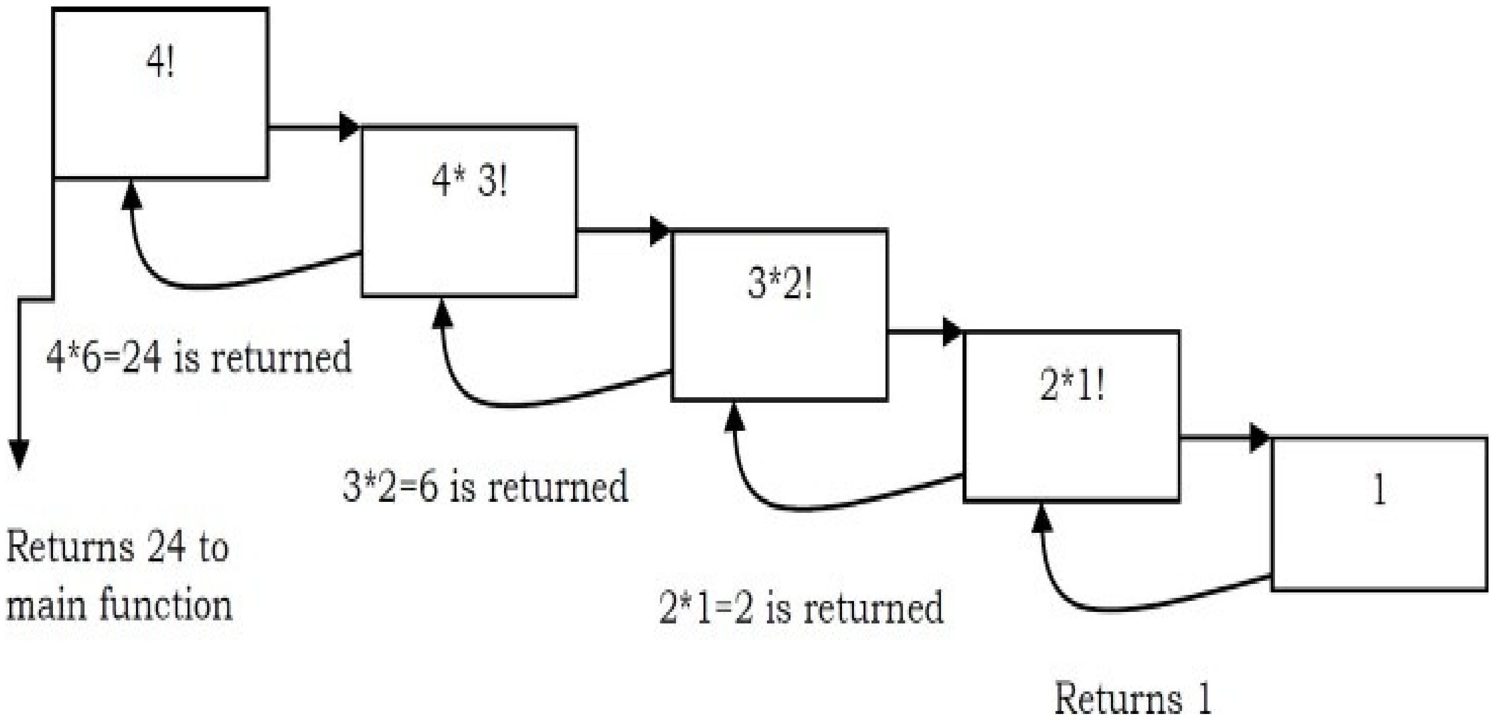
Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding, let us consider the following example.

```
//print numbers 1 to n backward
int Print(int n) {
 if( n = = 0) // this is the terminating base case
     return 0;
 else {
     printf (“%d”,n);
     return Print(n-1);        // recursive call to itself again
 }
}
```

For this example, if we call the print function with n=4, visually our memory assignments may look like:



Now, let us consider our factorial function. The visualization of factorial function with n=4 will look like:



2.6 Recursion versus Iteration

While discussing recursion, the basic question that comes to mind is: which way is better? – iteration or recursion? The answer to this question depends on what we are trying to do. A recursive approach mirrors the problem that we are trying to solve. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers. But, recursion adds overhead for each recursive call (*n*eeds space on the stack frame).

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

2.7 Notes on Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

2.8 Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi

- Backtracking Algorithms [we will discuss in next section]

2.9 Recursion: Problems & Solutions

In this chapter we cover a few problems with recursion and we will discuss the rest in other chapters. By the time you complete reading the entire book, you will encounter many recursion problems.

Problem-1 Discuss Towers of Hanoi puzzle.

Solution: The Towers of Hanoi is a mathematical puzzle. It consists of three rods (or pegs or towers), and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks on one rod in ascending order of size, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, satisfying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Algorithm:

- Move the top $n - 1$ disks from *Source* to *Auxiliary* tower,
- Move the n^{th} disk from *Source* to *Destination* tower,
- Move the $n - 1$ disks from *Auxiliary* tower to *Destination* tower.
- Transferring the top $n - 1$ disks from *Source* to *Auxiliary* tower can again be thought of as a fresh problem and can be solved in the same manner. Once we solve *Towers of Hanoi* with three disks, we can solve it with any number of disks with the above algorithm.

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

    /* Move remaining disks from A to C */
    printf("\nMove disk %d from peg %c to peg %c", n, frompeg, topeg);

    /* Move n-1 disks from B to C using A as auxiliary */
    TowersOfHanoi(n-1, auxpeg, topeg, frompeg);
}
```

Problem-2 Given an array, check whether the array is in sorted order with recursion.

Solution:

```
int isArrayInSortedOrder(int A[],int n){
    if(n == 1)
      return 1;
    return (A[n-1] < A[n-2])?0:isArrayInSortedOrder(A,n-1);
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*) for recursive stack space.

2.10 What is Backtracking?

Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options. In backtracking, we start with one possible option out of many available options and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other option and try to solve it. If none if the options work out we will claim that there is no solution for the problem.

Backtracking is a form of recursion. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of