Chapter 5 Recursion

1. Principle of Recursion

The recursion is a process by which a function calls itself. We use recursion to solve complex problem if the complex problem can be divided into smaller sub-problems and have the same solution process. One thing we have to keep in mind, that if each sub-problem is following same kind of patterns, then only we can use the recursive approach.

A recursive function has two different parts. The base case and the recursive case. The base case is used to terminate the task of recurring. If base case is not defined, then the function will recur infinite number of times.

In computer program, when we call one function, the value of the program counter is stored into the internal stack before jumping into the function area. After completing the task, it pops out the address and assign it into the program counter, then resume the task. During recursive call, it will store the address multiple times, and jumps into the next function call statement. If one base case is not defined, it will recur again and again, and store address into stack. If the stack has no space anymore, it will raise an error as “Internal Stack Overflow”.

One example of recursive call is finding the factorial of a number. We can see that the factorial of a number n = n! is same as the n * (n-1)!, again it is same as n * (n - 1) * (n - 2)!. So if the factorial is a function, then it will be called again and again, but the argument is decreased by 1. When the argument is 1 or 0, it will return 1. This could be the base case of the recursion.

Example

```
#include<iostream>
using namespace std;
long fact(long n){
   if(n <= 1)
   return 1;
   return n * fact(n-1);
}
main(){
   cout << "Factorial of 6: " << fact(6);
}
```

Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Base condition in recursion

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
```

```
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Solving problem using recursion

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Stack Overflow error in recursion

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Disadvantages of recursive over iterative function

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

Advantages of recursive over iterative function

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi. Most importantly, recursive function are simpler to solve complex problems. Therefore, it is better idea to use recursive function to solve complex problem because to write the iterative function for some complex function yield in very complex function.

2. Recursion and Iteration

Both recursion and iteration are used for executing some instructions repeatedly until some condition is true. A same problem can be solved with recursion as well as iteration but still there are several differences in their working and performance.

Below are the detailed example to illustrate the difference between the two:

Time Complexity: Finding the Time complexity of Recursion is more difficult than that of Iteration.

- **Recursion**: Time complexity of recursion can be found by finding the value of the nth recursive call in terms of the previous calls. Thus, finding the destination case in terms of the base case, and solving in terms of the base case gives us an idea of the time complexity of recursive equations.
- **Iteration**: Time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

Usage: Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.

- **Recursion**: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.
- **Iteration**: Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

Overhead: Recursion has a large amount of Overhead as compared to Iteration.

- **Recursion**: Recursion has the overhead of repeated function calls, that is due to repetitive calling of the same function, the time complexity of the code increases manifold.
- **Iteration**: Iteration does not involve any such overhead.

Infinite Repetition: Infinite Repetition in recursion can lead to CPU crash but in iteration, it will stop when memory is exhausted.

- **Recursion**: In Recursion, Infinite recursive calls may occur due to some mistake in specifying the base condition, which on never becoming false, keeps calling the function, which may lead to system CPU crash.
- **Iteration**: Infinite iteration due to mistake in iterator assignment or increment, or in the terminating condition, will lead to infinite loops, which may or may not lead to system errors, but will surely stop program execution any further.

Difference between Recursion and Iteration

	Recursion	Iteration
Definition	Recursion refers to a situation where a function calls itself again and again until some base condition is not reached.	Iteration refers to a situation where some statements are executed again and again using loops until some condition is true.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, and execution of statement within loop and update (increments and decrements) the control variable.
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.
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Doesn't use stack

Performance	It is comparatively slower because before each function call the current state of function is stored in stack. After the return statement the previous function state is again restored from stack.	Its execution is faster because it doesn't use stack.
Memory	Memory usage is more as stack is used to store the current function state.	Memory usage is less as it doesn't use stack.
Code Size	Size of code is comparatively smaller in recursion.	Iteration makes the code size bigger.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.

Lets write the implementation of finding factorial of number using recursion as well as iteration.

Recursion Example

Below is the C program to find factorial of a number using recursion.

```
#include <stdio.h>
int factorial(int n){
    if(n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
int main() {
    printf("Factorial for 5 is %d", factorial(5));
    return 0;
}
```

Iteration Example

Below is the C program to find factorial of a number using iteration.

```
#include <stdio.h>

int main() {
    int i, n = 5, fac = 1;
```

```
    for(i = 1; i <= n; ++i)
        fac = fac * i;

    printf("Factorial for 5 is %d", fac);

    return 0;
}
```

3. Factorial and Fibonacci sequence

The Fibonacci sequence is a sequence where the next term is the sum of the previous two terms. The first two terms of the Fibonacci sequence are 0 followed by 1.

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21…

Except for the first two terms of the sequence, every other term is the sum of the previous two terms.

Fibonacci series program in C using iteration

```c
#include <stdio.h>
int main()
{
  int n, first = 0, second = 1, next, c;
  printf("Enter the number of terms\n");
  scanf("%d", &n);
  printf("First %d terms of Fibonacci series are:\n", n);
  for (c = 0; c < n; c++)
  {
    if (c <= 1)
      next = c;
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d\n", next);
  }
  return 0;
}
```

Fibonacci series C program using recursion

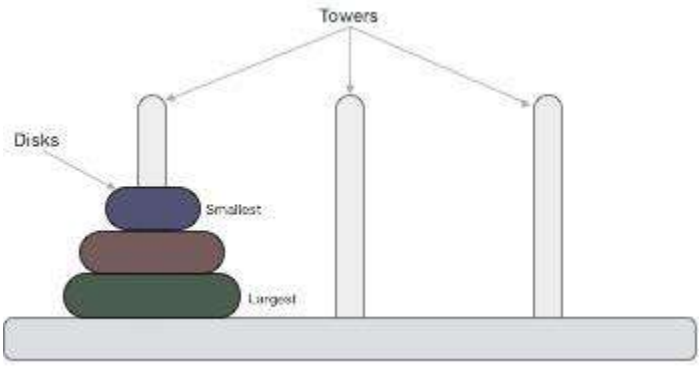
```c
#include<stdio.h>
int f(int);
int main()
{
  int n, i = 0, c;
  scanf("%d", &n);
  printf("Fibonacci series terms are:\n");
  for (c = 1; c <= n; c++)
  {
```

```
    printf("%d\n", f(i));
    i++;
  }
  return 0;
}
int f(int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return (f(n-1) + f(n-2));
}
```

The recursive method is less efficient as it involves repeated function calls that may lead to stack overflow while calculating larger terms of the series. Using Memoization (storing Fibonacci numbers that are calculated in an array and using the array for lookup), we can reduce the running time of the recursive algorithm.

4. Tower of Hanoi

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



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

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- 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.

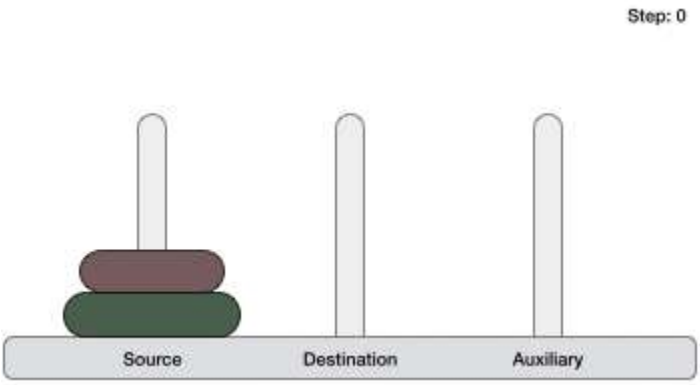
Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

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 the 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 (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

Algorithm:

Step 1 − Move n-1 disks from **source** to **aux**

Step 2 − Move n^{th} disk from **source** to **dest**

Step 3 − Move n-1 disks from **aux** to **dest**

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

```
START
Procedure Hanoi(disk, source, dest, aux)

   IF disk == 1, THEN
      move disk from source to dest
   ELSE
      Hanoi(disk - 1, source, aux, dest)     // Step 1
      move disk from source to dest          // Step 2
      Hanoi(disk - 1, aux, dest, source)     // Step 3
```

```
    END IF

END Procedure
STOP
```

Implementation of TOH using C++

```cpp
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char source_rod, char destination_rod, char auxi_rod)
{
    if (n == 1)
    {
        cout << "Move disk 1 from rod " << source_rod <<" to rod " << destination_rod<<endl;
        return;
    }
    towerOfHanoi(n - 1, source_rod, auxi_rod, destination_rod);  // step1
    cout << "Move disk " << n << " from rod " << source_rod <<" to rod " << destination_rod << endl; //step2
    towerOfHanoi(n - 1, auxi_rod, destination_rod, source_rod);  // step3
}


int main()
{
    int n = 1; // Number of disks
    towerOfHanoi(n, 'S', 'D', 'A'); // S = source rod, D = Destination rod and A auxiliary rod
    return 0;
}
```

5. Applications of recursion and Validity of an Expression

The application of recursion is everywhere in computer science. Few of them are:

1. Recursion is used to solve Factorial, Fibonacci series, Tower of Hanoi etc.
2. The data structure linked list, tree etc. follows recursion.
3. Many sorting algorithms like Quick sort, Merge sort etc. uses recursion.
4. Searching algorithm like BST uses recursion.
5. All the puzzle games like Chess, Candy crush etc. uses recursion.
6. Tree traversal algorithms like inorder, preorder and post order algorithm uses recursion.
7. Graph traversal like BFS and DFS uses recursion.
8. Recursion is used for evaluation of algebraic expressions.
9. Recursion is used for parsing of algebraic expressions.
10. Recursion is used for validating algebraic expressions like prefix and postfix expressions.