Recursion Basics

What is Recursion?

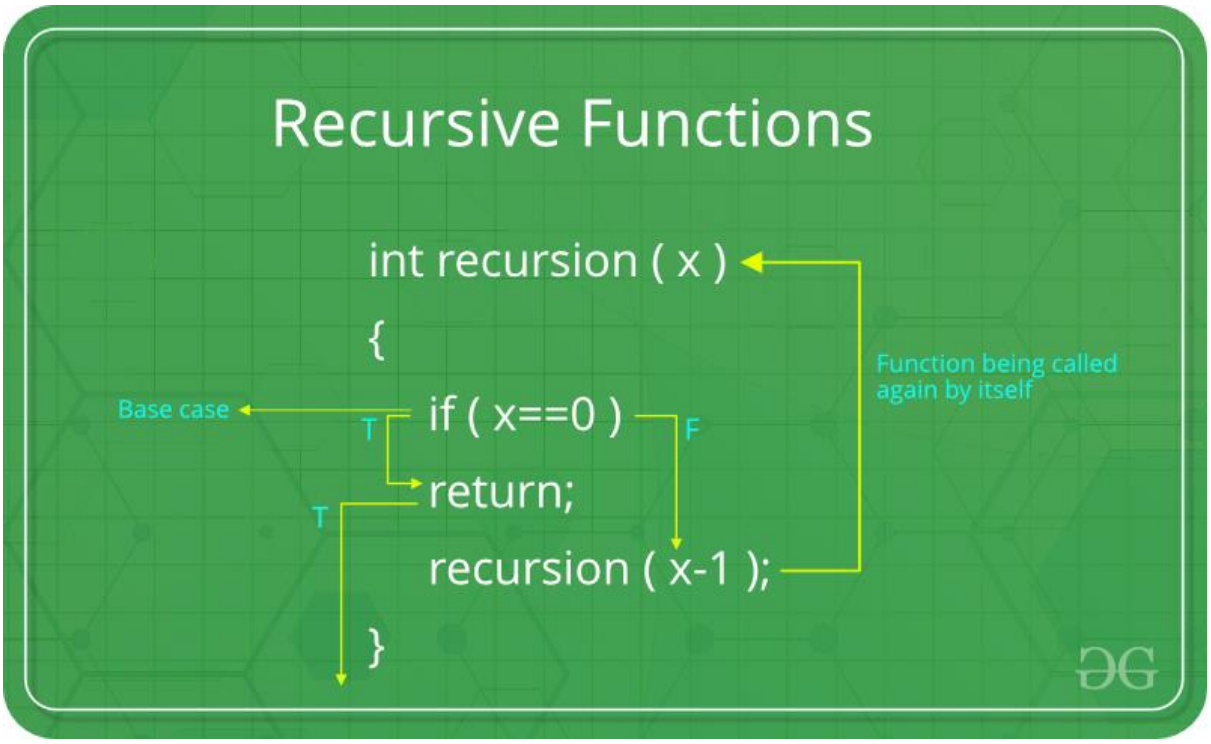
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

What is the base condition in recursion?

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



How a particular problem is solved 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 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.

Why Stack Overflow error occurs 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?

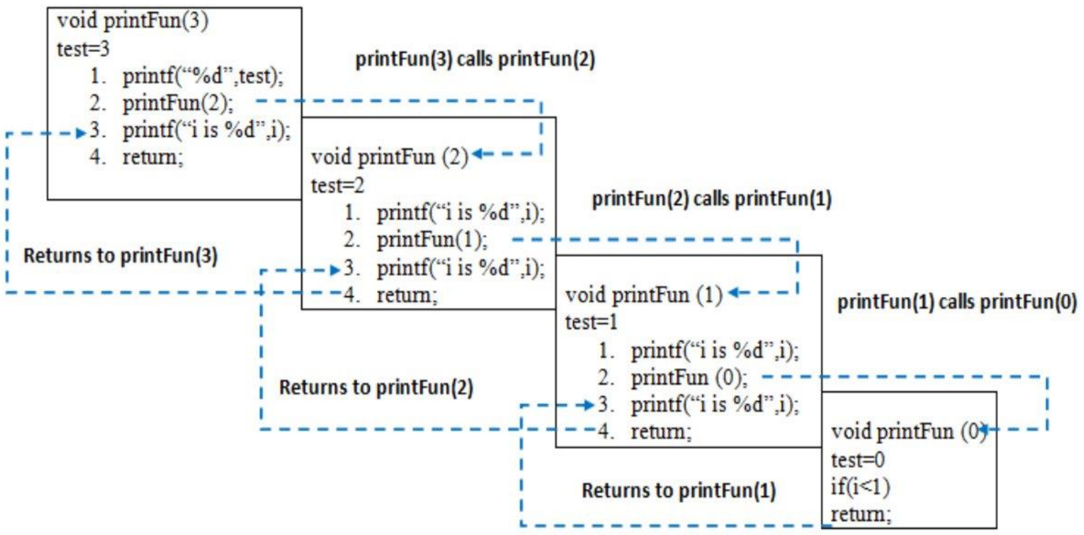
When any function is called from main(), the memory is allocated to it on stack. A recursive function calls itself, the memory for the called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base 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. Let us take the example of how recursion works by taking a simple function:

```
void printFun(int test)

{

    if (test < 1)

        return;

    else

    {

        print test;

        printFun(test-1);    // statement 2

        print test;

        return;
    }
}
// Calling function printFun()

int test = 3;

printFun(test);
```

Output :

```
3 2 1 1 2 3
```

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statements 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun (0)** goes to if statement and it returns to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Disadvantage of Recursion: 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. Recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. A recursive program also has greater time requirements because of function calls and return overhead. **Advantages of Recursion**: 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 the stack data structure.

Print N to 1 Using Recursion

Given a number **N**, the task is to print the numbers from **N** to **1**.
Examples:

Input: *N = 10*
Output: *10 9 8 7 6 5 4 3 2 1*
Input: *N = 7*
Output: *7 6 5 4 3 2 1*

Approach 1: Run a loop from N to 1 and print the value of N for each iteration. Decrement the value of N by 1 after each iteration.
Below is the implementation of the above approach.

C++

```cpp
// C++ program to print all numbers between 1
// to N in reverse order

#include <bits/stdc++.h>
using namespace std;

// Recursive function to print from N to 1
void PrintReverseOrder(int N)
{

        for (int i = N; i > 0; i--)
                cout << i << " ";

}

// Driven Code
int main()
{
        int N = 5;

        PrintReverseOrder(N);

        return 0;
}
```

Output

```
5 4 3 2 1
```

Time Complexity: *O(N)*

Auxiliary Space: *O(1)*

Approach 2: We will use recursion to solve this problem.

1. Check for the base case. Here it is N<=0.
2. If base condition satisfied, return to the main function.
3. If base condition not satisfied, print N and call the function recursively with value (N – 1) until base condition satisfies.

Below is the implementation of the above approach.

C++

```cpp
// C++ program to print all numbers between 1
// to N in reverse order

#include <bits/stdc++.h>
using namespace std;

// Recursive function to print from N to 1
void PrintReverseOrder(int N)
{
	// if N is less than 1
	// then return void function
	if (N <= 0) {
		return;
	}
	else {
		cout << N << " ";

		// recursive call of the function
		PrintReverseOrder(N - 1);
	}
}
```

```
// Driven Code
int main()
{
        int N = 5;

        PrintReverseOrder(N);

        return 0;
}
```

Output

```
5 4 3 2 1
```

Time Complexity: O(N)

Auxiliary Space: O(N)

Print 1 to N Using Recursion

Given a number **N**, the task is to print the numbers from **N to 1.**

Examples:

Input: *N = 10*
Output: *1 2 3 4 5 6 7 8 9 10*
Input: *N = 7*
Output: *1 2 3 4 5 6 7*

<u>Method-1:</u>

C++

```
// C++ program to How will you print
// numbers from 1 to 10 without using a loop?
#include <iostream>
using namespace std;

class gfg
{
```

```
// It prints numbers from 1 to n.
public:
void printNos(unsigned int n)
{
        if(n > 0)
        {
                printNos(n - 1);
                cout << n << " ";
        }
        return;
}
};

// Driver code
int main()
{
        gfg g;
        g.printNos(10);
        return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

Time Complexity: *O(n)*

Auxiliary Space: *O(n)*

Method 2:

C++

```
// C++ program
#include <bits/stdc++.h>
using namespace std;

void printNos(int initial, int last)
{
        if (initial <= last) {
                cout << initial << " ";
                printNos(initial + 1, last);
        }
}

int main()
{
        printNos(1, 10);
        return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

Time Complexity : O(n)

Auxiliary Space: *O(n)*

Tail Recursion

As we read before, that recursion is defined when a function invokes/calls itself.

Tail Recursion: A recursive function is said to be following Tail Recursion if it invokes itself at the end of the function. That is, if all of the statements are executed before the function invokes itself then it is said to be following Tail Recursion.

For Example:

CPP

```
// This is a Tail Recursion

void printN(int N)
{
    if(N==0)
        return;
    else
        cout<<N<<" ";

    printN(N-1);
}
```

The above function call for **N** = **5** will print:

```
5 4 3 2 1
```

Which one is Better-Tail Recursive or Non Tail-Recursive?

The tail-recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

Can a non-tail recursive function be written as tail-recursive to optimize it?

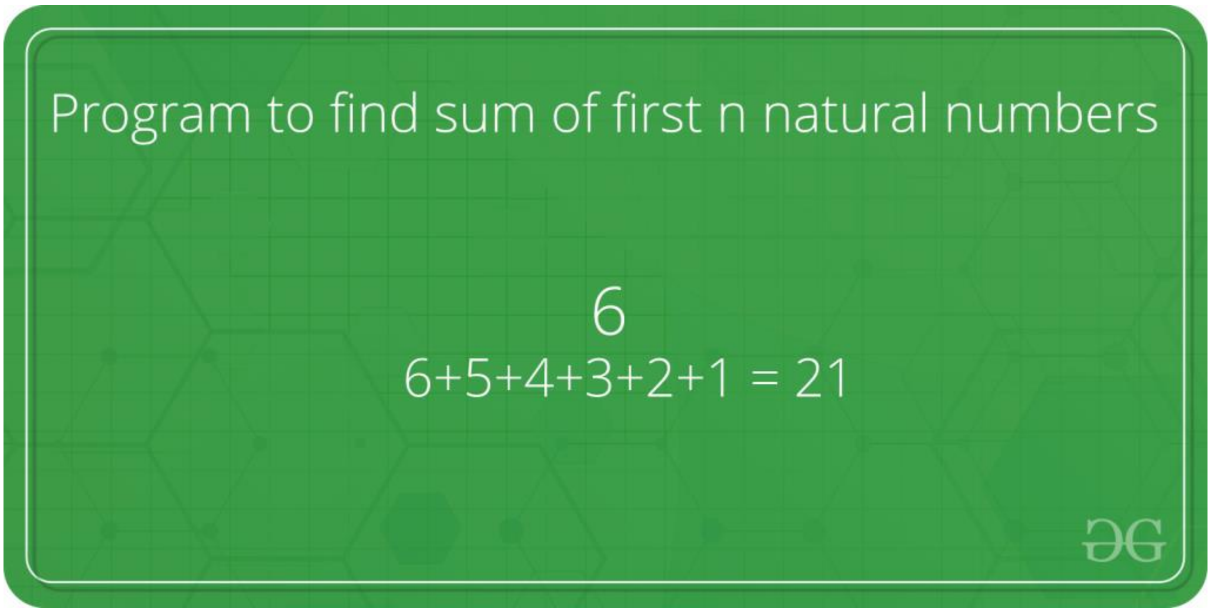
Consider the following function to calculate factorial of N. Although it looks like Tail Recursive at first look, it is a non-tail-recursive function. If we take a closer look, we can see that the value returned by **fact(N-1)** is used in **fact(N)**, so the call to fact(N-1) is not the last thing done by fact(N).

```
int fact(int N)
{
    if (N == 0)
        return 1;

    return N*fact(N-1);
}
```

The above function can be written as a Tail Recursive function. The idea is to use one more argument and accumulate the factorial value in the second argument. When N reaches 0, return the accumulated value.

```
int factTR(int N, int a)
{
    if (N == 0)
        return a;

    return factTR(N-1, N*a);
}
```

Natural Number Sum using Recursion



Given a number n, find sum of first *n* natural numbers. To calculate the sum, we will use a recursive function recur_sum().
Examples :

```
Input : 3
Output : 6
Explanation : 1 + 2 + 3 = 6

Input : 5
Output : 15
Explanation : 1 + 2 + 3 + 4 + 5 = 15
```

Below is code to find the sum of natural numbers up to n using recursion :

C++

```
// C++ program to find the
// sum of natural numbers up
// to n using recursion
#include <iostream>
using namespace std;

// Returns sum of first
// n natural numbers
int recurSum(int n)
{
    if (n <= 1)
        return n;
    return n + recurSum(n - 1);
}

// Driver code
int main()
{
    int n = 5;
    cout << recurSum(n);
    return 0;
}
```

Output :

```
15
```

Time complexity : O(n)

Auxiliary space : O(n)

<u>To solve this question , **iterative approach** is the best approach because it takes constant or O(1) auxiliary space and the time complexity will be same O(n).</u>

Given a string, write a recursive function that checks if the given string is a palindrome, else, not a palindrome.

Examples:

```
Input : malayalam
```

```
Output : Yes
Reverse of malayalam is also
malayalam.

Input : max
Output : No
Reverse of max is not max.
```

The idea of a recursive function is simple:

```
1) If there is only one character in string
   return true.
2) Else compare first and last characters
   and recur for remaining substring.
```

Below is the implementation of the above idea:

C++

```cpp

// A recursive C++ program to
// check whether a given number
// is palindrome or not
#include <bits/stdc++.h>
using namespace std;

// A recursive function that
// check a str[s..e] is
// palindrome or not.
bool isPalRec(char str[],
              int s, int e)
{

    // If there is only one character
    if (s == e)
    return true;

    // If first and last
    // characters do not match
    if (str[s] != str[e])
    return false;

    // If there are more than
    // two characters, check if
    // middle substring is also
    // palindrome or not.
    if (s < e + 1)
    return isPalRec(str, s + 1, e - 1);

    return true;
}

bool isPalindrome(char str[])
{
    int n = strlen(str);

    // An empty string is
    // considered as palindrome
    if (n == 0)
        return true;

    return isPalRec(str, 0, n - 1);
}

// Driver Code
int main()
{
    char str[] = "geeg";

    if (isPalindrome(str))
    cout << "Yes";
    else
    cout << "No";

    return 0;
}
```

```
// This code is contributed by shivanisinghss2110
```

Output

```
Yes
```

Time Complexity: O(n)
Auxiliary Space: O(n)

Another Approach :

Basically while traversing check whether ith and n-i-1th index are equal or not.

If there are not equal return false and if they are equal then continue with the recursion calls.

C++

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

bool isPalindrome(string s, int i){

    if(i > s.size()/2){
       return true ;
    }

    return s[i] == s[s.size()-i-1] && isPalindrome(s, i+1) ;

}


int main()
{
    string str = "geeg" ;
    if (isPalindrome(str, 0))
    cout << "Yes";
    else
    cout << "No";

    return 0;

}
```

Output

```
Yes
```

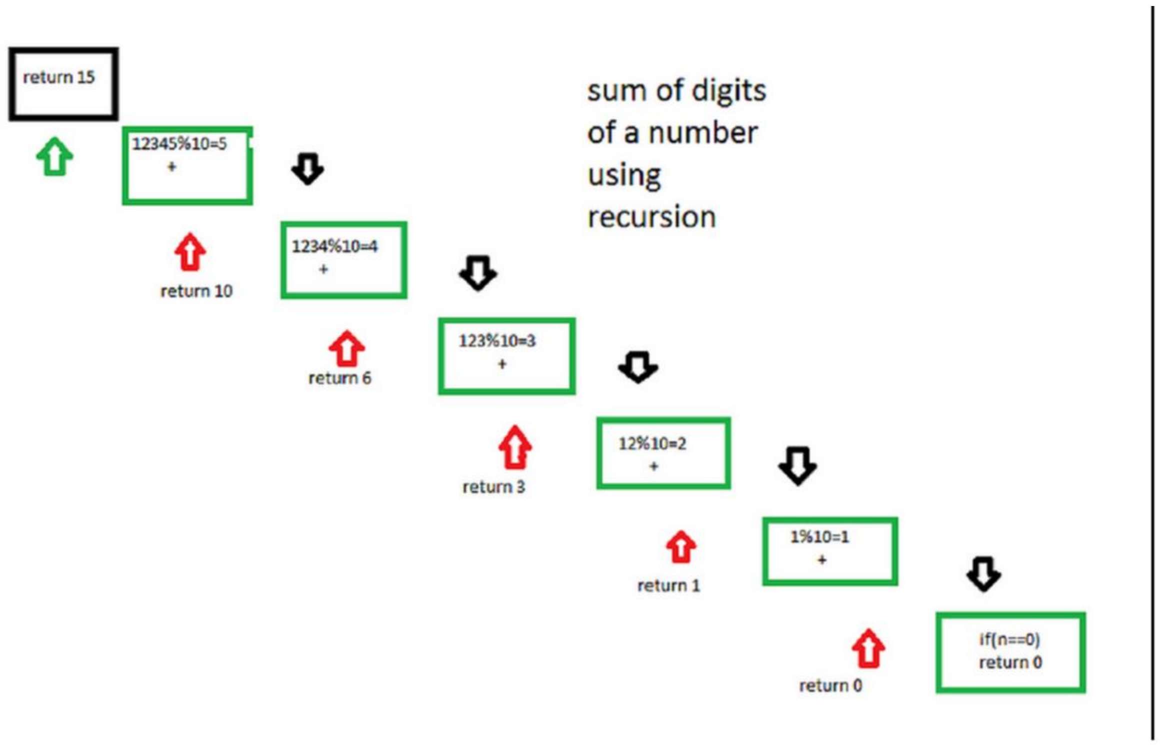
Time Complexity: O(n)
Auxiliary Space: O(n)

Sum of Digits Using Recursion

Given a number, we need to find sum of its digits using recursion.
Examples:

```
Input : 12345

Output : 15


Input : 45632

Output :20
```

The step-by-step process for a better understanding of how the algorithm works.
Let the number be 12345.
Step 1-> 12345 % 10 which is equal-too 5 + (send 12345/10 to next step)
Step 2-> 1234 % 10 which is equal-too 4 + (send 1234/10 to next step)
Step 3-> 123 % 10 which is equal-too 3 + (send 123/10 to next step)
Step 4-> 12 % 10 which is equal-too 2 + (send 12/10 to next step)
Step 5-> 1 % 10 which is equal-too 1 + (send 1/10 to next step)
Step 6-> 0 algorithm stops
following diagram will illustrate the process of recursion



C++Java

```
// Recursive C++ program to find sum of digits
// of a number
#include <bits/stdc++.h>
using namespace std;

// Function to check sum of digit using recursion
int sum_of_digit(int n)
{
    if (n == 0)
    return 0;
    return (n % 10 + sum_of_digit(n / 10));
}

// Driven code
int main()
{
    int num = 12345;
    int result = sum_of_digit(num);
    cout << "Sum of digits in "<< num
       <<" is "<<result << endl;
    return 0;
}
```

Output:

```
Sum of digits in 12345 is 15
```

Besides writing (n==0 , then return 0) in the code given above we can also write it in this manner , there will be no change in the output .

```
if(n<10) return n; By writing this there will be no need to call the fu
nction for the numbers which are less than 10
```

Time complexity : O(logn) where n is the given number.

Auxiliary space : O(logn) due to recursive stack space.

Rope Cutting Problem

Given a rope of length **N** meters, and the rope can be cut in only 3 sizes **A**, **B** and **C.** The task is to maximizes the number of cuts in rope. If it is impossible to make cut then print **-1**. **Examples:**

Input:

N = 17, A = 10, B = 11, C = 3

Output: 3

Explanation: The maximum cut can be obtain after making 2 cut of length 3 and one cut of length 11.

Input: N = 10, A = 9, B = 7, C = 11

Output: -1

Explanation: It is impossible to make any cut so output will be -1.

Naive Approach: Using Recursion

C++Java

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

```

```
int maxCuts(int n, int a, int b, int c)
{
        if(n == 0)
                return 0;
        if(n <= -1)
                return -1;

        int res = max(maxCuts(n-a, a, b, c),
                   max(maxCuts(n-b, a, b, c),
                   maxCuts(n-c, a, b, c)));

        if(res == -1)
                return -1;

        return res + 1;
}
int main() {

        int n = 5, a = 2, b = 1, c = 5;

        cout<<maxCuts(n, a, b, c);

        return 0;
}
```

Output:

```
5
```

Time Complexity : O(3^n)

Space Complexity : O(n), due to recursive call stack.

Generate Subsets

Given a set represented as a string, write a recursive code to print all the subsets of it. The subsets can be printed in any order.

Examples:

```
Input :  set = "abc"
Output : "". "a", "b", "c", "ab", "ac", "bc", "abc"

Input : set = "abcd"
Output : "" "a" "ab" "abc" "abcd" "abd" "ac" "acd"
         "ad" "b" "bc" "bcd" "bd" "c" "cd" "d"
```

The idea is to consider two cases for every character. (i) Consider current character as part of the current subset (ii) Do not consider current character as part of the current subset.

C++Java

```
// CPP program to generate power set
#include <bits/stdc++.h>
using namespace std;

// str : Stores input string
// curr : Stores current subset
// index : Index in current subset, curr
void powerSet(string str, int index = 0, string curr = "")
{
    int n = str.length();

    // base case
    if (index == n) {
        cout << curr << endl;
        return;
    }

    // Two cases for every character
    // (i) We consider the character
    // as part of current subset
    // (ii) We do not consider current
    // character as part of current
    // subset
    powerSet(str, index + 1, curr + str[index]);
    powerSet(str, index + 1, curr);
}

// Driver code
int main()
{
    string str = "abc";
    powerSet(str);
    return 0;
}
```

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods (**A**, **B**, and **C**) and **N** disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod **A**. The objective of the puzzle is to move the entire stack to another rod (here considered **C**), obeying the following simple rules:

- Only one disk can be moved at a time.
- 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.
- No disk may be placed on top of a smaller disk.

Examples:

Input: *2*
Output: *Disk 1 moved from A to B*
Disk 2 moved from A to C
Disk 1 moved from B to C

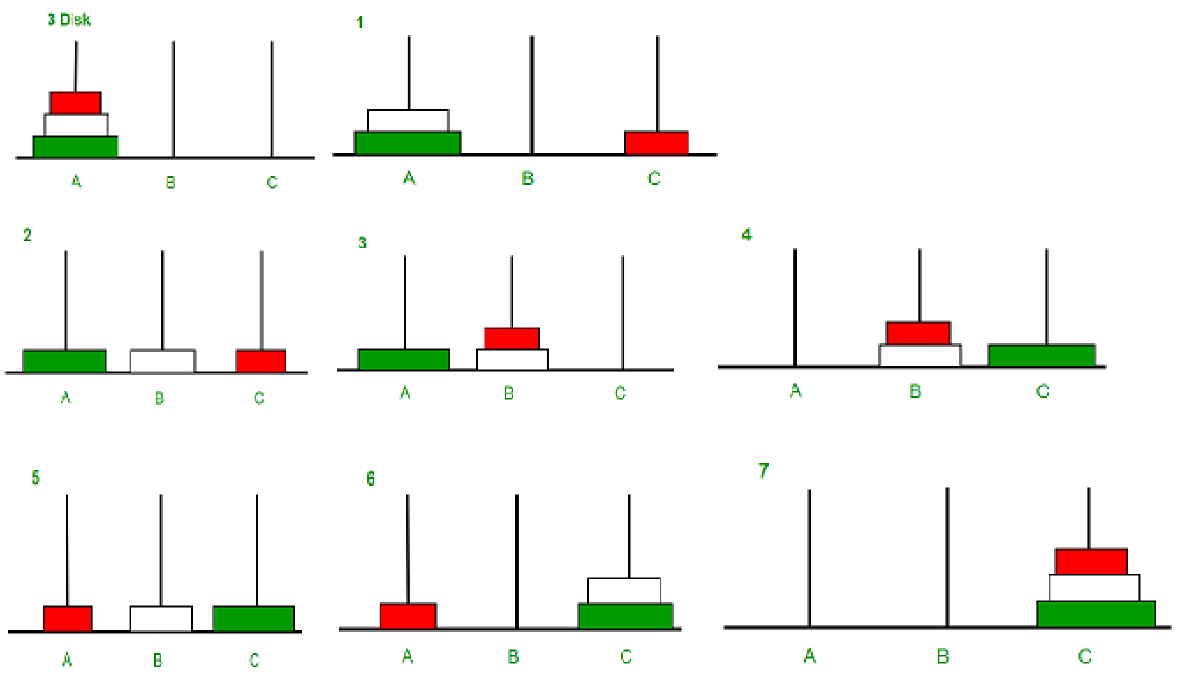
Input: *3*
Output: *Disk 1 moved from A to C*
Disk 2 moved from A to B
Disk 1 moved from C to B

Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C

Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- *Shift 'N-1' disks from 'A' to 'B', using C.*
- *Shift last disk from 'A' to 'C'.*
- *Shift 'N-1' disks from 'B' to 'C', using A.*



Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod.**
- Make a function call for N – 1 th disk.
- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for N – 1 th disk.

Below is the implementation of the above approach.

JavaC++

```
// JAVA recursive function to
// solve tower of hanoi puzzle
import java.io.*;
import java.math.*;
import java.util.*;
class GFG {
	static void towerOfHanoi(int n, char from_rod,
	                                             char to_rod, char
aux_rod)
	{
		if (n == 0) {
			return;
		}
		towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
		System.out.println("Move disk " + n + " from rod "
		                           + from_rod + " to rod "
		                           + to_rod);
		towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
	}

	// Driver code
	public static void main(String args[])
	{
		int N = 3;

		// A, B and C are names of rods
		towerOfHanoi(N, 'A', 'C', 'B');
	}
}
```

Output

```
Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C
```

Time complexity: O(2^N), There are two possibilities for every disk. Therefore, 2 * 2 * 2 * . . . * 2(N times) is 2^N
Auxiliary Space: O(N), Function call stack space

Josephus Problem

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are skipped and kth person is killed in a circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

For example, if n = 5 and k = 2, then the safe position is 3. Firstly, the person at position 2 is killed, then the person at position 4 is killed, then the person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives.
If n = 7 and k = 3, then the safe position is 4. The persons at positions 3, 6, 2, 7, 5, 1 are killed in order, and the person at position 4 survives.

The problem has following recursive structure.

```
josephus(n, k) = (josephus(n - 1, k) + k-1) % n + 1
josephus(1, k) = 1
```

How does this recursion work?
When we kill k-th person, n-1 persons are left, but numbering starts from k+1 and goes in modular way.

(k+1)-th person in the original circle is now the first person.
n-th person in the original circle is now (n-k)-th person.
1-st person in the original circle is now (n-k+1)-th person.
(k-1)-th person in the original circle is now (n-1)-th person.

So we add (k-1) to the returned position to handle all cases and keep the modulo under n. Finally, we add 1 to the result.

This solution is not easy to think at the first moment.

A simple solution that comes to our mind is

(josephus(n-1, k) + k) % n

We add k because we shifted the positions by k after the first killing. The problem with the above solution is that the value of (josephus(n-1, k) + k) can become n and overall solution can become 0. But positions are from 1 to n. To ensure that, we never get n, we subtract 1 and add 1 later. This is how we get

(josephus(n-1, k) + k - 1) % n + 1

Following is a simple recursive implementation of the Josephus problem. The implementation simply follows the recursive structure mentioned above.

C++Java

```
#include <stdio.h>

int josephus(int n, int k)
{
  if (n == 1)
    return 1;
  else
    /* The position returned by josephus(n - 1, k)
       is adjusted because the recursive call
       josephus(n - 1, k) considers the original
       position k%n + 1 as position 1 */
    return (josephus(n - 1, k) + k-1) % n + 1;
}

// Driver Program to test above function
int main()
{
  int n = 14;
  int k = 2;
  printf("The chosen place is %d",
                    josephus(n, k));
  return 0;
}
```

Time Complexity: O(n)

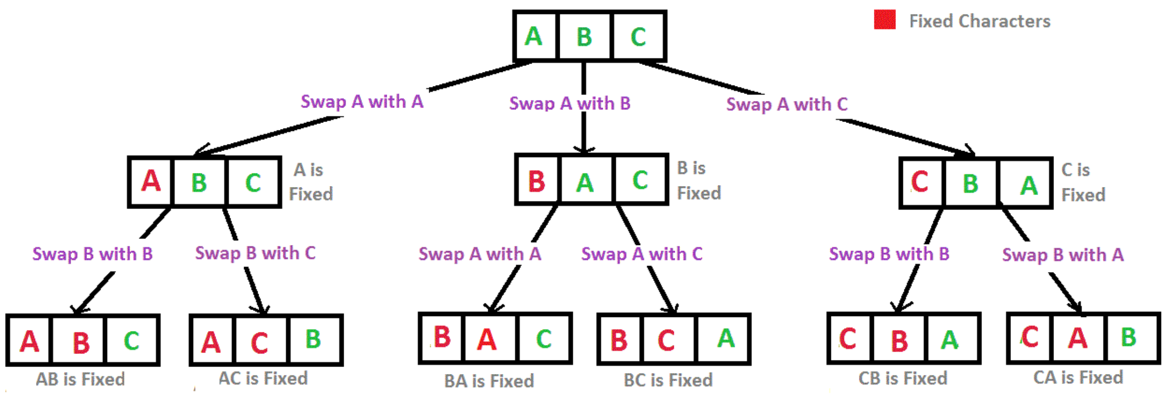
Printing all Permutations

Given a string, print all permutations of it.

Input : str = "ABC"
Output : ABC ACB BAC BCA CAB CBA

Idea: We iterate from first to last index. For every index i, we swap the i-th character with the first index. This is how we fix characters at the current first index, then we recursively generate all permutations beginning with fixed characters (by parent recursive calls). After we have recursively generated all permutations with the first character fixed, then we move the first character back to its original position so that we can get the original string back and fix the next character at first position.

Illustration:
We swap 'A' with 'A'. Then we recursively generate all permutations beginning with A. While returning from the recursive calls, we revert the changes made by them using the same swap again. So we get the original string "ABC". Then we swap 'A' with 'B' and generate all permutations beginning with 'B'. Similarly, we generate all permutations beginning with 'C'



Recursion Tree for Permutations of String "ABC"

C++Java

```
// C++ program to print all
// permutations with duplicates allowed
#include <bits/stdc++.h>
using namespace std;

/* Function to print permutations of string
This function takes three parameters:
1. String
2. Starting index of the string
3. Ending index of the string. */
void permute(string &str, int l, int r)
{
    if (l == r)
        cout << str << " ";
    else
    {
        for (int i = l; i <= r; i++)
        {
            swap(str[l], str[i]);
            permute(str, l+1, r);
            swap(str[l], str[i]);
        }
    }
}


/* Driver program to test above functions */
int main()
{
    string str = "ABC";
    permute(str, 0, str.length()-1);
    return 0;
}
```