Back To Course

Learn
Classroom
Theory
Quiz

Recursion Basics

What is Recursion?

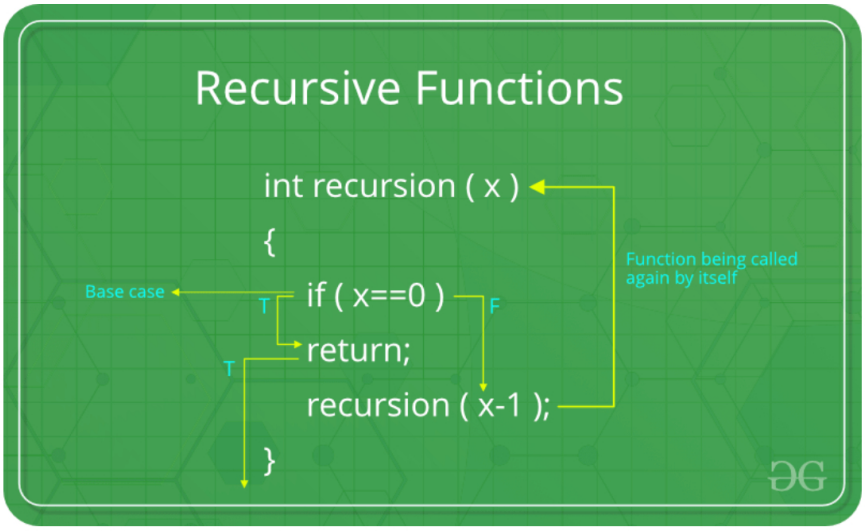
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder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

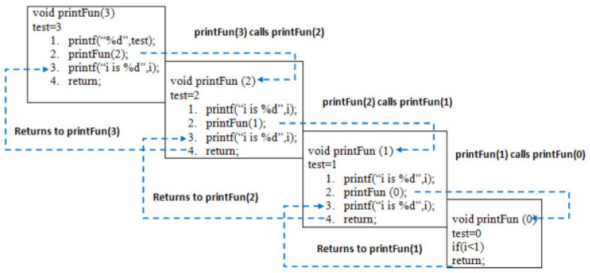
Let us take the example of how recursion works by taking a simple function:

```
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.

- Basic Problems on Recursion

Problem 1

Given an unsorted array of **N** elements and an element **X**. The task is to write a recursive function to check whether the element X is present in the given array or not.

Example:

```
array[] = {1, 2, 3, 4, 5}
X = 3.

The function should return True, as 3 is
present in the array.
```

Solution: The idea is to compare the first element of the array with X. If the element matches with X then return True otherwise recur for the remaining part of the array.

The **recursive function** will somewhat look like as shown below:

```
// arr[] is the given array
// l is the lower bound in the array
// r is the upper bound
// x is the element to be searched for
// l and r defines that search will be
// performed between indices l to r

bool recursiveSearch(int arr[], int l,
                           int r, int x)
{
    if (r < l)
        return false;
    if (arr[l] == x)
        return true;
    if (arr[r] == x)
        return true;

    return recursiveSearch(arr, l + 1,
                             r - 1, x);
}
```

Time Complexity: The above algorithm runs in O(N) time where N is the number of elements present in the array.
Space Complexity: There is no extra space used however the internal stack takes O(N) extra space for recursive calls.

Problem 2

Given a string, the task is to write a recursive function to check if the given string is palindrome or not.

Examples:

```
Input : string = "malayalam"
Output : Yes
Reverse of malayalam is also
malayalam.

Input : string = "max"
Output : No
Reverse of max is not max.
```

Solution: The idea to write the recursive function is simple and similar to the above problem:

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

Recursive Function:

```
// s and e defines the start and end index of string

bool isPalindrome(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)
        return isPalindrome(str, s + 1, e - 1);

    return true;
}
```

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:

```
1
2  // This is a Tail Recursion
3
4  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);
}
```

Explanation of Subset Generation Problem

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
```

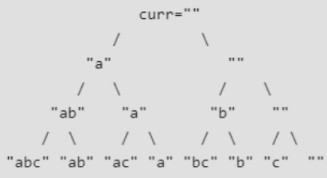
```
    // (i) We consider the character
    // as part of current subset
    // (ii) We do not consider current
    // character as part of current
    // subset
    powerSet(str, index + 1, curr + str[index]);
    powerSet(str, index + 1, curr);
}

// Driver code
```

Run

Output:

```
abc
ab
ac
a
bc
b
c
```

Let us understand the recursion with an example "abc". Every node in the below tree represents the string **curr**.



At root, index = 0.
At next level of tree index = 1
At third level, index = 2
At fourth level index = 3 (becomes equal to string length), so we print the subset.

Explanation of Joesphus 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.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

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 Python3

```cpp
#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;
}
```

Run

Time Complexity: O(n)

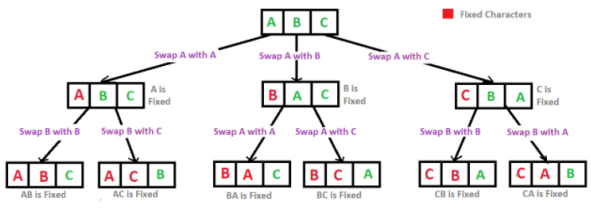
Explanation of permutations of a string

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 Python

```cpp
// C++ program to print all
```

```cpp
// 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";
```

Run

Output:

```
ABC ACB BAC BCA CBA CAB
```

How do we get permutations in lexicographically sorted order? We can put all permutations in an array instead of printing them immediately. Then we can sort the array and print the array.

Report An Issue

If you are facing any issue on this page. Please let us know.