Recursion

Introduction

Recursion acts as a platform for **dynamic problems**. Recursion means, when a function calls itself, inside its definition. Recursion has proved to be an important technique in which a function calls for itself, on a smaller portion of the same problem, to make a complex problem simpler by breaking down the original problem into small chunks. Recursive techniques have been proven as a source to provide simple and meaningful solutions to complex problems, however, they are not always efficient. Recursive solutions are often used in scenarios, where the time taken to develop software is an important factor.

```
Void Rec()
{
   print("Coding Ninjas");
   Rec();//function calling itself, inside its own definition
}

main()
{
  Rec();
}
```

In the above code snippets, we have 2 functions, namely **Rec ()** and **main ()** function. The main function of the **Rec ()** function is to print the text i.e., **"coding ninjas** ", after printing

the text the function again calls itself inside its definition, and the **main ()** function is also called the **Rec ()** function. When we encounter such a type of phenomenon, we term them as recursion.

Types of Recursions:

(a) Direct Recursion.

In direct recursion, there is only one function call, i.e., Function calling itself.

```
Void Rec()
{
   print("Coding Ninjas");
   Rec();//function calling itself, inside its own definition
}

main()
{
  Rec();
}
```

The code given in the above snippet is an example of direct recursion because the function is calling itself again, inside the function definition.

(b) Indirect function.

Indirect recursion occurs when a function is called not by itself but by another function that is called (either directly or indirectly). For example, if a function F calls G and then this G function calls F.

```
Void Rec1()

{

   print("Coding Ninjas");

   Rec();//function calling another function. Which is called by itself.

}


Void Rec()

{

   Rec1();

}
```

Benefits of Recursion

- Recursion helps you to divide your problems into smaller chunks, and after that solve those smaller chunks, to find a solution for the whole problem.
- Recursive code has better readability, and it's easy to write.

Structure of Recursive Function

```
Void Rec(){

            Base Case;

            Recursive Condition;

}
```

Every recursive function consists of **2** components i.e **Base case and recursive condition**. The **base case** is considered as a case at which our recursion will end. The recursive case acts as a general case of the problem we are trying to solve.

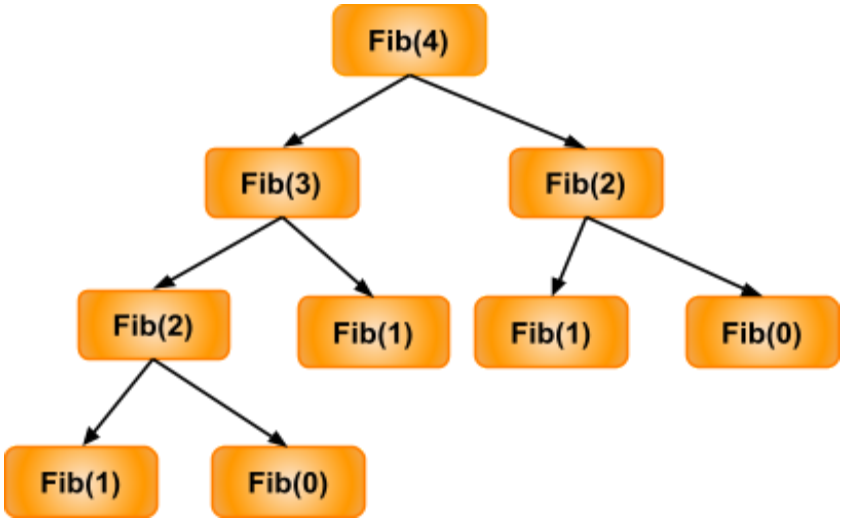
Find the Nth fibonacci numbers

Problem Statement: Find the **N^{th}** Fibonacci number**.**

Algorithm:

- We can break the algorithm of Fibonacci in the teams of the previous 2 states **F(n) = F(n-1) + F(n-2).**
- Now, we know that the **0^{th}** Fibonacci number is **0** and the **1^{st}** Fibonacci number is **1.** (by the definition). We can use this as a base case condition.
- Now, we can implement the above algorithm using recursion because we have both the components (Base case and Recursive condition).

  Recursive tree: if n=4.



Code:

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;

int fib(int n){
    // base condition
    if(n==0) return 0;
    if(n==1) return 1;

    int nth = fib(n-1)+fib(n-2);
    return nth;
}

int main(){
    int n;
    cin>>n;
    int ans = fib(n);
    cout<<ans<<"\n";
    return 0;
}
```

Time Complexity: O(2^N) where **N** is the position of the Fibonacci number we need to find.

Because for every value of **n (1<=n<=N)**, there are two function calls fib(n-1) and fib(n-2). Therefore, the time complexity of the algorithm is exponential.

Space Complexity: O(N) where **N** is the position of the Fibonacci number we need to find. Due to recursive stack space, the height of the recursive tree is equal to **N.**

Sum of numbers in an array

Problem statement:

The problem statement says that you are given an array of **N**, integers and you have to find the sum of all integers, which is present in the array using recursion.

Algorithm:

To apply recursion we need to find the recurrence relation, which is not very hard to think about in this problem. We can see that the sum of all the numbers is nothing but the sum of the last element and the sum of the rest of the array.

Sum = arr[N] + Sum_of(a[1], a[N-1]).

- First of all, we declare a global variable **sum** that will store our answer.
- Then we initialize this variable with 0 as in the beginning the sum is **0**.
- Now, we create a pointer **curr** that is pointing at the beginning of the array initially. It will keep the track of the current element of the array.
- At every function call, we add the value present at the beginning of the array or on which our pointer **curr** is pointing to our global variable **sum.**
- Now, as we are done with the first element of the array we will pass the rest of the array to our function and repeat the same process.
- We keep repeating this process until we traverse the whole array. As our **curr** pointer starts pointing beyond the last element of the array we stop our process and this will be our base case.

Code:

```
#include <bits/stdc++.h>

using namespace std;

Int sum=0;

int sumOfVector(int curr , vector<int>& a){

      int n=a.size();

      if(curr == n) return 0; //Base case

      return sum+=(a[curr]+sumOfVector(curr+1,a));

}

int main() {

      int n;

      cin>>n;

      vector<int>a(n);

      for(int i=0;i<n;i++) cin>>a[i];

      int ans = sumOfVector(0,a); //Recursive function

      cout<<ans<<"\n";

}
```

Time Complexity: O(N) where **N** is the size of the array.
As we are traveling the entire array.

Space-complexity: O(N) where **N** is the size of the array.

Due to recursive stack space, the height of the recursive tree is equal to **N.**

Sum of digits

Problem statement:

Find the **sum of digits** entered by the user using recursion.

Algorithm:

To apply recursion we need to components first is base condition and the recursive condition:

- First of all, we extract the rightmost digit of the number using modulo **(N%10)**.
- Now we add this to our **answer.**
- Now as we utilize the rightmost digit we don't need it anymore so we remove it and apply the same process on the number which remains after removing the rightmost digit **(N/10)**.
- We keep doing this process until we left with **0**, Which is our base case.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int sumOfDigits(int num){

      if(num==0) return 0; //Base Case

      return (num%10)+sumOfDigits(num/10);

}

int main() {

      int num;

      cin>>num;

      cout<<sumOfDigits(num); //Recursive function
```

```
    return 0;
}
```

Time Complexity: O(D) Where **D** is the number of digits present in the number**.** Because our function is operating on all the digits.

Space-complexity: O(D) Where **D** is the number of digits present in the number**.** Due to recursive stack space, the height of the recursive tree is equal to **D.**

String to Integer.

Problem Statement:

You are given an integer in string format (for eg:"1235") and you need to convert this string in integer format. The number must remain the same.

Algorithm:

We are given a string, while solving a recursive problem we first need to find a recursive condition. Firstly, let's divide the problem into smaller chunks.

For example, let's assume that we first extract element **5** from our string and then convert it into an integer, and after that add this extracted element (which is converted into an integer), to the remaining elements present in the string.

After performing the above-mentioned task, our equation will look like **("123") + 5.**

- If the string consists of only one value, the value will be returned as it is.
- Otherwise, the digit will be multiplied by the appropriate power of **10** and added to the result, which is initially 0.
- This character will be removed from the string and the remaining string will be passed to the recursive function.

- These steps will be repeated till there is only a single character left in the remaining string.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int ConvertToInt(int curr, string s){

      if(curr<0) return 0;

      return ((s[curr]-'0') + 10*ConvertToInt(curr-1,s));

}

int main() {

      string s="32833";

      cout<<ConvertToInt(s.size()-1,s);

}
```

Time Complexity: **O(N)**, where **N** is the length of the string because we are traversing through every character present in the string.

Space Complexity: **O(N)**, where **N** is the length of the string.
Due to recursive stack space, the height of the recursive tree is equal to **N.**