Day 75/180 Recursion Problem on Leetcode

1: fibonacci:

- What is fibonacci number ?

```
The Fibonacci numbers are a sequence of numbers in
which each number is the sum of the two preceding
ones, usually starting with 0 and 1. The sequence
goes: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on.

Formally, the Fibonacci sequence is defined by the
recurrence relation:

[ F(n) = F(n-1) + F(n-2) ]

with initial conditions:

[ F(0) = 0,  F(1) = 1 ]
```

- So, clearly to this problem recursively :

  F(N) = F(n-1) + F(n-2).

  And the base cases are :

  F(0) = 0, and F(1) = 1.

```
int fib(int n) {
         if(n == 0)
           return 0;
        if(n == 1)
           return 1;

        return fib(n-1) + fib(n-2);

    }
```

Time complexity : O(2^n)

2: Nth Stair:

This is recursive approach, so, it will not pass all the test cases, to pass all test cases we need DP concept (will taught later in course).

The base case is when `**n**` is **1** or **2**, in which case the number of ways is simply `**n**`. Otherwise, the function calls itself recursively with **`climbStairs(n-1)`** and **`climbStairs(n-2)`,** representing the two possible ways to reach the current step. The base cases ensure that the recursion terminates.

However, keep in mind that this recursive solution can become inefficient for large values of `n` due to repeated calculations.
To optimize it, you might want to use memoization to store and reuse previously computed results.

- So, clearly to this problem recursively :
  F(N) = F(n-1) + F(n-2).
  And the base cases are :
  F(1) = 1, and F(2) = 2.

```
int climbStairs(int n) {
        if(n == 1)
           return 1;
        if(n == 2)
           return 2;
        return  climbStairs(n-1) + climbStairs(n-2);
    }
```

Time complexity : O(2^n)

3:GCD:

The greatest common divisor (GCD) of two positive integers A and B is the largest positive integer that divides both A and B without leaving a remainder. There are several algorithms to find the GCD of two numbers. One commonly used algorithm is the Euclidean Algorithm.

1. **Base Case**: If `b` is 0, it means that the GCD is found, and the function returns `a`.

2. **Recursive Case**: If `b` is not 0, the function calls itself with the arguments `b` and `a % b`, effectively replacing the original pair `(a, b)` with `(b, a % b)`.

3. The recursion continues until the base case is reached, at which point the GCD is returned.

The algorithm exploits the fact that the GCD of two numbers is the same as the GCD of the smaller number and the remainder when the larger number is divided by the smaller number. The recursion repeats this process until the remainder becomes 0, at which point the GCD is found.

```
public int gcd(int a , int b)
    {
        //code here
        if (b == 0)
        {
            return a;
        }
        else
        {
           return gcd(b, a % b);
        }
    }
```

Time complexity of the given gcd (greatest common divisor) function is logarithmic, specifically O(log(min(a, b))). This is because each recursive call reduces the problem size by dividing the larger number by the remainder of the division of the two numbers (a % b)

4:Count number of hops

Similar to **N stairs** problem.

This is recursive approach, so, it will not pass all the test cases, to pass all test cases we need DP concept (will taught later in the course).

The recursive function `countWays` that calculates the number of ways to climb a staircase with `n` steps, where a person can take 1, 2, or 3 steps at a time.

The base cases are provided for `n = 1`, `n = 2`, and `n = 3`. For any other value of `n`,

the function recursively calculates the total number of ways by summing the counts for `(n-1)`, `(n-2)`, and `(n-3)` steps.

The approach relies on breaking down the problem into smaller subproblems and combining their solutions to find the overall number of ways to climb the staircase.

```
  long countWays(int n)
   {
       // add your code here
       if(n == 1)
        return 1;

       if(n == 2)
        return  2;

        if(n == 3)
          return 4;

       long ans = countWays(n-1) + countWays(n-2) +
countWays(n-3);
       return ans;

   }
```

Time complexity : O(3^n)

5: Fibonacci Series up to Nth term

You can solve this problem by calculating fibonacci for every position , then, return the answer. But you will get time limit exceed.

Below the DP approach is given, you will learn DP concept later on the course.

This code generates a Fibonacci series up to the Nth term and returns the result as a vector of long long integers.

1. **Initialization**: It starts by initializing an empty vector named `fibonacci`.

2. **Base Cases**: If `N` is greater than or equal to 0, it adds 0 to the vector. If `N` is greater than or equal to 1, it adds 1 to the vector. These are the first two terms of the Fibonacci series.

3. **Fibonacci Calculation**: It then enters a loop starting from `i = 2` up to `N`. In each iteration, it calculates the next Fibonacci number by summing up the last two numbers in the sequence (`fibonacci[i-1]` and `fibonacci[i-2]`). It adds this new Fibonacci number to the vector.

4. **Return**: Finally, it returns the vector containing the Fibonacci series up to the Nth term.

This code efficiently generates the Fibonacci series iteratively, avoiding the need for recursive calculations, which can be computationally expensive for large values of N.

```
vector<long long> Series(int N) {
        // COde here
        vector<long long> fibonacci;
    if(N >= 0)
        fibonacci.push_back(0);
    if(N >= 1)
        fibonacci.push_back(1);
    for(int i=2; i<=N; i++)
        fibonacci.push_back(fibonacci[i-1] + fibonacci[i-2]);
    return fibonacci;
    }
```

Time Complexity : O(N) , **Space Complexity :** O(N)