Math and Bitmanipulation

Math

1.Extended Euclidean Algorithm(Gcd and Hcf)

We already know Basic Euclidean Algorithm. Now using the Extended Euclidean Algorithm, given a and b calculate the **GCD** and integer coefficients **x, y.** Using the same. x and y must satisfy the equation **ax + by = gcd(a, b)** .

Example 1:

```
Input:
a = 35
b = 15
Output:
5 1 -2
Explanation:
gcd(a,b) = 5
35*1 + 15*(-2) = 5
```

Example 2:

```
Input:
a = 30
b = 20
Output:
10 1 -1
Explanation:
gcd(30,20) = 10
30*(1) + 20*(-1) = 10
```

```java
class Solution {
    static int[] gcd(int a, int b) {
        // code here
         if (b == 0) {
            return new int[]{a, 1, 0};
        }

        // Recursively call gcd(b, a % b)
        int[] vals = gcd(b, a % b);
        int gcd = vals[0];
        int x1 = vals[1];
        int y1 = vals[2];

        // Update x and y using results of recursion
        int x = y1;
        int y = x1 - (a / b) * y1;

        return new int[]{gcd, x, y};
    }
}
```

2.Sieve of Eratosthenes

Given a positive integer **n**, calculate and return all prime numbers less than or equal to **n** using the **Sieve of Eratosthenes** algorithm.

A **prime number** is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Examples:

Input: n = 10
Output: 2 3 5 7
Explanation: Prime numbers less than equal to 10 are 2 3 5 and 7.

Input: n = 35
Output: 2 3 5 7 11 13 17 19 23 29 31
Explanation: Prime numbers less than equal to 35 are 2 3 5 7 11 13 17 19 23 29 and 31.

Constraints:

```java
// User function Template for Java
class Solution {
    static ArrayList<Integer> sieveOfEratosthenes(int n) {
        // code here
        ArrayList<Integer> primes = new ArrayList<>();

        boolean[] isPrime = new boolean[n + 1];
        for (int i = 2; i <= n; i++) {
            isPrime[i] = true;
        }

        for (int i = 2; i <=  Math.sqrt(n); i++) {
            if (isPrime[i]) {
                for (int j = i * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }

        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                primes.add(i);
            }
        }

        return primes;
    }
}
```

4.Nth Fibonacci Number

Given a non-negative integer **n**, your task is to find the **nth Fibonacci number**.

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

The Fibonacci sequence is defined as follows:

- F(0) = 0
- F(1) = 1
- F(n) = F(n - 1) + F(n - 2) for n > 1

Examples :

```
Input: n = 5
Output: 5
Explanation: The 5th Fibonacci number is 5.
```

```
Input: n = 0
Output: 0
Explanation: The 0th Fibonacci number is 0.
```

```
Input: n = 1
Output: 1
Explanation: The 1st Fibonacci number is 1.
```

```java
class Solution {
    public int nthFibonacci(int n) {
        // code here
        if(n==0 )
            return 0;
        if(n==1)
            return 1;
        int ans=1;
        int b = 0;
        for(int i=0;i<n-1;i++)
        {
            int temp = ans;
            ans = ans+b;
            b = temp;

        }
        return ans;
    }
}
```

4.Euler Totient Function

Find the **Euler Totient Function (ETF)** Φ(N) for an input N. ETF is the count of numbers in {1, 2, 3, …, N} that are relatively prime to N, i.e., the numbers whose GCD (Greatest Common Divisor) with N is 1.

Example 1:

```
Input:
N = 11
Output:
10
Explanation:
From 1 to 11,
1,2,3,4,5,6,7,8,9,10
are relatively prime to 11.
```

Example 2:

```
Input:
N = 16
Output:
8
Explanation:
From 1 to 16
1,3,5,7,9,11,13,15
are relatively prime
to 16.
```

```java
// User function Template for Java
class Solution {
    static long ETF(long N) {
        // code here
        long result = N;
        for (long p = 2; p * p <= N; ++p) {
            if (N % p == 0) {
                while (N % p == 0)
                    N /= p;
                result -= result / p;
            }
        }
        if (N > 1)
            result -= result / N;
        return result;
    }
}
```

Learn About Catalan Number(Imp)

https://www.geeksforgeeks.org/catalan-numbers/

3.Learn About Modular Arithmetic

https://www.geeksforgeeks.org/modular-arithmetic/

Learn About Prime Factorisation

https://www.geeksforgeeks.org/prime-factorization/

Learn About Chinese remainder theorem

https://www.geeksforgeeks.org/chinese-remainder-theorem/

Bit Manipulation

1.Count set bit in an integer(No. of 1 bit)

Given a positive integer **n**. Your task is to return the **count** of set bits.

Examples:

Input: n = 6
Output: 2
Explanation: Binary representation is '110', so the count of the set bit is 2.

Input: n = 8
Output: 1
Explanation: Binary representation is '1000', so the count of the set bit is 1.

Input: n = 3
Output: 2

```java
// User function Template for Java
class Solution {
    static int setBits(int n) {
        // code here
        int countN=Integer.bitCount(n);
        return countN;
    }
}
```

2.Bit difference

You are given two numbers **a** and **b**. The task is to count the number of bits needed to be flipped to convert a to b.

Examples:

Input: a = 10, b = 20
Output: 4
Explanation:
a = 01010
b = 10100
As we can see, the bits of A that need to be flipped are **0101**0. If we flip these bits, we get 10100, which is B.

Input: a = 20, b = 25
Output: 3
Explanation:
a = 10100
b = 11001
As we can see, the bits of A that need to be flipped are 1**01**0**0**. If we flip these bits, we get 11001, which is B.

Constraints:

```java
class Solution {
    public static int countBitsFlip(int a, int b) {
        // code here
            return Integer.bitCount(a ^ b);
    }
}


//} Driver Code Ends
```

3.Count total Set bits

You are given a number **n**. Find the total count of set bits for all numbers from 1 to n (both inclusive).

Examples :

Input: n = 4
Output: 5
Explanation: For numbers from 1 to 4. For 1: 0 0 1 = 1 set bits For 2: 0 1 0 = 1 set bits For 3: 0 1 1 = 2 set bits For 4: 1 0 0 = 1 set bits Therefore, the total set bits is 5.

Input: n = 17
Output: 35
Explanation: From numbers 1 to 17(both inclusive), the total number of set bits is 35.

Expected Time Complexity: O(logn)

Expected Auxiliary Space: O(1)

Constraints:
$1 \leq n \leq 10^8$

```java
class Solution {

public static int countSetBits(int n){

        if(n==0) return 0;
         // Sum = x * 2^(x-1) + (n-2^x+1) + countSetBits(n-2^x)

        int x = calculateX(n);
        int countBitsTillTwoX = x * (1<<(x-1) ); // x * (2 to power x-1)
        int msb2xtoN = n - (1<<x) + 1;
        int rest = n - (1<<x); // solve for 3 for n = 11 ( 11 - 2 power 3)

        return countBitsTillTwoX + msb2xtoN + countSetBits(rest);

    }

    public static int calculateX(int n){
        int x = 0;
        while( (1<<x) <= n ){ //  2 raise to power x is less than or equal to n
            x++;
        }
        return x-1;
    }
}
```

4.Find Position of set bit

Given a number **n** having only one '1' and all other '0's in its binary representation, find the position of the only set bit. If there are 0 or more than 1 set bit the answer should be -1. The position of set bit '1' should be counted starting with 1 from the LSB side in the binary representation of the number.

Examples:

Input: n = 2
Output: 2
Explanation: 2 is represented as "10" in Binary. As we see there's only one set bit and it's in position 2.

Input: n = 5
Output: -1
Explanation: 5 is represented as "101" in Binary. As we see there's two set bits and thus the output -1.

Constraints:
0 <= n <= 10^8

```java
// User function Template for Java

class Solution {
    static int findPosition(int n) {
        // code here
        if(n<=0 ||( n&(n-1))!=0){
            return -1;
        }
        int pos=1;
        while(n > 1){
            n >>=1;
            pos++;
        }
        return pos;
    }
};
```

5.Copy set bit in range

Given two numbers **X** and **Y**, and a range [**L**, **R**] where 1 <= L <= R <= 32. You have to copy the set bits of **'Y'** in the range L to R in **'X'**. Return this modified X.

Note: Range count will be from Right to Left & start from 1.

Example 1:

```
Input:
X = 44, Y = 3
L = 1,  R = 5
Output:
47
Explaination:
Binary represenation of 44 and 3 is 101100 and 000011. So in the range 1 to 5 there are
two set bits of 3 (1st & 2nd position). If those are set in 44 it will become 101111 which
is 47.
```

Example 2:

```
Input:
X = 16, Y = 2
L = 1,  R = 3
Output: 18
Explaination: Binary represenation of 16 and 2 is 10000 and 10. If the mentioned
conditions are applied then 16 will become 10010 which is 18.
```

```java
class Solution {
    static int setSetBit(int x, int y, int l, int r) {
        // code here
         for(int i = l ; i<=r;i++){
            if((y&(1<<(i-1)))!=0){//Check
                x = x|(1<<(i-1));//Set
            }
        }
        return x;
    }
}
```

6.Power Set

Given a string **s** of length **n**, find all the **possible non-empty** subsequences of the string **s** in **lexicographically-sorted** order.

Example 1:

```
Input :
s = "abc"
Output:
a ab abc ac b bc c
Explanation :
There are a total 7 number of subsequences possible for the given string, and they are
mentioned above in lexicographically sorted order.
```

Example 2:

```
Input:
s = "aa"
Output:
a a aa
Explanation :
There are a total 3 number of subsequences possible for the given string, and they are
mentioned above in lexicographically sorted order.
```

```java
class Solution {
    public static void solve (String p,int index,  List<String> ans, String s){

      if(index==s.length()){
          if(p.length()>0){
              ans.add(p);
          }
          return;
      }

        solve(p,index+1,ans,s);
        p+=s.charAt(index);
        solve(p,index+1,ans,s);
    }

    public List<String> AllPossibleStrings(String s) {
        // Code here

         List<String> ans= new  ArrayList<String> ();
         String p ="";
         solve(p,0,ans,s) ;
         Collections.sort(ans);
         return ans;

    }
}
```