# **Sieve of Eratosthenes** Intuition:- The **Sieve of Eratosthenes** is an efficient algorithm for finding all prime numbers up to a given limit. The intuition behind the algorithm is based on the idea that if a number is prime, then all of its multiples are not prime. By systematically marking the multiples of each prime starting from the smallest, the algorithm "sifts out" the non-prime numbers, leaving only the primes. **Key Points:**

* **Efficiency**: The algorithm avoids checking each number against every other number. Instead, it uses the properties of primes and multiples to eliminate large groups of numbers at once.
* **Intuition**: The core idea is that composite numbers are "built" by multiplying primes, so by marking multiples, we remove those composite numbers, revealing the primes.

Code:  
  
import java.util.\*;

public class Main{

public static void main(String args[])

{

Scanner sc = new Scanner(System.in);

System.out.println("Enter a number");

int num = sc.nextInt();

boolean[] bool = new boolean[num];

for (int i = 0; i< bool.length; i++)

{

bool[i] = true;

}

for (int i = 2; i<=Math.sqrt(num);i++)

{

if(bool[i] == true)

{

for(int j = (i\*i); j<num; j = j+i)

{

bool[j] = false;

}

}

}

System.out.println("List of prime numbers up to given number are : ");

for (int i = 2; i<bool.length; i++)

{

if(bool[i]==true)

{

System.out.println(i);

}

}

}

}

# **Segmented Sieve** Intuition:-

The **Segmented Sieve** is an extension of the Sieve of Eratosthenes, designed to find all prime numbers within a specific range, particularly when the range is large. Instead of generating all primes up to a certain limit in one go, the segmented sieve divides the range into smaller segments and processes each segment individually. This approach is particularly useful when the range is too large to fit into memory at once.

### **Key Points:**

* **Memory Management**: By breaking down the problem, you manage memory more effectively, processing one segment at a time.
* **Parallel Processing**: The segmented sieve can be parallelized, with each segment processed independently, making it suitable for large-scale computations.
* **Generalization**: The segmented sieve can handle very large ranges, making it a powerful tool when you don't need all primes up to a limit, but only those within a specific range.

Code:  
  
import java.util.\*;

class Main {

static int N = 100000;

static boolean arr[] = new boolean[N+1];

static void simpleSieve()

{

for(int i=2;i<=N;i++)

{

arr[i] = true;

}

for(int i=2;i<Math.sqrt(N);i++)

{

if(arr[i] == true)

{

for(int j=i\*i; j<=N; j=j+i)

{

arr[j] = false;

}

}

}

}

static ArrayList<Integer> generatePrimes(int n)

{

ArrayList<Integer> al = new ArrayList();

for(int i=2;i<Math.sqrt(n);i++)

{

if(arr[i] == true)

{

al.add(i);

}

}

return al;

}

public static void main (String[] args)

{

Scanner sc = new Scanner(System.in);

int low = sc.nextInt(); //80

int high = sc.nextInt(); //90

simpleSieve();

ArrayList<Integer> al = generatePrimes(high);

boolean dummy[] = new boolean[high-low+1];

for(int i=0;i<high-low+1;i++)

{

dummy[i] = true;

}

for(int prime: al)

{

int firstMultiple = (low/prime) \* prime;

if(firstMultiple < low)

{

firstMultiple = firstMultiple + prime;

}

int start = Math.max(firstMultiple, prime\*prime);

for(int j=start; j<=high; j+=prime)

{

dummy[j-low] = false;

}

}

for(int i=low;i<=high;i++)

{

if(dummy[i-low] == true)

{

System.out.print(i + " ");

}

}

}

}

# **Incremental Sieve** Intuition:-

The Incremental Sieve is a variation of the traditional Sieve of Eratosthenes, optimized to generate prime numbers incrementally by focusing on odd numbers. Instead of processing all numbers up to a given limit, the incremental sieve iteratively identifies and eliminates non-prime numbers by marking multiples of each discovered prime. This approach is particularly effective when you want to efficiently find primes up to a small or moderate limit without generating a full sieve up to the square root of the limit.  
  
**Key Points:**

* **Odd Number Optimization**:The incremental sieve skips even numbers, processing only odd numbers starting from 3. This reduces the computation by approximately half, compared to a standard sieve that processes every number.
* **Prime Identification and Marking**:The algorithm begins by adding 2 as the first prime, then iterates over the list of odd numbers. For each odd number that hasn't been marked as non-prime (i.e., not set to -1), it identifies it as a prime and then marks its multiples as non-prime within the same list.
* **Incremental Approach**:As the algorithm processes each odd number, it marks multiples of identified primes, incrementally eliminating non-primes. This ensures that unmarked numbers in the list are primes, allowing the sieve to build up the list of primes step by step.
* **Efficiency for Small Ranges**:The incremental sieve is particularly efficient for generating primes in small or moderate ranges, where its memory usage and processing speed are optimized by focusing only on odd numbers and incrementally eliminating non-primes.
* **Scalability**:While effective for smaller ranges, the incremental sieve is less suited for very large ranges compared to more advanced sieves (e.g., segmented sieve) due to its need to revisit the entire list of numbers to mark multiples. However, its simplicity and focus on odd numbers make it a good choice for less demanding applications.

Code:  
  
import java.util.\*;

public class Main{

public static List < Integer > incrementalSieve (int limit)

{

List < Integer > oddNumber = new ArrayList <> ();

for (int i = 3; i <= limit; i += 2)

{

oddNumber.add (i);

}

List < Integer > primes = new ArrayList <> ();

primes.add (2);

for (int i = 0; i < oddNumber.size (); i++)

{

int current = oddNumber.get (i);

if (current != -1)

{

primes.add (current);

for (int j = i; j < oddNumber.size (); j++)

{

if (oddNumber.get (j) % current == 0)

{

oddNumber.set (j, -1);

}

}

}

}

return primes;

}

public static void main (String[]args)

{

Scanner sc=new Scanner(System.in);

int n = sc.nextInt();

List < Integer > primes = incrementalSieve (n);

System.out.println ("Prime numbers up to " + n + ": " + primes);

}

}

# **Euler’s Phi** Intuition:-

Euler's Totient Function, often denoted as ϕ(n)\phi(n)ϕ(n), is a fundamental function in number theory that counts the number of positive integers up to n that are coprime with n. Two numbers are coprime if their greatest common divisor (GCD) is 1. The function is especially useful in various fields like cryptography (e.g., RSA algorithm), modular arithmetic, and more.

The core idea is that ϕ(n)\phi(n)ϕ(n) tells you how many integers are "relatively prime" to n, which means they do not share any common factors with n other than 1.

### **Key Points:**

* **Prime Numbers**:If n is a prime number, then all integers less than n are coprime with n. Therefore, ϕ(p)=p−1\phi(p) = p - 1ϕ(p)=p−1 for any prime ppp.
* **Multiplicative Property:**Euler's Totient function is multiplicative for coprime numbers, meaning if mmm and n are coprime, then ϕ(m×n)=ϕ(m)×ϕ(n)\phi(m \times n) = \phi(m) \times \phi(n)ϕ(m×n)=ϕ(m)×ϕ(n).
* **Formula for ϕ(n)\phi(n)ϕ(n)**:For any integer nnn, the value of ϕ(n)\phi(n)ϕ(n) can be calculated using the formula: ϕ(n)=n×(1−1p1)×(1−1p2)×⋯×(1−1pk)\phi(n) = n \times \left(1 - \frac{1}{p\_1}\right) \times \left(1 - \frac{1}{p\_2}\right) \times \dots \times \left(1 - \frac{1}{p\_k}\right)ϕ(n)=n×(1−p1​1​)×(1−p2​1​)×⋯×(1−pk​1​) where p1,p2,…,pkp\_1, p\_2, \dots, p\_kp1​,p2​,…,pk​ are the distinct prime factors of nnn.
* **Reduction of Non-Coprime Numbers**:The idea behind the formula is that for each prime factor pip\_ipi​ of n, numbers that are multiples of pip\_ipi​ are not coprime with n. The function reduces the count by eliminating these non-coprime numbers.
* **Applications:**
  + ϕ(n)\phi(n)ϕ(n) is crucial in the RSA encryption algorithm, where the security relies on the difficulty of determining ϕ(n)\phi(n)ϕ(n) for large composite numbers.
  + It's also used in problems involving modular inverses, where ϕ(n)\phi(n)ϕ(n) helps determine the existence of an inverse modulo n.

Code:

import java.util.\*;

public class Main

{

static int phi(int n)

{

int result = n;

for (int 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;

}

public static void main (String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

System.out.println(phi(n));

}

}

# Strobogrammatic Number

### Intuition:-

A **strobogrammatic number** is a number that looks the same when rotated 180 degrees (turned upside down). These numbers are symmetric with respect to a central axis, meaning that after the rotation, the number appears identical to its original form. Strobogrammatic numbers are interesting in both number theory and computer science, particularly in problems related to symmetry and palindromes.

### **Key Points:**

* **Symmetric Digits:**The key to understanding strobogrammatic numbers lies in recognizing which digits look the same when rotated 180 degrees. The strobogrammatic pairs are:
  + - 0 ↔ 0
    - 1 ↔ 1
    - 6 ↔ 9
    - 8 ↔ 8
    - 9 ↔ 6
* **Central Symmetry:**For a number to be strobogrammatic, each digit must have a corresponding symmetric pair at the opposite end of the number. For example, in the number 69, 6 becomes 9 and vice versa when rotated.
* **Odd and Even Lengths:**For even-length strobogrammatic numbers, the entire number is made up of symmetric digit pairs. For odd-length numbers, the middle digit must be one of the digits that looks the same when rotated (0, 1, 8), while the remaining digits must form symmetric pairs.
* **Examples**:
  + Even Length: 69, 96, 88, 11, 1001
  + Odd Length: 818, 101, 609
* **Non-Strobogrammatic Digits:**Digits like 2, 3, 4, 5, and 7 do not have corresponding symmetric counterparts and cannot be part of a strobogrammatic number. Including these digits in any position will make the number non-strobogrammatic.
* **Applications:**Strobogrammatic numbers are used in various mathematical puzzles and can also appear in problems related to digital displays, where numbers need to be readable from different orientations.

Code:

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

System.out.println("Enter a number: ");

String num = sc.nextLine();

if(isStrobogrammatic(num))

{

System.out.println(num + " is a strobogrammatic number");

}

else

{

System.out.println(num + " is not a strobogrammatic number");

}

sc.close();

}

public static boolean isStrobogrammatic(String num)

{

Map<Character, Character> strobogrammaticDictonary = new HashMap<>();

strobogrammaticDictonary.put('0', '0');

strobogrammaticDictonary.put('1', '1');

strobogrammaticDictonary.put('6', '9');

strobogrammaticDictonary.put('8', '8');

strobogrammaticDictonary.put('9', '6');

int n = num.length();

for(int i = 0 , j = (n-1) ; i <= j ; i++, j--)

{

char digit\_left = num.charAt(i);

char digit\_right = num.charAt(j);

char mapping = strobogrammaticDictonary.getOrDefault(digit\_left, '-');

if(mapping == '-')

{

return false;

}

if(mapping != digit\_right)

{

return false;

}

}

return true;

}

}

# **Chinese Remainder Theorem** Intuition:-

The **Chinese Remainder Theorem** (CRT) is a powerful theorem in number theory that provides a solution to systems of simultaneous congruences with different moduli. It allows you to determine a unique solution modulo the product of the moduli when certain conditions are met. The CRT is particularly useful in cryptography, coding theory, and solving complex modular arithmetic problems.

### **Key Points:**

* **Simultaneous Congruences:**
  + The CRT addresses problems where you need to find an integer x that satisfies multiple congruences simultaneously, such as: x≡a1 (mod m1)x \equiv a\_1 \ (\text{mod} \ m\_1)x≡a1​ (mod m1​) x≡a2 (mod m2)x \equiv a\_2 \ (\text{mod} \ m\_2)x≡a2​ (mod m2​) …\dots… x≡ak (mod mk)x \equiv a\_k \ (\text{mod} \ m\_k)x≡ak​ (mod mk​)
  + Here, m1,m2,…,mkm\_1, m\_2, \dots, m\_km1​,m2​,…,mk​ are the moduli, and a1,a2,…,aka\_1, a\_2, \dots, a\_ka1​,a2​,…,ak​ are the remainders.
* **Coprime Moduli:**
  + The theorem requires that the moduli m1,m2,…,mkm\_1, m\_2, \dots, m\_km1​,m2​,…,mk​ be pairwise coprime, meaning the greatest common divisor (GCD) of any pair of moduli is 1 (i.e., gcd(mi,mj)=1\text{gcd}(m\_i, m\_j) = 1gcd(mi​,mj​)=1 for i≠ji \neq ji=j).
  + When this condition is met, there exists a unique solution xxx modulo M=m1×m2×⋯×mkM = m\_1 \times m\_2 \times \dots \times m\_kM=m1​×m2​×⋯×mk​.
* **Constructive Solution:**
  + The CRT not only guarantees the existence of a solution but also provides a method to construct it:
    1. Compute the product M=m1×m2×⋯×mkM = m\_1 \times m\_2 \times \dots \times m\_kM=m1​×m2​×⋯×mk​.
    2. For each congruence, compute Mi=MmiM\_i = \frac{M}{m\_i}Mi​=mi​M​.
    3. Find the modular inverse of MiM\_iMi​ modulo mim\_imi​, denoted as yiy\_iyi​, such that Mi×yi≡1 (mod mi)M\_i \times y\_i \equiv 1 \ (\text{mod} \ m\_i)Mi​×yi​≡1 (mod mi​).
    4. The solution xxx is given by: x=∑i=1kai×Mi×yi (mod M)x = \sum\_{i=1}^k a\_i \times M\_i \times y\_i \ (\text{mod} \ M)x=i=1∑k​ai​×Mi​×yi​ (mod M)
* **Uniqueness**:
  + The solution xxx is unique modulo MMM. This means that any other solution will be congruent to xxx modulo MMM, so xxx is the smallest positive integer that satisfies all the given congruences.
* **Applications**:
  + The CRT is widely used in computational number theory, especially in algorithms dealing with large numbers, such as RSA encryption.
  + It also simplifies complex modular arithmetic by breaking it down into smaller, more manageable pieces.

Code:

import java.util.\*;

public class Main

{

static int CRT(int a[], int m[], int n, int p)

{

int x = 0;

for (int i = 0; i < n; i++)

{

int M = p / m[i];

int y = 0;

for (int j = 0; j < m[i]; j++)

{

if ((M \* j) % m[i] == 1)

{

y = j;

break;

}

}

x = x + a[i] \* M \* y;

}

return x % p;

}

public static void main(String args[])

{

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of congruence relations: ");

int size = sc.nextInt();

int a[] = new int[size];

System.out.println("Enter the values of a (remainders): ");

for (int i = 0; i < size; i++)

{

a[i] = sc.nextInt();

}

int m[] = new int[size];

int p = 1;

System.out.println("Enter the values of m (moduli): ");

for (int i = 0; i < size; i++)

{

m[i] = sc.nextInt();

p = p \* m[i];

}

System.out.println("The solution is " + CRT(a, m, size, p));

}

}

# **Binary Palindrome**

### Intuition:-

A **Binary Palindrome** is a binary number that reads the same forwards and backwards. This property is similar to a palindromic number in base 10 but applies to binary digits (0s and 1s). Binary palindromes are particularly interesting in computer science and digital systems, where binary representation is fundamental. The concept of symmetry is central to understanding binary palindromes.

### **Key Points:**

* **Symmetric Structure:**A binary palindrome is symmetric about its center. This means that for a binary number to be a palindrome, the sequence of digits must be the same when read from the leftmost bit to the rightmost bit and vice versa.
* **Odd and Even Lengths:**
  + If the binary number has an odd number of digits, the middle digit doesn't need to be paired with another digit, as it sits alone in the center.
  + For binary numbers with an even number of digits, each digit on the left side must have a corresponding matching digit on the right side.
* **Checking for a Palindrome:**
  + Convert the number to its binary representation.
  + Compare the binary string with its reverse. If they are identical, the number is a binary palindrome.
  + Alternatively, you can check bit by bit from the outermost to the innermost digits using bitwise operations.
* **Examples**:
  + 101 (binary for 5) is a binary palindrome because it reads the same forwards and backwards.
  + 1001 (binary for 9) is a binary palindrome because the first and last digits are 1, and the middle digits are 0, making it symmetric.
* **Applications**:
  + Binary palindromes are used in various fields of computer science, including error detection, cryptography, and digital signal processing.
  + They can be useful in designing algorithms where symmetry and patterns in binary data are important, such as in palindromic sequences or pattern recognition tasks.

Code:

import java.util.\*;

public class Main

{

public static boolean isBinaryPalindrome(int num)

{

int revBinary = 0;

int copyNum = num;

while (copyNum != 0)

{

revBinary = (revBinary << 1) | (copyNum & 1);

copyNum >>= 1;

}

return revBinary == num;

}

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int num=sc.nextInt();

System.out.println(isBinaryPalindrome(num));

}

}

# **Booth’s Algorithm**

### Intuition:-

Booth's Algorithm is a multiplication algorithm that efficiently multiplies two signed binary numbers using two's complement notation. The algorithm is particularly useful because it minimizes the number of arithmetic operations required by reducing the number of additions and subtractions during the multiplication process. Booth's Algorithm works by encoding the multiplier in a way that reduces the complexity of the multiplication operation.

### **Key Points:**

* **Two's Complement Representation:**Booth's Algorithm operates on signed binary numbers represented in two's complement form, which allows for straightforward handling of both positive and negative numbers.
* **Encoding the Multiplier:**
  + The key idea behind Booth's Algorithm is to analyze pairs of bits in the multiplier to determine whether to add, subtract, or do nothing during each step of the multiplication.
  + Specifically, the algorithm looks at the current bit and the previous bit of the multiplier (including a "phantom" bit initially set to 0) to decide the operation:
    - 01: Add the multiplicand.
    - 10: Subtract the multiplicand.
    - 00 or 11: No operation (continue shifting).
* **Efficient Multiplication:**Booth's Algorithm can efficiently handle sequences of 0s and 1s in the multiplier, reducing the number of additions or subtractions needed. For example, a sequence of 0s means the multiplicand doesn't need to be added at each step, while a sequence of 1s is handled in a single subtraction operation.
* **Shifting and Arithmetic Operations:**The algorithm involves shifting the bits of the multiplicand and the partial product to the right after each step. This shifting simulates the multiplication by powers of two (binary shifts), with the addition or subtraction of the multiplicand adjusting the partial product as necessary.
* **Example**:
  + Suppose we want to multiply 3 (binary 0011) by -4 (binary 1100 in two's complement).
  + Booth's Algorithm will encode the multiplier -4 and perform a series of shifts and additions/subtractions based on the bit pairs of the multiplier, resulting in the final product in two's complement form.
* **Handling Negative Numbers:**By using two's complement and Booth's encoding, the algorithm naturally handles both positive and negative multiplicands and multipliers, producing the correct signed product.
* **Applications**:
  + Booth's Algorithm is used in computer arithmetic units, especially in processors where hardware resources are limited, and efficient multiplication is required.
  + It is particularly advantageous in situations where the multiplier has long sequences of 0s or 1s, which the algorithm can efficiently compress into fewer operations.

Code:

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int a = sc.nextInt();

int b = sc.nextInt();

int product = 0;

int n = Integer.toBinaryString(a).length();

for (int i = 0; i < n; i++)

{

int currentBit = (a & 1);

if (currentBit == 1)

{

product += b;

}

b <<= 1;

a >>= 1;

}

System.out.println("Result: " + product);

}

}