SIMPLE SIEVE:

The Simple Sieve is a basic algorithm for finding all prime numbers up to a given limit. Here are the steps:

1. Create a list of consecutive integers from 2 through the limit.

2. Set a variable p to 2, the smallest prime number.

3. Cross out all multiples of p from the list, starting from p^2.

4. Find the next number in the list that is not crossed out. This is the next prime number.

5. Set p to the next prime number found in step 4 and repeat steps 3-5 until p^2 is greater than the limit.

6. The remaining numbers in the list are all prime numbers.

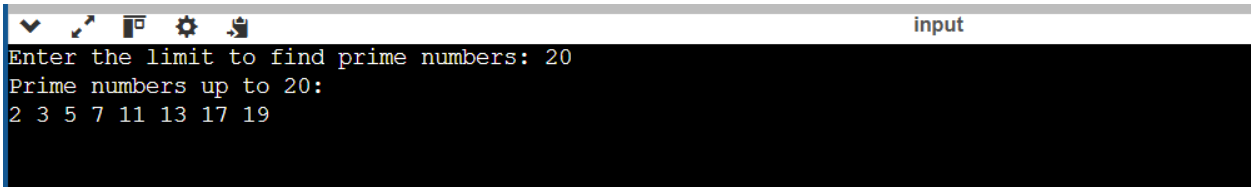
CODE:

```
import java.util.Scanner;


public class SieveMain {
    public static void simpleSieve(int limit) {
        boolean[] prime = new boolean[limit + 1];
        for (int i = 2; i <= limit; i++) {
            prime[i] = true;
        }

        // Mark all the multiples of the prime numbers
        for (int p = 2; p * p <= limit; p++) {
            // If prime[p] is not changed, then it is a prime
            if (prime[p]) {
                // Update all multiples of p
                for (int i = p * p; i <= limit; i += p) {
                    prime[i] = false;
                }
```

```
            }
        }

        // Print all prime numbers
        System.out.println("Prime numbers up to " + limit + ":");
        for (int p = 2; p <= limit; p++) {
            if (prime[p]) {
                System.out.print(p + " ");
            }
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the limit to find prime numbers: ");
        int limit = scanner.nextInt();

        // Validate input
        if (limit >= 2) {
            simpleSieve(limit);
        } else {
            System.out.println("Please enter a number greater than or equal to 2.");
        }

        scanner.close();
    }
```

}

OUTPUT:



Time complexity: O(nlog(logn))

Space complexity:O(n)

SEGMENTED SIEVE:

The Segmented Sieve and Incremental Sieve are two algorithms used for finding prime numbers in a certain range.

The Segmented Sieve algorithm is used to find all prime numbers within a given range `[L, R]` where `L` and `R` are two non-negative integers such that `L <= R`. The basic idea behind this algorithm is to use the Sieve of Eratosthenes method to find all prime numbers up to the square root of `R`, and then use these primes to eliminate all composite numbers in the range `[L, R]`.

The Incremental Sieve algorithm, on the other hand, is used to find all prime numbers starting from a given number `N`. The basic idea behind this algorithm is to use the Sieve of Eratosthenes method to find all prime numbers up to a certain limit, say `L`, and then use these primes to check if `N + i` is a prime number for `i = 0, 1, 2, ...`.

Both of these algorithms are efficient for finding prime numbers in certain ranges. The Segmented Sieve is particularly useful for finding primes in large ranges, while the Incremental Sieve is useful for finding primes starting from a given number.

```
import java.util.Scanner;

public class Segmented {
```

```
    static void SegSieve(int l, int h) {
        // Step 1: Create a boolean array to mark numbers as prime or not
        boolean[] prime = new boolean[h + 1];

        // Step 2: Segmented Sieve algorithm
        for (int p = 2; p * p <= h; p++) {
            // Find the smallest multiple of p in the range [l, h]
            int sm = (l / p) * p;
            if (sm < l) {
                sm += p;  // Make sure sm is in the range [l, h]
            }

            // Mark all multiples of p as non-prime
            for (int i = sm; i <= h; i += p) {
                prime[i] = true;
            }
        }

        // Step 3: Print all prime numbers in the range [l, h]
        for (int i = l; i <= h; i++) {
            if (!prime[i] && i > 1) {  // Prime numbers are those that are not marked as true, and not
1
                System.out.print(i + " ");y
            }
        }
        System.out.println();  // Move to a new line after printing primes
    }

    public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);

        // Get user input for the lower and upper bounds of the range
        System.out.print("Enter the lower bound (l): ");
        int l = scanner.nextInt();
        System.out.print("Enter the upper bound (h): ");
        int h = scanner.nextInt();

        // Call the SegSieve function to print primes in the range [l, h]
        SegSieve(l, h);

        scanner.close();  // Close the scanner
    }
}
```

OUTPUT:

Time complexity: $O((h-l+1)\times\log(\log h))$

Space Complexity: $O(h-l+1)$

EULER'S PHI ALGORITHM:

Euler's phi algorithm is a method for computing Euler's totient function, which counts the number of positive integers up to a given integer n that are relatively prime to n.

```
import java.util.*;  // Import the Scanner class for taking user input

public class EulerPhiAlgorithm {
```

```
// This method calculates Euler's Totient Function φ(n)
public static int phi(int n) {
    int result = n;  // Start with the assumption that all numbers less than n are coprime with n

    // Loop through all numbers from 2 to sqrt(n)
    for (int p = 2; p * p <= n; p++) {

        // If p is a factor of n (i.e., p divides n)
        if (n % p == 0) {
            // Remove all occurrences of p from n
            while (n % p == 0) {
                n /= p;  // Divide n by p until it's no longer divisible
            }

            // Subtract the fraction result/p from result
            // This implements the formula: result = result * (1 - 1/p)
            result -= result / p;
        }
    }

    // If n is greater than 1 at this point,
    // it means n is a prime number greater than sqrt(original_n)
    if (n > 1) {
        result -= result / n;  // Apply the formula for this prime factor
    }

    return result;  // Return the final totient value
```

```
    }

    // Main method to execute the program
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);  // Create Scanner object to read user input

        System.out.print("Enter the value of n: ");  // Prompt user for input
        int n = sc.nextInt();  // Read the integer n

        int phi_n = phi(n);  // Call the phi() method to compute φ(n)

        System.out.println("phi(" + n + ") = " + phi_n);  // Print the result

        sc.close();  // Close the Scanner object to free resources
    }
}
```

OUTPUT:

Time complexity: O(sqrt(n))

Space complexity:O(1)

STROBOGRAMMATIC NUMBER:

A strobogrammatic number is a number that looks the same when viewed upside down or rotated 180 degrees. In other words, it is a number that remains unchanged when rotated by 180 degrees. The digits 0, 1, 8 are strobogrammatic digits, as they appear the same when viewed upside down or rotated. The digits 2 and 5 can also be considered strobogrammatic when they are rotated 180 degrees. However, the digits 3, 4, 6, and 9 are not strobogrammatic because they appear different when viewed upside down or rotated.

Strobogrammatic numbers are often used in number puzzles and games, and they can also be found in some real-world applications, such as odometers, digital clocks, and certain medical equipment.

```
import java.util.HashMap; // Import HashMap to store valid digit rotations
import java.util.Map;     // Import Map interface
import java.util.Scanner; // Import Scanner to read user input

public class Strobo {

    // Function to check if a number string is strobogrammatic
    static boolean isStrobogrammatic(String num) {
        // Create a mapping of digits to their 180-degree rotated counterparts
        Map<Character, Character> map = new HashMap<Character, Character>();
        map.put('6', '9');  // 6 becomes 9 when rotated
        map.put('9', '6');  // 9 becomes 6 when rotated
        map.put('0', '0');  // 0 remains 0
        map.put('1', '1');  // 1 remains 1
        map.put('8', '8');  // 8 remains 8
```

```
    int l = 0, r = num.length() - 1; // Two pointers: one from start (l), one from end (r)

    // Loop while left pointer is less than or equal to right pointer
    while (l <= r) {
        // If character at position l is not in the valid map, return false
        if (!map.containsKey(num.charAt(l))) return false;

        // If the mapped value of num[l] is NOT equal to num[r], it's not strobogrammatic
        if (map.get(num.charAt(l)) != num.charAt(r)) return false;

        // Move the pointers inward
        l++;
        r--;
    }

    // If all checks pass, it's strobogrammatic!
    return true;
}

// Main method to take input and test the function
public static void main(String args[]) {
    Scanner sc = new Scanner(System.in); // Create Scanner object for user input

    System.out.print("Give num :"); // Prompt the user
    String n = sc.next();          // Read the input number as a string

    // Call the function and print the result (true or false)
```

```
        System.out.println(isStrobogrammatic(n));


        sc.close(); // Close the scanner to free system resources
    }
}
```

OUTPUT:

Time complexity:O(n)

Space complexity:O(1)

CHINESE REMAINDER THEOREM:

The Chinese Remainder Theorem (CRT) is a mathematical theorem that provides a solution to a system of congruences. It allows finding a unique solution to a set of simultaneous congruences when the moduli are pairwise coprime (i.e., they have no common factors).

The CRT states that if we have a system of congruences of the form:

$x \equiv a_1 \pmod{m_1}$

$x \equiv a_2 \pmod{m_2}$

...

$x \equiv a_n \pmod{m_n}$

where m_1, m_2, ..., m_n are pairwise coprime integers and a_1, a_2, ..., a_n are any integers, then there exists a unique solution for x modulo M, where M is the product of all the moduli ($M = m_1 * m_2 * ... * m_n$). Moreover, this solution x can be represented in the form:

$x \equiv b \pmod{M}$

```
import java.util.*;
class ChineseRemainder {
   int calculate(int size , int div[], int rem[])
   {
      int j , x = 1;
      while(true)
      {
         for( j=0;j<size;j++)
         {
            if( x % div[j] != rem[j] )
            break;
         }
         if(j == size)
         return x;
         x++;} }}
class ChineseRemainderTheorem
{
   public static void main (String[] args)
   {
Scanner sc=new Scanner(System.in);
      System.out.println("Enter Divisor");
      int size = sc.nextInt();
      int [] div = new int[size];
      for(int i=0; i<size; i++)
      {
         div[i] = sc.nextInt();
```

```
        }

        System.out.println("Enter Remainder");
        int rem[]=new int[size];
        for(int i=0; i<size; i++)
        {
           rem[i] = sc.nextInt();
        }

        ChineseRemainder  c = new ChineseRemainder();
        System.out.println( c.calculate(size, div, rem));
    }  }
```

OUTPUT:

Time complexity:O(size)

Space complexity:O(1)

TOGGLE SWITCH:

Problem statement

There are 100 closed doors. A cage holding 100 monkeys is placed nearby. Every monkey that visits a door either opens a closed door or closes an open door. The first monkey that is let out of the cage visits and opens all the hundred doors. The second monkey that is released visits doors of the order 2, 4, 6, 8, 10…. . The third monkey released visits doors 3, 6, 9,12, 15……, and so on.

After all the monkeys from the cage are released and have opened or closed at least one door, how many doors are left open?

Approach

Let us understand how to solve this problem by observing the activity of doors 13, 50, and 16. 13 is a prime number, 50 is a composite number, and 16 is a perfect square.

```
import java.util.*;


public class Toggle {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();  // Read the number of switches
        boolean[] b = new boolean[n + 1];  // Array to represent the state of each switch


        // Iterate through each switch number
        for (int i = 1; i <= n; i++) {
            // Toggle all switches that are multiples of i
            for (int j = i; j <= n; j += i) {
                b[j] = !b[j];  // Toggle the state of switch j
            }
        }


        int openCount = 0;  // To count the number of switches that are open
        int closedCount = 0;  // To count the number of switches that are closed
```

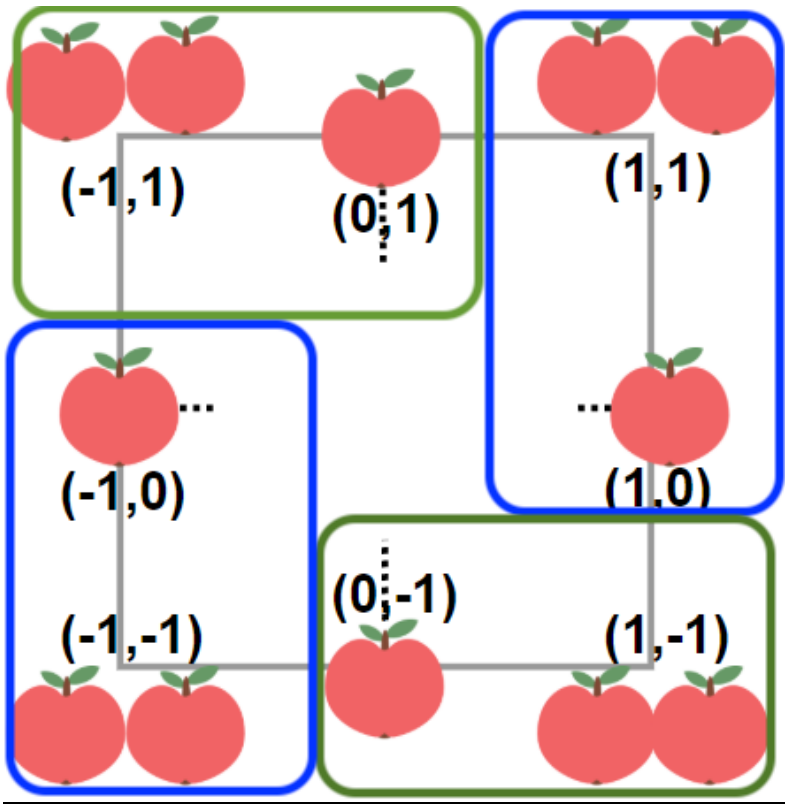
```
        // Count the number of open and closed switches
        for (int i = 1; i <= n; i++) {
            if (b[i]) {
                openCount++;
            } else {
                closedCount++;
            }
        }

        // Output the results
        System.out.println("Number of switches open: " + openCount);
        System.out.println("Number of switches closed: " + closedCount);
    }
}
```

OUTPUT:

TIME COMPLEXITY:O(n^2)

SPACE COMPLEXITY:O(n)

ALICE APPLE TREE:



m	n 12*m*m	N Sigma(n)
1	12	12
2	48	60
3	108	168
4	192	360

```
import java.util.*;
public class AliceAppleTree {
    public static void main(String[] args) {
```

```
        Scanner sc=new Scanner(System.in);
        int apple=sc.nextInt();
        int cnt=0,sum=0;
        while(sum<apple){
           cnt++;
           sum+=(12*cnt*cnt);

        }
        System.out.println((8*(cnt)));
    }
}
```

OUTPUT:

TIME COMPLEXITY:O(sqrt(apple))

SPACE COMPLEXITY:O(1)

BINARY PALINDROME:

In binary representation, a palindrome is a sequence of binary digits that reads the same forwards and backwards. It means that if you reverse the sequence, it will remain unchanged.

> For N = 16, the binary representation is 10000. As you correctly pointed out, this binary sequence is not a palindrome because if we reverse it, we get 00001, which is not the same as the original sequence.
> For N = 17, the binary representation is 10001. This binary sequence is a palindrome because if we reverse it, we still get 10001, which is the same as the original sequence.

In summary, the concept of binary palindromes involves examining whether a binary sequence remains unchanged when its digits are reversed. If it does, it is considered a binary palindrome.

CODE:

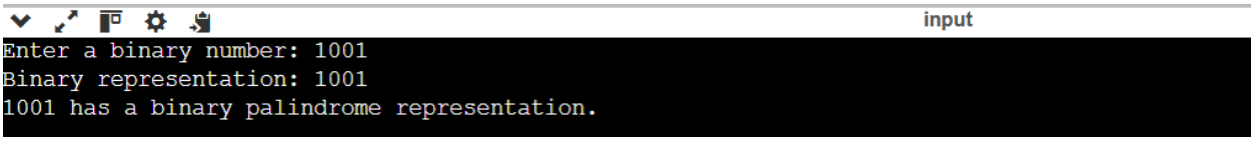
```
import java.util.*;
public class BinaryPalindromeChecker {
    public static boolean isBinaryPalindrome(int x) {
        int reversed = 0;
        int original = x;
        while (x > 0) {
            reversed <<= 1;          // Left shift by 1 bit
            reversed |= (x & 1);     // Add the least significant bit of 'x' to'reversed'
            x >>= 1;                 // Right shift by 1 bit
        }
        return reversed == original;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int x = sc.nextInt();
        if (isBinaryPalindrome(x)) {
            System.out.println(x + " has a binary palindrome representation.");
        } else {
            System.out.println(x + " does not have a binary palindrome representation.");
        }
    }
}
```

Program to get binary number:

```
import java.util.Scanner;

public class BinaryPalindromeChecker {

    public static boolean isBinaryPalindrome(int x) {
        int reversed = 0;
        int original = x;

        // Reverse the binary representation of the integer
        while (x > 0) {
            reversed <<= 1;          // Left shift by 1 bit
            reversed |= (x & 1);     // Add the least significant bit of 'x' to 'reversed'
            x >>= 1;                 // Right shift by 1 bit
        }

        // Check if the reversed binary is the same as the original
        return reversed == original;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Get binary input as a string from the user
        System.out.print("Enter a binary number: ");
        String binaryInput = sc.nextLine();
```

```
        // Convert binary string to integer
        int x = Integer.parseInt(binaryInput, 2); // Converts binary string to integer

        // Display the binary representation
        System.out.println("Binary representation: " + binaryInput);

        // Check if the binary number is a palindrome
        if (isBinaryPalindrome(x)) {
            System.out.println(binaryInput + " has a binary palindrome representation.");
        } else {
            System.out.println(binaryInput + " does not have a binary palindrome representation.");
        }

        sc.close(); // Close the scanner to avoid resource leak
    }
}
```

OUTPUT:



Time complexity:O(log n)

Space complexity:O(1)