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.

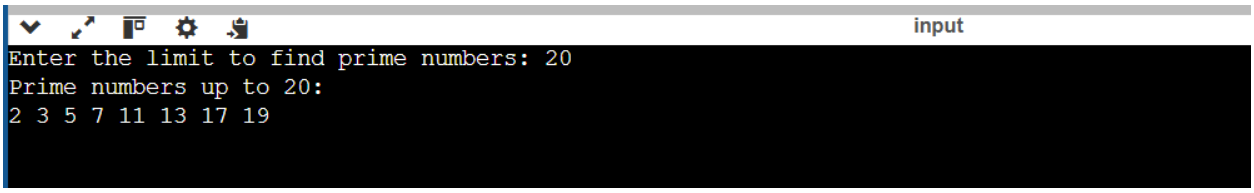
CODE:

```
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 + " ");
            }
        }
        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.*;
public class EulerPhiAlgorithm {

  // Returns the value of Euler's totient function phi(n)
  public static int phi(int n) {
    int result = n;  // Initialize result as n

    // Check for all prime factors of n and subtract their multiples from result
    for (int p = 2; p * p <= n; p++) {
      if (n % p == 0) {  // p is a prime factor of n
        while (n % p == 0) {  // Remove all multiples of p from n
          n /= p;
        }
        result -= result / p;
      }
    }
    // If n has a prime factor greater than sqrt(n), then add its contribution
    if (n > 1) {
      result -= result / n;
    }
    return result;
  }
  // Main method to test the program
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the value of n: ");
        int n = sc.nextInt();
        int phi_n = phi(n);
        System.out.println("phi(" + n + ") = " + phi_n);
        sc.close();
    }
}
```

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 java.util.Map;
import java.util.Scanner;
public class Strobo {
     static boolean isStrobogrammatic(String num) {
    Map<Character, Character> map = new HashMap<Character, Character>();
     map.put('6', '9');
     map.put('9', '6');
     map.put('0', '0');
     map.put('1', '1');
     map.put('8', '8');
     int l = 0, r = num.length() - 1;
     while (l <= r) {
         if (!map.containsKey(num.charAt(l))) return false;
         if (map.get(num.charAt(l)) != num.charAt(r))
             return false;
         l++;
         r--;
     }
     return true;}
public static void main(String args[])
{
```

```
    Scanner sc =new Scanner(System.in);
   System.out.print("Give num :");
   String n= sc.next();
    System.out.println(isStrobogrammatic(n));


 }
}
```

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
            }
        }
```

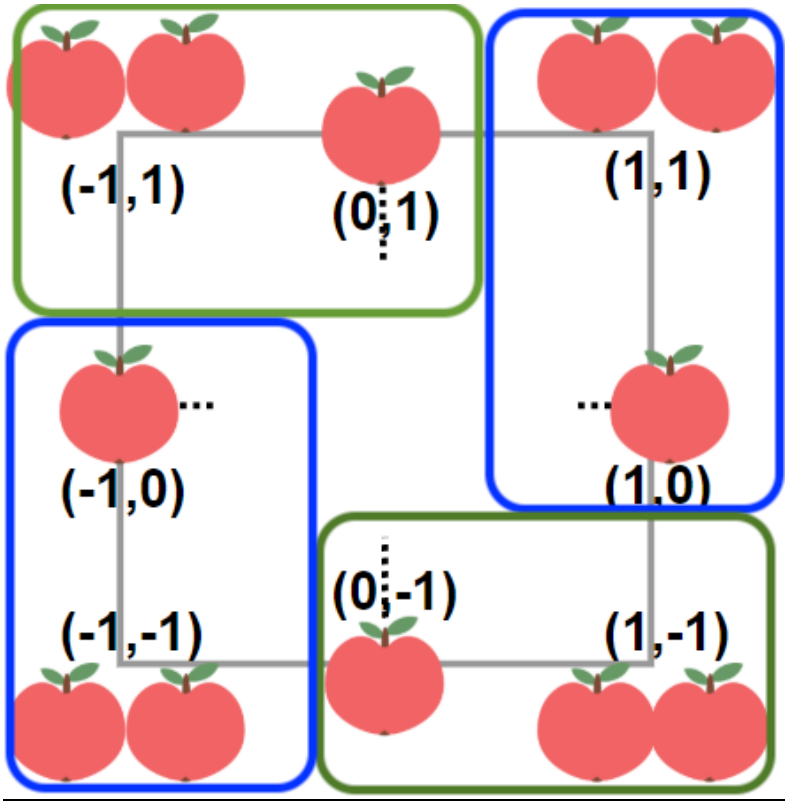
```
        int openCount = 0;  // To count the number of switches that are open
        int closedCount = 0;  // To count the number of switches that are closed

        // 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);
    }
}
```

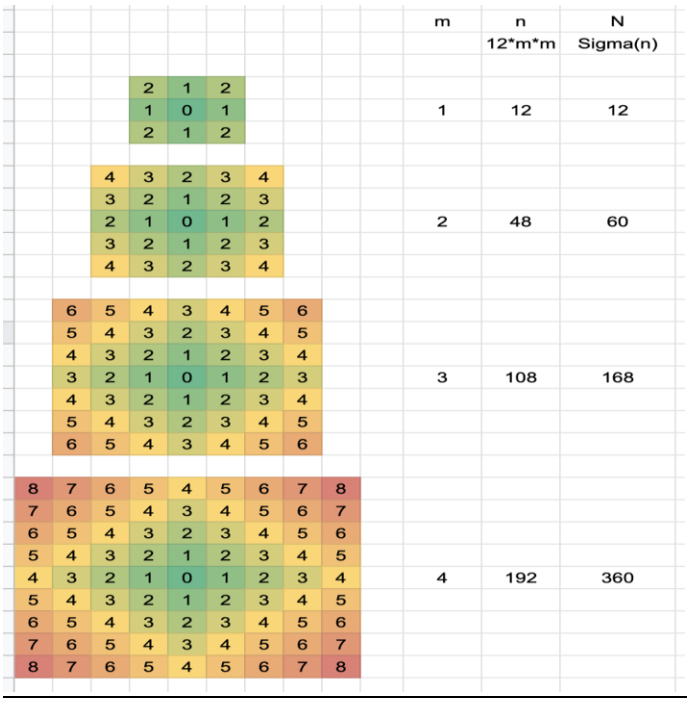
OUTPUT:

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:

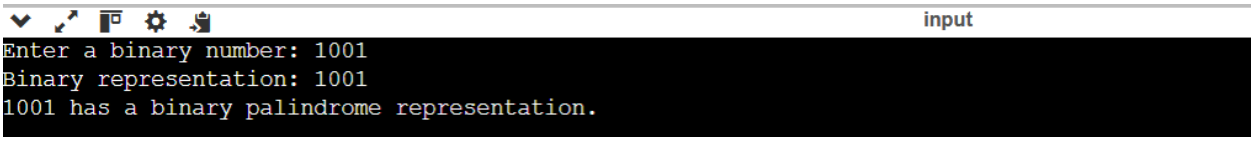
```
import java.util.*;
public class BinaryPalindromeChecker {
    public static boolean isBinaryPalindrome(int x) {
        int reversed = 0;
        int original = x;
        while (x > 0) {
            reversed <<= 1;          // Left shift by 1 bit
            reversed |= (x & 1);     // Add the 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)

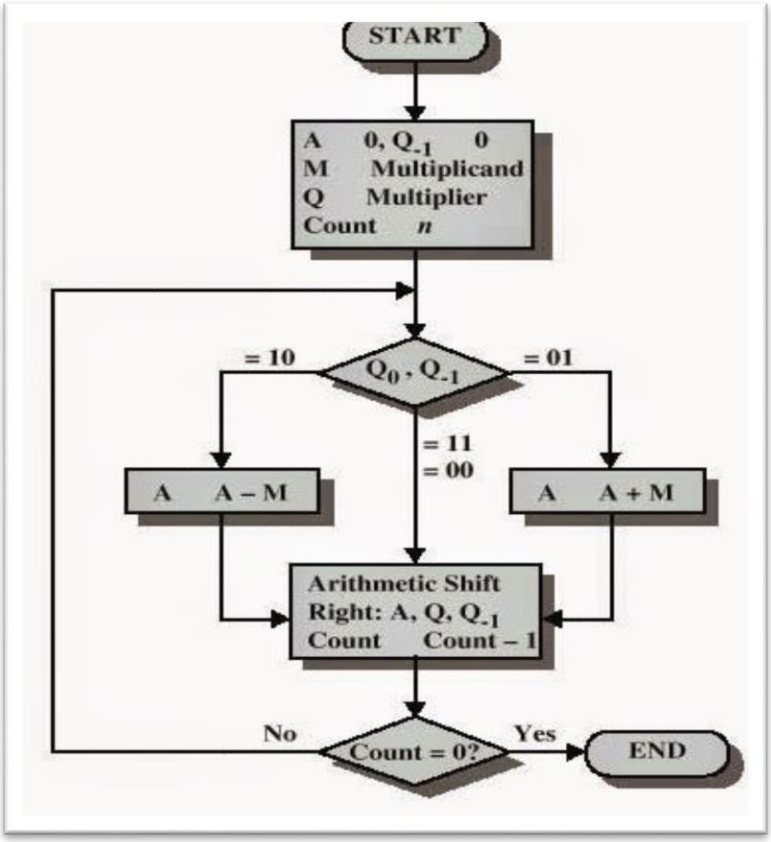
BOOTH'S ALGORITHM:

Multiplies 2 signed binary numbers in 2's complement notation

Invented by Andrew Donald booth in 1970

CONDITIONS:

1. If $Q_0 Q_{-1}$ are same i.e. 00 or 11 then, perform arithmetic right shift by 1 bit.
2. If $Q_0 Q_{-1}$ = 10 then perform
   A←A-M
   And then perform arithmetic right shift.
3. If $Q_0 Q_{-1}$ = 01 then perform
   A←A+M
   And then perform arithmetic right shift.



Code:

```
import java.util.Scanner;
public class Booth {
    public int multiply(int n1, int n2) {
```

```
        int m = n1;
        int r = n2;
        int A = n1;
        int S = -n1;
        int P = 0;
        int count = Integer.SIZE;
                while (count > 0) {
           if ((r & 1) == 1) {
              P += A;
              S += m;
           }
           A <<= 1;
           S <<= 1;
           count--;
           r >>= 1;
        }
        return P;
    }
public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        Booth b = new Booth();
        System.out.println("Enter two integer numbers -");
        int n1 = scan.nextInt();
        int n2 = scan.nextInt();
        int result = b.multiply(n1, n2);
        System.out.println("\n\nResult : " + n1 + " * " + n2 + " = " + result);
    }
}
```

Time complexity:O(n)

Space complexity:O(1)

EUCLID'S ALGORITHM:

For computing greatest common divisor of 2 numbers

Code:

```
import java.util.Scanner;

public class Main {

    public static int findgcd(int a, int b) {
        if (a == 0) {
            return b;
        }
        return findgcd(b % a, a);  // Recursively call with the new pair
    }

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

        // Get user input for two numbers
```

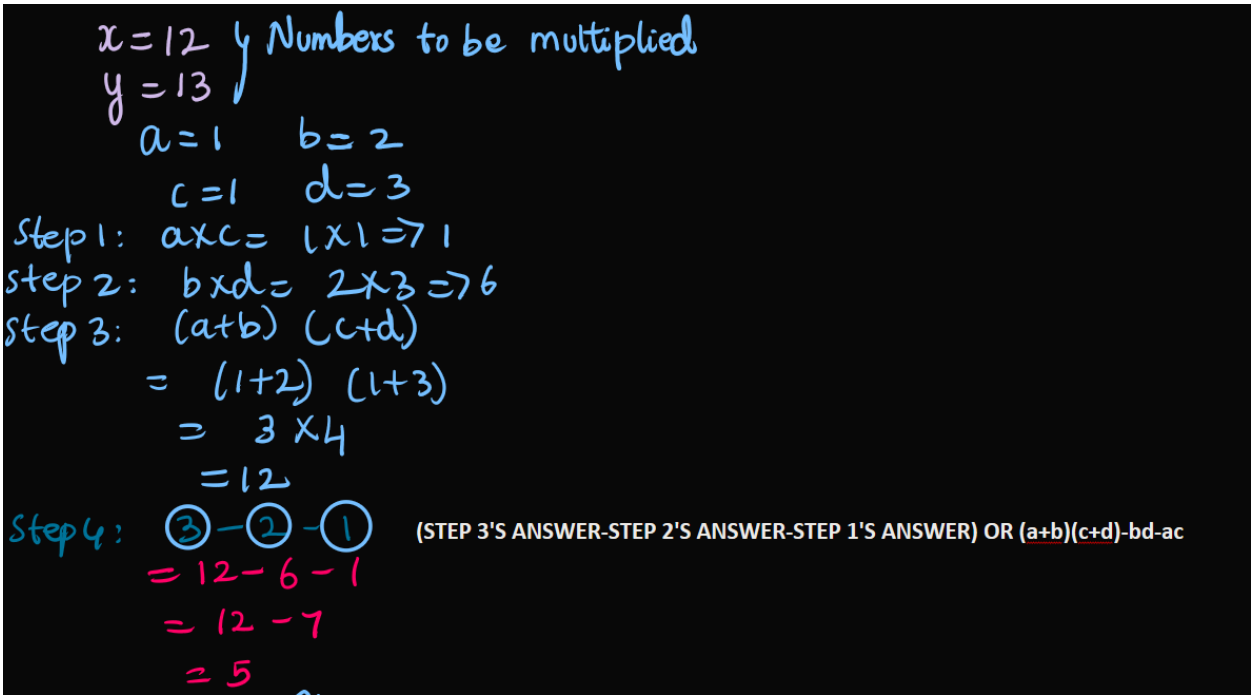
```
        System.out.print("Enter first number: ");
        int a = sc.nextInt();

        System.out.print("Enter second number: ");
        int b = sc.nextInt();

        // Find GCD using the findgcd method
        int g = findgcd(a, b);

        // Print the result
        System.out.println("GCD(" + a + "," + b + ") = " + g);

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

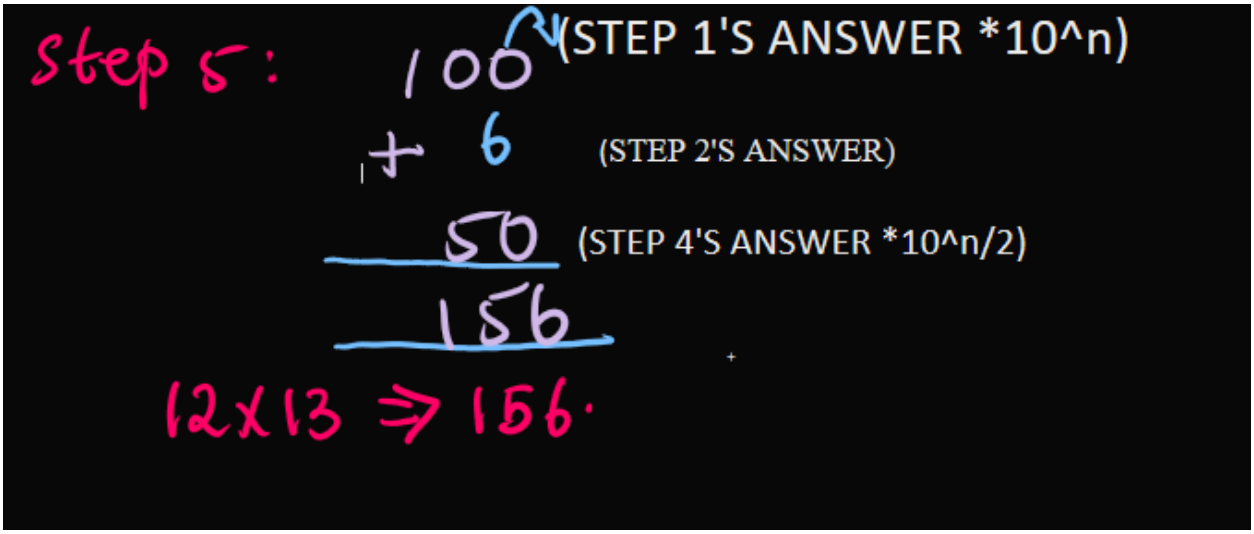
OUTPUT:

Time complexity:O(log(min(a,b)))

Space complexity: O(1)

KARATSUBA ALGORITHM:





Code:

```
import java.util.Scanner;

public class KaratsubaMultiplication {
    public static long karatsubaMultiply(long x, long y) {
        if (x < 10 || y < 10) {
            return x * y;
        }
        int n = Math.max(Long.toString(x).length(), Long.toString(y).length());
        int half = (n + 1) / 2;

        long a = x / (long) Math.pow(10, half);
        long b = x % (long) Math.pow(10, half);
        long c = y / (long) Math.pow(10, half);
        long d = y % (long) Math.pow(10, half);

        long ac = karatsubaMultiply(a, c);
        long bd = karatsubaMultiply(b, d);
```

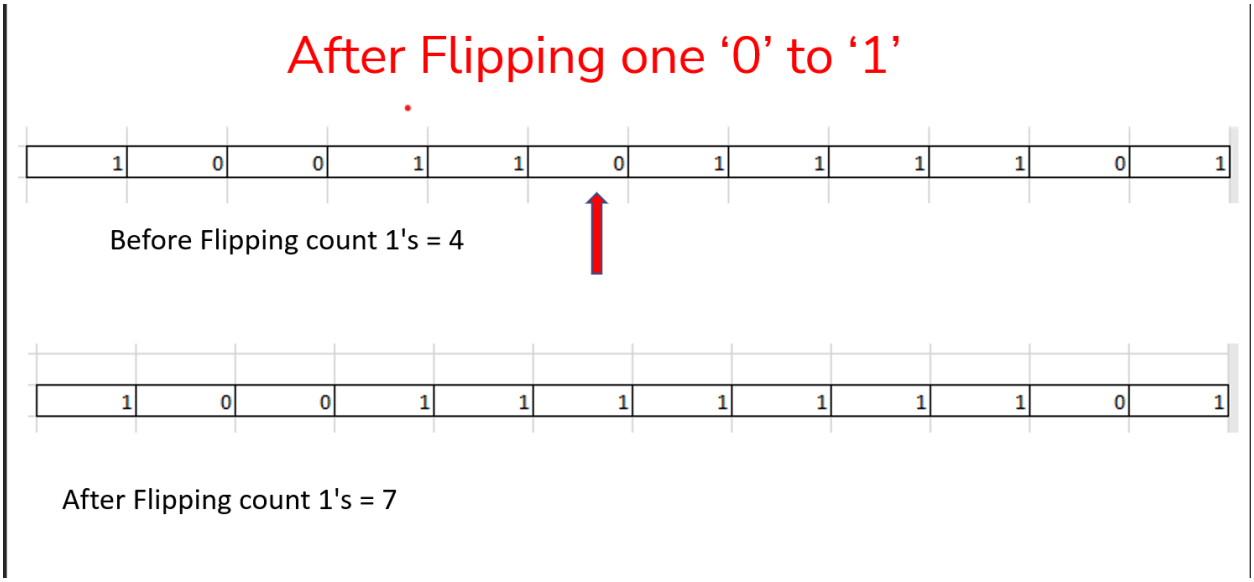
```
        long adbc = karatsubaMultiply(a + b, c + d) - ac - bd;

        return (long) (ac * Math.pow(10, 2 * half) + adbc * Math.pow(10, half) + bd);
    }

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

        System.out.print("Enter the first number: ");
        long x = scanner.nextLong();

        System.out.print("Enter the second number: ");
        long y = scanner.nextLong();

        long product = karatsubaMultiply(x, y);
        System.out.println("Product: " + product);

        scanner.close();
    }
}
```

OUTPUT:

Time complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$

Space complexity:O(log n)

Longest Sequence of 1 after flipping a bit:




```
import java.util.Scanner;


public class Longafterflip {
    public static int longestConsecutiveOnes(int n) {
        String binary = Integer.toBinaryString(n);
        int maxLength = 0;
        int currentLength = 0;
        int previousLength = 0;

        for (char bit : binary.toCharArray()) {
            if (bit == '1') {
                currentLength++;
            } else {
                maxLength = Math.max(maxLength, currentLength + previousLength + 1);
                previousLength = currentLength;
```

```
            currentLength = 0;
        }
    }
    maxLength = Math.max(maxLength, currentLength + previousLength + 1);
    return maxLength;
  }

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

    System.out.print("Enter an integer: ");
    int n = scanner.nextInt();

    System.out.println("Longest consecutive ones after one flip: " +
longestConsecutiveOnes(n));

    scanner.close();
  }
}
```

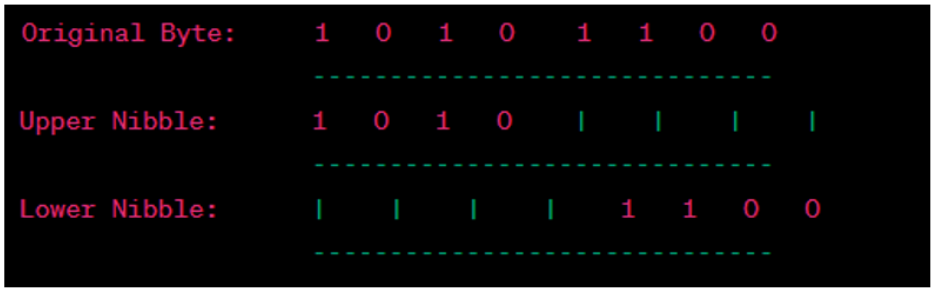
Output:

Time complexity:O(log n)

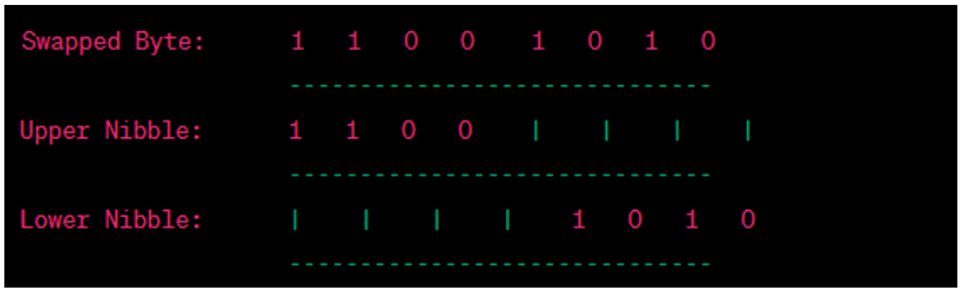
Space complexity: O(log n)

Swap two nibbles in a byte:

Consider the original byte 1010 1100:



After swapping the nibbles, the resulting byte 1100 1010 is obtained:




```
public class NibbleSwap {
    public static byte swapNibbles(byte b) {
        // Extract the upper and lower nibbles
        byte upperNibble = (byte) ((b & 0xF0) >>> 4);
        byte lowerNibble = (byte) (b & 0x0F);

        // Shift the nibbles and combine them
        byte swappedByte = (byte) ((lowerNibble << 4) | upperNibble);

        return swappedByte;
    }
    public static void main(String[] args) {
        byte byteValue = (byte) 0xAB;  // 1010 1011 in binary
        byte swappedByte = swapNibbles(byteValue);
```

```
        System.out.println("Original byte: " + Integer.toBinaryString(byteValue & 0xFF));
        System.out.println("Swapped byte: " + Integer.toBinaryString(swappedByte & 0xFF));
    }
}
```

Output:

GETTING BINARY INPUT FROM USER:

```
import java.util.Scanner;

public class NibbleSwap {
    public static byte swapNibbles(byte b) {
        // Extract the upper and lower nibbles
        byte upperNibble = (byte) ((b & 0xF0) >>> 4);
        byte lowerNibble = (byte) (b & 0x0F);

        // Shift the nibbles and combine them
        return (byte) ((lowerNibble << 4) | upperNibble);
    }

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

        System.out.print("Enter a binary value: ");
        String binaryInput = scanner.nextLine();

        // Convert binary string to byte
```

```
        byte byteValue = (byte) Integer.parseInt(binaryInput, 2);
        byte swappedByte = swapNibbles(byteValue);

        System.out.println("Original byte: " + binaryInput);
        System.out.println("Swapped byte: " + String.format("%8s",
Integer.toBinaryString(swappedByte & 0xFF)).replace(' ', '0'));

        scanner.close();
    }
}
```

OUTPUT:

Time complexity:O(1)

Space complexity:O(1)

BLOCK SWAP ALGORITHM :(for array rotation)

Initialize A = arr[0..d-1] and B = a+rr[d..n-1]

1) Do following until size of A is equal to size of B

a) If A is shorter, divide B into Bl and Br such that Br is of same length as A. Swap A and Br to change ABlBr into BrBlA. Now A is at its final place, so recur on pieces of B.

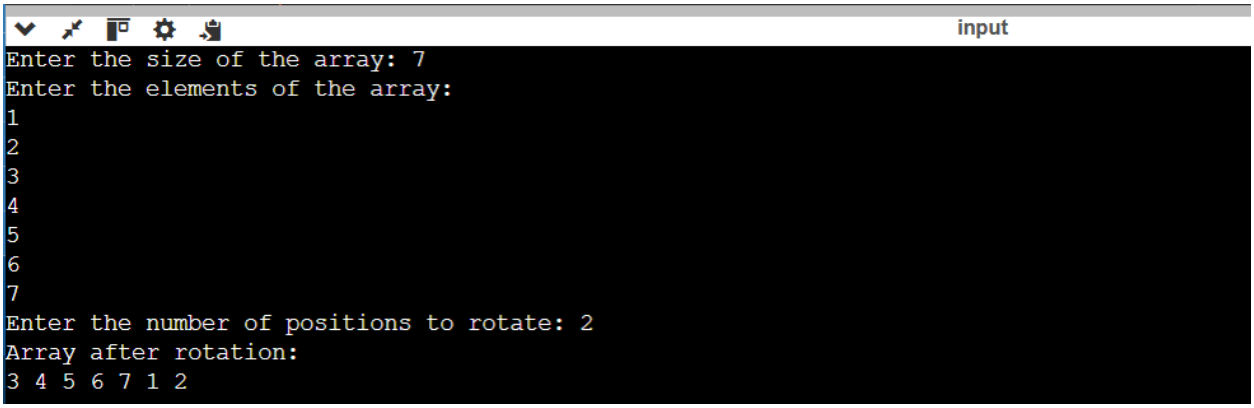
b) If A is longer, divide A into Al and Ar such that Al is of same length as B Swap Al and B to change AlArB into BArAl. Now B is at its final place, so recur on pieces of A.

```
import java.util.Scanner;

public class Main {
    static void leftRotate(int arr[], int d, int n) {
        if (d == 0 || d == n)
            return;
        if (d > n)
            d = d % n;

        int i = d, j = n - d;
        while (i != j) {
            if (i < j) {
                swap(arr, d - i, d + j - i, i);
                j -= i;
            } else {
                swap(arr, d - i, d, j);
                i -= j;
            }
        }
        swap(arr, d - i, d, i);
    }

    public static void printArray(int arr[], int size) {
        for (int i = 0; i < size; i++)
```

```
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void swap(int arr[], int fi, int si, int d) {
    for (int i = 0; i < d; i++) {
        int temp = arr[fi + i];
        arr[fi + i] = arr[si + i];
        arr[si + i] = temp;
    }
}

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

    // Input array size
    System.out.print("Enter the size of the array: ");
    int n = scanner.nextInt();

    // Input array elements
    int[] arr = new int[n];
    System.out.println("Enter the elements of the array:");
    for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
    }

    // Input rotation value
    System.out.print("Enter the number of positions to rotate: ");
```

```
        int d = scanner.nextInt();

        // Perform rotation and print the result
        leftRotate(arr, d, n);
        System.out.println("Array after rotation:");
        printArray(arr, n);

        scanner.close();
    }
}
```

Output:



Getting input with comas:

```
import java.util.Scanner;

public class Main {
    static void leftRotate(int arr[], int d, int n) {
        if (d == 0 || d == n)
            return;
        if (d > n)
            d = d % n;
```

```
    int i = d, j = n - d;
    while (i != j) {
        if (i < j) {
            swap(arr, d - i, d + j - i, i);
            j -= i;
        } else {
            swap(arr, d - i, d, j);
            i -= j;
        }
    }
    swap(arr, d - i, d, i);
}

public static void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void swap(int arr[], int fi, int si, int d) {
    for (int i = 0; i < d; i++) {
        int temp = arr[fi + i];
        arr[fi + i] = arr[si + i];
        arr[si + i] = temp;
    }
}

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

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

// Input number of elements in the array
System.out.print("Enter the number of elements in the array: ");
int n = scanner.nextInt();
scanner.nextLine(); // Consume the newline character

// Input array elements as a comma-separated string
System.out.print("Enter the elements of the array (comma-separated): ");
String input = scanner.nextLine();

// Split the string and convert to an integer array
String[] parts = input.split(",");
if (parts.length != n) {
    System.out.println("Error: The number of elements does not match the specified size.");
    return;
}

int[] arr = new int[n];
for (int i = 0; i < n; i++) {
    arr[i] = Integer.parseInt(parts[i].trim());
}

// Input rotation value
System.out.print("Enter the number of positions to rotate: ");
int d = scanner.nextInt();

// Perform rotation and print the result
```

```
        leftRotate(arr, d, arr.length);
        System.out.println("Array after rotation:");
        printArray(arr, arr.length);

        scanner.close();
    }
}
```

Time complexity:O(n)

Space complexity:O(1)

MAXIMUM PRODUCT SUBARRAY:

The subarrays for the array {4, 0, -2, 6} are:

[4]
[4, 0]
[0]
[4, 0, -2]
[0, -2]
[4, 0, -2, 6]
[-2]
[0, -2, 6]
[-2, 6]
[6]

In general, for an array of length N, there are N(N+1)/2 subarrays.

The subarrays for the array {4, 0, -2, 6} products are:

[4]x1 =1

[4x0]=0

[0]x1=0 [4x 0x -2]=0

[0x-2]=0 [4x 0x -2x 6]=0

[-2]x1=-2 [0x -2x 6]=0

[-2x6]=-12

[6]x1=6

The max product of sub array is 6

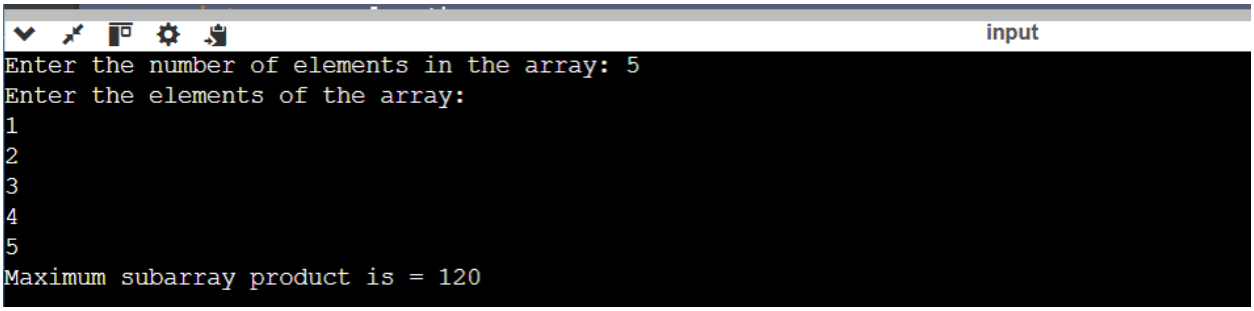
Code:

```
import java.util.Scanner;

class Main {
    // Function to find the maximum product subarray
    static int maxSubarrayProduct(int arr[]) {
        int n = arr.length;

        // Initialize variables to keep track of the max, min, and overall max product
        int maxProduct = arr[0];
        int minProduct = arr[0];
        int result = arr[0];

        // Traverse the array from the 2nd element onward
        for (int i = 1; i < n; i++) {
            // If the current element is negative, swap the max and min
```

```
        if (arr[i] < 0) {
            int temp = maxProduct;
            maxProduct = minProduct;
            minProduct = temp;
        }

        // Update maxProduct and minProduct
        maxProduct = Math.max(arr[i], maxProduct * arr[i]);
        minProduct = Math.min(arr[i], minProduct * arr[i]);

        // Update the result with the maximum product so far
        result = Math.max(result, maxProduct);
    }

    return result;
}

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

    // Input the size of the array
    System.out.print("Enter the number of elements in the array: ");
    int n = scanner.nextInt();

    // Input the array elements
    int[] arr = new int[n];
    System.out.println("Enter the elements of the array:");
    for (int i = 0; i < n; i++) {
```

```
            arr[i] = scanner.nextInt();
        }

        // Calculate and print the maximum subarray product
        int result = maxSubarrayProduct(arr);
        System.out.println("Maximum subarray product is = " + result);

        scanner.close();
    }
}
```

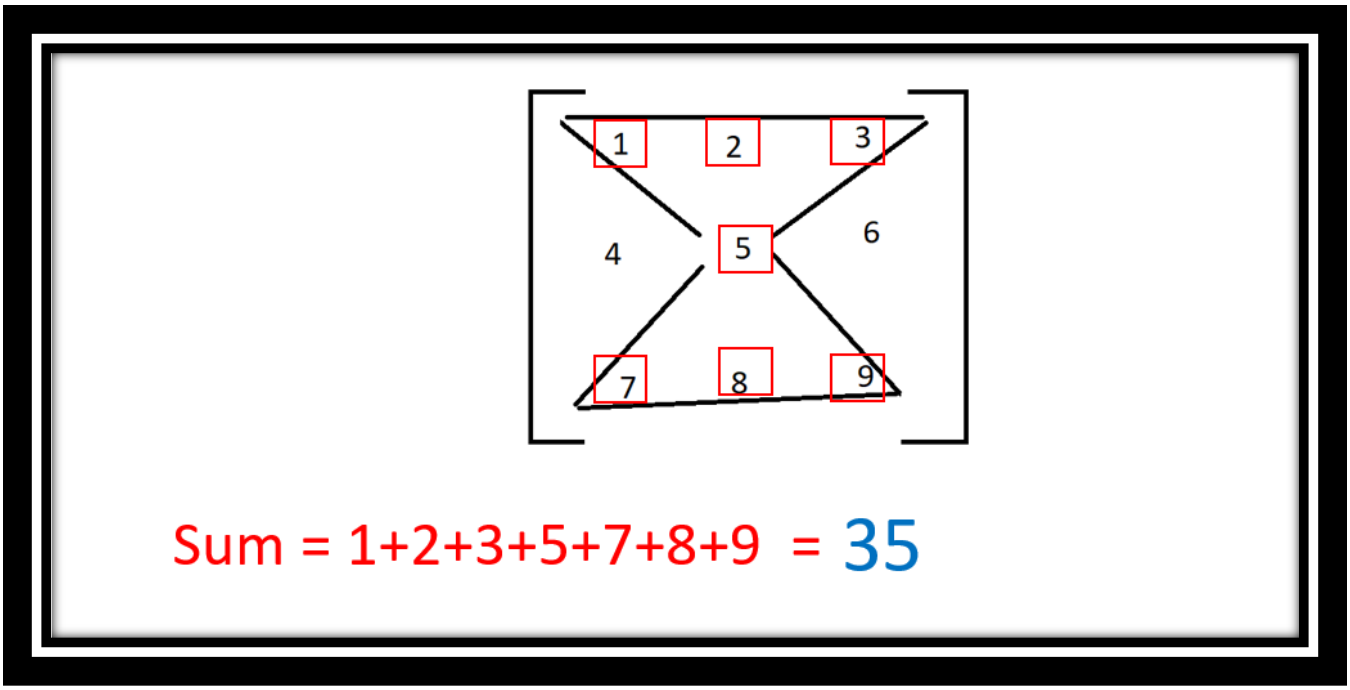
Output:



Time complexity:O(n)

Space complexity:O(1)

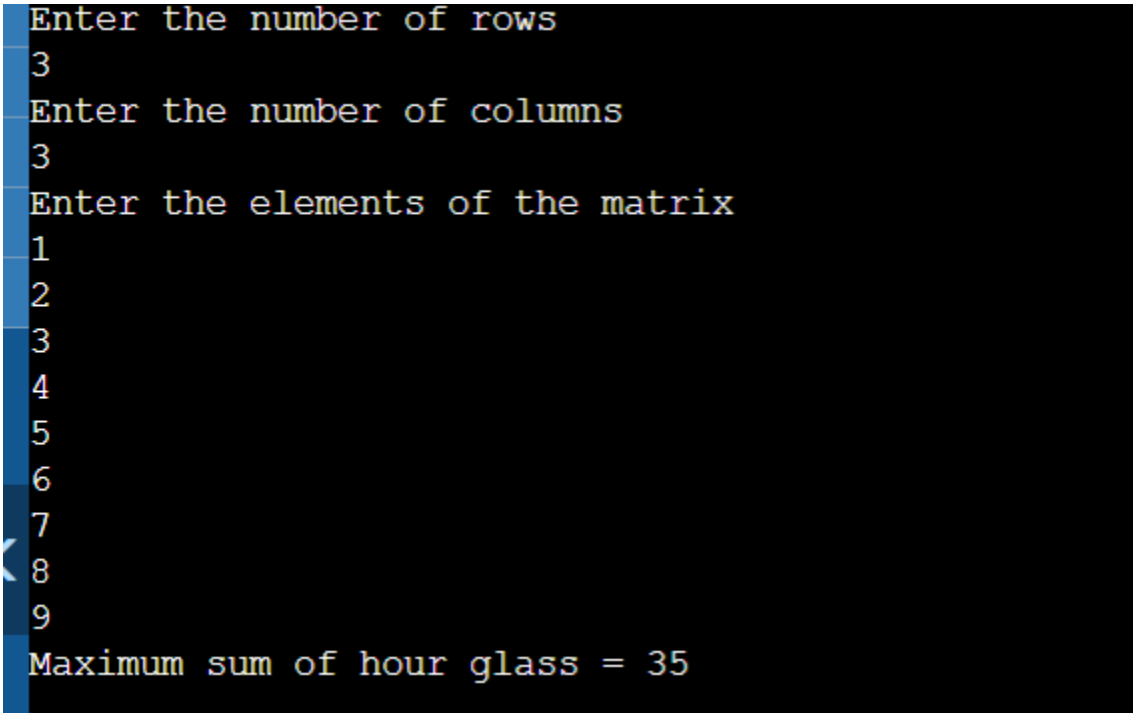
MAXIMUM HOUR GLASS SUM:




```
import java.util.Scanner;
public class Hourglass {
    static int findMaxSum(int [][]mat,int R,int C)
    {
        int max_sum=0;
        int sum;

        if (R < 3 || C < 3){
            System.out.println("Not possible");
            System.exit(0);
        }
        for (int i = 0; i < R - 2; i++)
        {
            for (int j = 0; j < C - 2; j++)
            {
                sum = (mat[i][j] + mat[i][j + 1] +
                        mat[i][j + 2]) + (mat[i + 1][j + 1]) +
```

```
                    (mat[i + 2][j] + mat[i + 2][j + 1] +
                    mat[i + 2][j + 2]);
                max_sum = Math.max(max_sum, sum);
            }
        }
        return max_sum;
    }
static public void main (String[] args)
        {
            Scanner sc = new Scanner(System.in);
            System.out.println("Enter the number of rows");

            int R = sc.nextInt();
            System.out.println("Enter the number of columns");
            int C= sc.nextInt();

            int [][]mat = new int[R][C];
            System.out.println("Enter the elements of the matrix");
            for (int i = 0; i < R; i++) {
                for (int j = 0; j < C; j++) {
                    mat[i][j]= sc.nextInt();
                }
            }
            int res = findMaxSum(mat,R,C);
            System.out.println("Maximum sum of hour glass = "+ res);
        }

}
```

Output:



Time complexity:O(R x C)

Space complexity:O(1)

Max Equilibrium Sum:

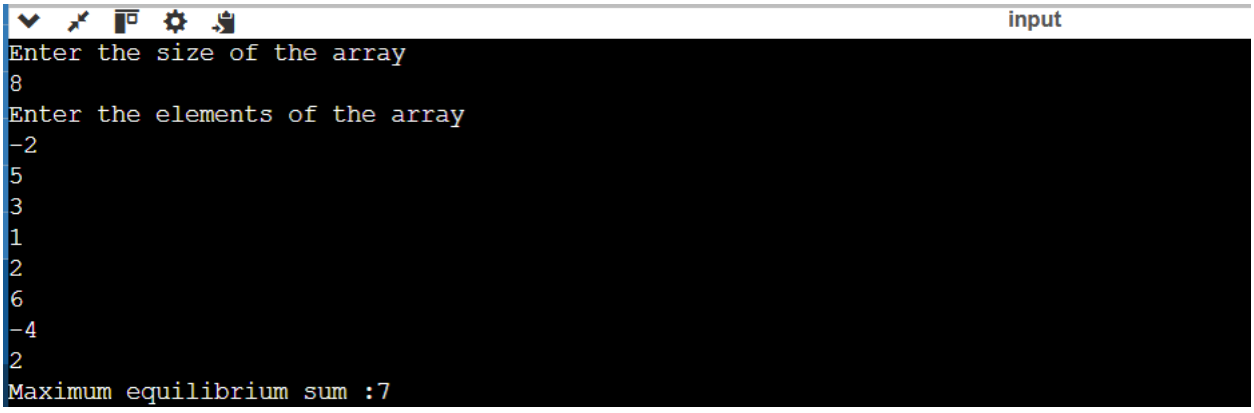
Code:

```
import java.util.Scanner;
import java.io.*;
public class Main
{
```

```
public static int equlibrium_sum(int a[],int n)
{
int i;
int sum =0,max =0,prefix_sum=0;
for( i=0;i<n;i++)
{
sum =sum+a[i];
} for(i=0;i<n;i++)
{
prefix_sum +=a[i];
if (prefix_sum == sum)
{
   if(max<prefix_sum)
   {
      max=prefix_sum;
   }
}
sum = sum - a[i];
}
return max;
}
public static void main (String[] args)
{
   Scanner sw = new Scanner(System.in);
   System.out.println("Enter the size of the array");

   int n=sw.nextInt();
int arr[] = new int[n];
```

```
System.out.println("Enter the elements of the array");
for(int i =0;i<n;i++) arr[i]=sw.nextInt();
int result=equlibrium_sum(arr, n);
if(result==0)
System.out.println("None");
else
System.out.println("Maximum equilibrium sum :"+result);


}
}
```

Output:



Time complexity:O(n)

Space complexity:O(1)

Leaders in array:

```
import java.io.*;
public class LeadersInArray {
    /*Java Function to print leaders in an array */
    void printLeaders(int arr[], int size)
    {
        for (int i = 0; i < size; i++) {
            int j;
            for (j = i + 1; j < size; j++) {
                if (arr[i] <= arr[j])
                    break;
            }
            if (j == size) // the loop didn't break
                System.out.print(arr[i] + " ");
        }
    }

    /* Driver program to test above functions */
    public static void main(String[] args)
    {
        LeadersInArray lead = new LeadersInArray();
        int arr[] = new int[] { 16, 17, 4, 3, 5, 2 };
        int n = arr.length;
        lead.printLeaders(arr, n);
    }
}
```

Output:

Time complexity:O(n^2)

Space complexity:O(1)

Majority element:

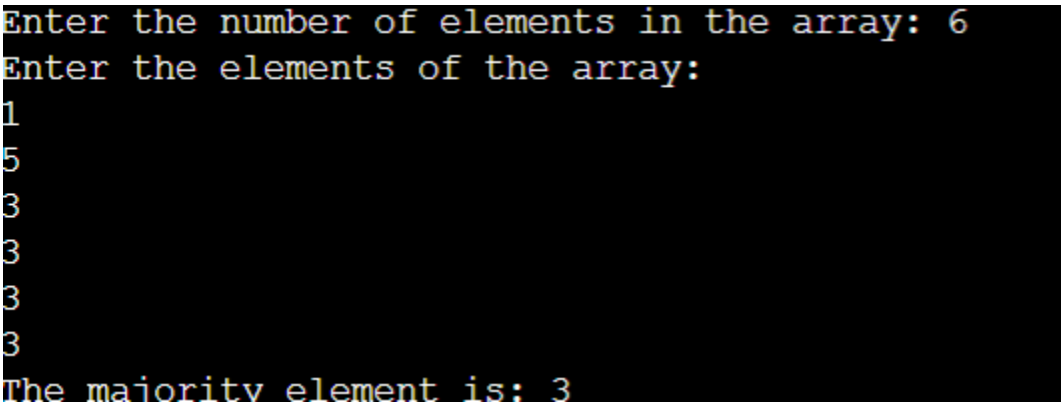
Boyer moore vote algorithm(for majority element):

```
import java.util.Scanner;

public class MajorityElementFinder {

    // Function to find the majority element
    public static int findMajorityElement(int[] arr, int n) {
        // Phase 1: Find the candidate for the majority element
        int candidate = -1;  // Use a valid placeholder for int
        int count = 0;

        for (int i = 0; i < n; i++) {
            if (count == 0) {
                candidate = arr[i];
                count = 1;
            } else if (candidate == arr[i]) {
                count++;
            } else {
                count--;
            }
```

```
    }

    // Phase 2: Verify if the candidate is the majority element
    count = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] == candidate) {
            count++;
        }
    }

    // Return the candidate if it is the majority element, otherwise return -1
    return (count > n / 2) ? candidate : -1;  // Returning -1 if no majority found
}

public static void main(String[] args) {
    // Create a scanner object to take input from the user
    Scanner scanner = new Scanner(System.in);

    // Taking input for the size of the array
    System.out.print("Enter the number of elements in the array: ");
    int n = scanner.nextInt();

    // Initialize the array with the input size
    int[] arr = new int[n];

    // Taking array input from the user
    System.out.println("Enter the elements of the array:");
    for (int i = 0; i < n; i++) {
```

```
            arr[i] = scanner.nextInt();
        }

        // Finding and printing the majority element
        int result = findMajorityElement(arr, n);
        if (result != -1) {
            System.out.println("The majority element is: " + result);
        } else {
            System.out.println("There is no majority element.");
        }

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

Output:



Time complexity:O(n)

Space complexity:O(1)

Lexiographically first palindrome:

```
class Main{

	static char MAX_CHAR = 26;

	// Function to count frequency of each char in the
	// string. freq[0] for 'a',...., freq[25] for 'z'
	static void countFreq(String str, int freq[], int len)
	{
		for (int i = 0; i < len; i++)
		{
			freq[str.charAt(i) - 'a']++;
		}
	}

	// Cases to check whether a palindr0mic
	// string can be formed or not
	static boolean canMakePalindrome(int freq[], int len)
	{
		// count_odd to count no of
		// chars with odd frequency
		int count_odd = 0;
		for (int i = 0; i < MAX_CHAR; i++)
		{
			if (freq[i] % 2 != 0)
			{
				count_odd++;
```

```
            }
        }

        // For even length string
        // no odd freq character
        if (len % 2 == 0)
        {
            if (count_odd > 0)
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        // For odd length string
        // one odd freq character
        if (count_odd != 1)
        {
            return false;
    }

        return true;
    }

    // Function to find odd freq char and
```

```
// reducing its freq by 1returns "" if odd freq
// char is not present
static String findOddAndRemoveItsFreq(int freq[])
{
        String odd_str = "";
        for (int i = 0; i < MAX_CHAR; i++)
        {
                if (freq[i] % 2 != 0)
                {
                        freq[i]--;
                        odd_str = odd_str + (char) (i + 'a');
                        return odd_str;
                }
        }
        return odd_str;
}

// To find lexicographically first palindromic
// string.
static String findPalindromicString(String str)
{
        int len = str.length();
        int freq[] = new int[MAX_CHAR];
        countFreq(str, freq, len);

        if (!canMakePalindrome(freq, len))
        {
                return "No Palindromic String";
```

```
}

// Assigning odd freq character if present
// else empty string.
String odd_str = findOddAndRemoveItsFreq(freq);

String front_str = "", rear_str = " ";

// Traverse characters in increasing order
for (int i = 0; i < MAX_CHAR; i++)
{
        String temp = "";
        if (freq[i] != 0)
        {
                char ch = (char) (i + 'a');

                // Divide all occurrences into two
                // halves. Note that odd character
                // is removed by findOddAndRemoveItsFreq()
                for (int j = 1; j <= freq[i] / 2; j++)
                {
                        temp = temp + ch;
                }

                // creating front string
                front_str = front_str + temp;

                // creating rear string
```

```
                        rear_str = temp + rear_str;
                    }
                }

                // Final palindromic string which is
                // lexicographically first
                return (front_str + odd_str + rear_str);
        }

        // Driver program
        public static void main(String[] args)
        {
                String str = "malayalam";
                System.out.println(findPalindromicString(str));
        }
}
```

Output:

Time complexity:O(n)

Space complexity:O(1)

NATURAL SORT ORDER:

Natural sort order is a way of sorting strings or text data in a manner that is more intuitive to humans. In natural sort order, alphanumeric strings are sorted based on their numeric components rather than just their character codes.

Typically, when sorting strings using a regular lexicographic sort order, the sorting is performed based on the ASCII or Unicode values of the characters. This means that numbers within strings are treated as individual characters rather than numeric values. As a result, strings like "file1", "file10", and "file2" would be sorted in lexicographic order as "file1", "file10", and "file2", which is not the intuitive ordering.

In contrast, natural sort order takes into account the numerical values within the strings. It treats numbers as numbers rather than individual characters. Using natural sort order, the same strings would be sorted as "file1", "file2", and "file10", which is the expected and intuitive ordering.

Natural sort order can be implemented using various algorithms or techniques. One common approach involves splitting the strings into chunks of numeric and non-numeric parts, and then sorting based on these parts. This allows the numbers within the strings to be compared and sorted in a meaningful way.

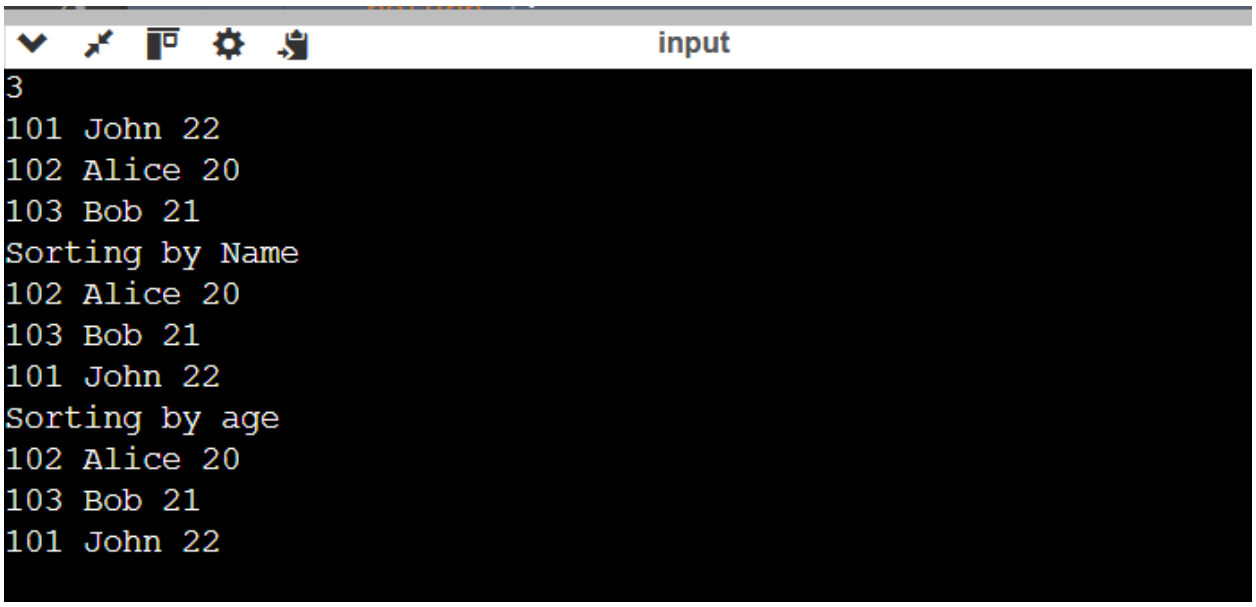
Natural sort order is particularly useful when dealing with file names, version numbers, or any other alphanumeric data that contains numbers. It provides a more human-friendly and logical sorting order compared to simple lexicographic sorting.

Code:

```
//Natural sort order

import java.util.*;

import java.io.*;


class Student {
```

```
    int rollno;
    String name;
    int age;

    Student(int rollno, String name, int age) {
        this.rollno = rollno;
        this.name = name;
        this.age = age;
    }
}

class NameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

class AgeComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        if (s1.age == s2.age)
            return 0;
        else if (s1.age > s2.age)
            return 1;
        else
            return -1;
    }
}
```

```
public class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        ArrayList<Student> al = new ArrayList<Student>();
        int r, a;
        String sname;
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            r = sc.nextInt();
            sname = sc.next();
            a = sc.nextInt();
            al.add(new Student(r, sname, a));
        }
        System.out.println("Sorting by Name");
        Collections.sort(al, new NameComparator());
        for (Student st : al) {
            System.out.println(st.rollno + " " + st.name + " " + st.age);
        }
        System.out.println("Sorting by age");
        Collections.sort(al, new AgeComparator());
        for (Student st : al) {
            System.out.println(st.rollno + " " + st.name + " " + st.age);
        }
    }
}OUTPUT:
```



Time complexity: O(n log n)

Space complexity: O(n)

QUICK SORT AND SELECTION SORT:

QUICK SORT:

Here's an example to illustrate the steps:

Consider the list: [8, 3, 1, 7, 0, 10, 2]

Step 1: Choose the pivot. Let's choose the last element, 2.

Step 2: Partition the list. Rearrange the elements such that all elements smaller than 2 come before it, and all elements greater than 2 come after it.

Partitioned list: [1, 0, 2, 7, 8, 10, 3]

Step 3: Recursively apply the above steps to the sub-arrays. In this case, we have two sub-arrays: [1, 0] and [7, 8, 10, 3].

For the sub-array [1, 0]:

- Choose the last element, 0, as the pivot.

- Partition the list: [0, 1]

For the sub-array [7, 8, 10, 3]:

- Choose the last element, 3, as the pivot.

- Partition the list: [7, 8, 3, 10]

Continue applying the steps recursively to the sub-arrays until each sub-array contains only one element or is empty.

Step 4: The recursion ends when each sub-array is sorted (contains only one element or is empty). At this point, the entire list is sorted.

Final sorted list: [0, 1, 2, 3, 7, 8, 10]

QuickSort has an average time complexity of O(n log n), making it efficient for large lists. However, its worst-case time complexity can be O(n^2) if the pivot selection is unbalanced, which can be mitigated by using randomized pivot selection or other techniques. Overall, QuickSort is a widely used and efficient sorting algorithm.

```
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {8, 3, 1, 7, 0, 10, 2};
        quickSort(arr, 0, arr.length - 1);
        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // Partition the array
```

```
            int partitionIndex = partition(arr, low, high);
            // Recursively sort the sub-arrays
            quickSort(arr, low, partitionIndex - 1);
            quickSort(arr, partitionIndex + 1, high);
        }
    }
public static int partition(int[] arr, int low, int high) {
int pivot = arr[high];
int i = low - 1;
        for (int j = low; j < high; j++) {
if (arr[j] <= pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}
```

OUTPUT:

Time complexity:O(n^2)

Space complexity:O(n)

SELECTION SORT:

Here's an example to illustrate the steps:

Iteration 1:

- Find the minimum element from the unsorted part [64, 25, 12, 22, 11]. The minimum is 11.
- Swap the minimum element (11) with the first element (64) of the unsorted part.
- The sorted part becomes [11], and the unsorted part becomes [64, 25, 12, 22].

Iteration 2:

- Find the minimum element from the unsorted part [64, 25, 12, 22]. The minimum is 12.
- Swap the minimum element (12) with the first element (64) of the unsorted part.
- The sorted part becomes [11, 12], and the unsorted part becomes [64, 25, 22].

Iteration 3:

- Find the minimum element from the unsorted part [64, 25, 22]. The minimum is 22.
- Swap the minimum element (22) with the first element (64) of the unsorted part.
- The sorted part becomes [11, 12, 22], and the unsorted part becomes [64, 25].

```
public class SelectionSort {
    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        selectionSort(arr);
        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
    public static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
```

```
int minIndex = i;
            for (int j = i + 1; j < n; j++) {
               if (arr[j] < arr[minIndex]) {
                  minIndex = j;
               }
            }
int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
         }
      }
}
```

OUTPUT:

Time complexity:O(n^2)

Space complexity:O(1)

WEIGHTED SUBSTRING:

Every character has a weight.

The weight of an English uppercase alphabet A-Z is given below. A=1 B=2*A+A C=3*B+B D=4*C+C . . . Y=25*X+X Z=26*Y+Y

The weight of any string made up of these characters is the summation of weights of each character.

Given a total string weight, determine shortest string of that given weight. If there is more than one solution, return the lexicographically smallest of them. For example, given weight = 25, and the weights of the first few characters of the alphabet are A=1, B=3, C=12, D=60 it is certain that no letter larger than C is required. Some of the strings with a total weight equal to the largest are ABBBBC, ACC, and AAAAAAABBBBBB. The shortest of these is ACC. While any

permutation of these characters will have the same weight, this is the lexicographically smallest of them. Function Description Complete the function smallest String in the editor below.

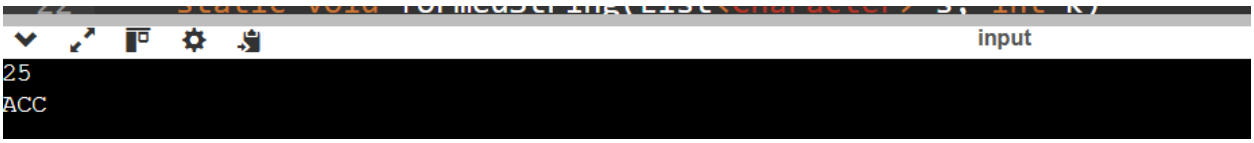
CODE:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
public class Main {
    static int[] values = new int[26];
    public static void main(String[] args) {
        insertValues();
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        List<Character> s = new ArrayList<>();
        formedString(s, n);
    }
    static void insertValues() {
        values[0] = 1;
        int prev = 1;
for (int i = 1; i < 26; i++) {
            values[i] = (i + 1) * prev + prev;
            prev = values[i];
            // System.out.println(prev);
        }
    }
    static void formedString(List<Character> s, int k)
    {
```

```
        int low = 0;
        int high = 25;
        while (k != 0) {
            int ind = findFloor(k, low, high);
            s.add((char) (ind + 'A'));
              k = k - values[ind];
            // System.out.println(k);
        }
        for (int i = s.size() - 1; i >= 0; i--){
            System.out.print(s.get(i));
}
    }
    static int findFloor(int k, int low, int high)
     {
        int ans = -1;
        while (low <= high)
            {
            int mid = (low + high) / 2;
            if (values[mid] <= k)
           {
                ans = mid;
                low = mid + 1;
            }
else {
                high = mid - 1;
            }
        }
        // System.out.println(ans);
```

```
        return ans;
    }
}
```

OUTPUT:



TIME COMPLEXITY:O(k)

SPACE COMPLEXITY:O(k)

MOVE HYPHEN TO THE BEGINNING

```
public class MoveHyphenToBeginning {
    public static void main(String[] args) {
        String originalString = "face-prep";
        String transformedString = moveHyphenToBeginning(originalString);
        System.out.println(transformedString);
    }

    public static String moveHyphenToBeginning(String string) {
        if (string.contains("-")) {
            int hyphenIndex = string.indexOf("-");
            return "-" + string.substring(0, hyphenIndex) + string.substring(hyphenIndex + 1);
        } else {
            return string;
        }
    }
}
```

OUTPUT:

Time complexity:O(n)

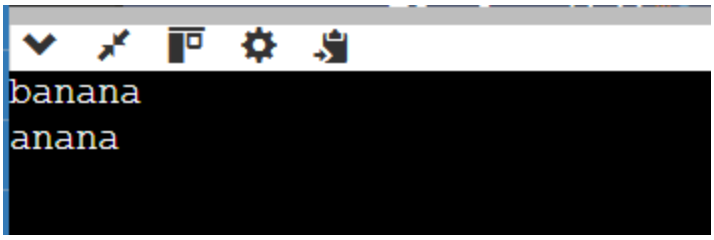
Space complexity:O(n)

MANACHER'S ALGORITHM:(Longest Palindromic substring)

```
import java.util.Scanner;
public class Main {
        public static void Long_palin(String s) {
           char[] T = new char[s.length()*2 + 3];
           T[0] = '$';
           T[s.length()*2 + 2] = '@';
           for (int i = 0; i < s.length(); i++) {
              T[2*i + 1] = '#';
              T[2*i + 2] = s.charAt(i);
           }
           T[s.length()*2 + 1] = '#';
           int[]  P = new int[T.length];
           int center = 0, right = 0;
           for (int i = 1; i < T.length-1; i++) {
              int mirr = 2*center - i;

              if (i < right)
                 P[i] = Math.min(right - i, P[mirr]);

            while (T[i + (1 + P[i])] == T[i - (1 + P[i])])
                   P[i]++;

              if (i + P[i] > right) {
```

```
                    center = i;
                    right = i + P[i];
                }
            }

            int length = 0;
            center = 0;
            for (int i = 1; i < P.length-1; i++) {
        if (P[i] > length) {
                    length = P[i];
                    center = i;
                }
            }
            //System.out.println(length+" "+center);
          System.out.println(s.substring((center - 1 - length) / 2, (center - 1 + length) / 2));

    }
          public static void main(String[] args)
          {
             Scanner sc=new Scanner(System.in);
             String s =sc.next();
             Long_palin(s);
          }
}
```

OUTPUT:



Time complexity:O(n)

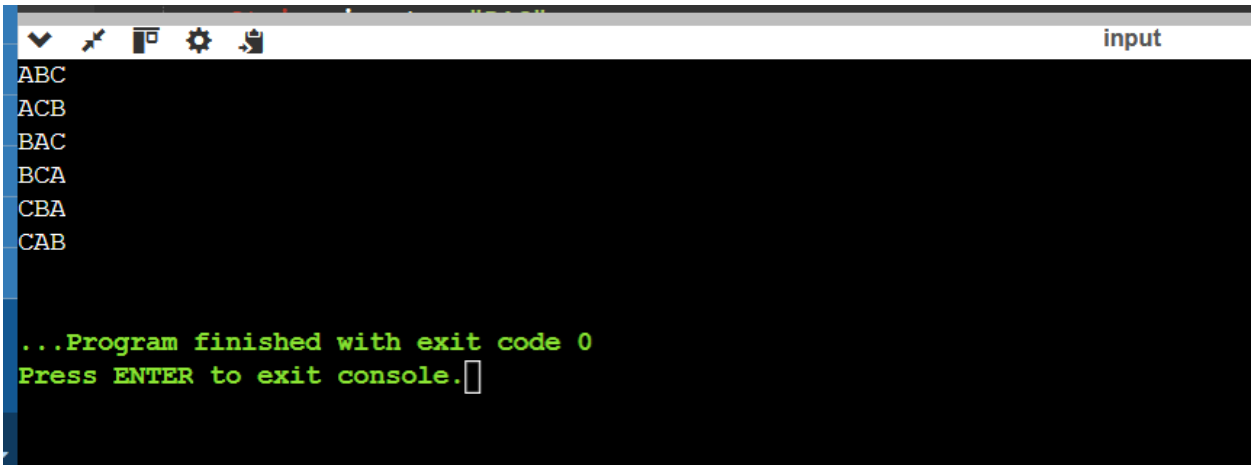
Space complexity:O(n)

SORTED UNIQUE PERMUTATION:

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
public class Permutations {
    public static void main(String[] args) {
        String input = "BAC";
        distinctPermutations(input);
    }
    public static void distinctPermutations(String input) {
        char[] chars = input.toCharArray();
        Arrays.sort(chars);
        input = new String(chars);
        permute(input.toCharArray(), 0);
    }
    public static void permute(char[] chars, int index) {
        if (index == chars.length - 1) {
            System.out.println(String.valueOf(chars));
            return;
```

```
        }
    Set<Character> used = new HashSet<>();
        for (int i = index; i < chars.length; i++) {
            if (used.contains(chars[i]))
                continue;
            used.add(chars[i]);
            swap(chars, index, i);
            permute(chars, index + 1);
            swap(chars, index, i);
        }
    }
    public static void swap(char[] chars, int i, int j) {
        char temp = chars[i];
        chars[i] = chars[j];
        chars[j] = temp;
    }
}
```

OUTPUT:



Time complexity: O(n * n!)

Space complexity:O(n)

MANUERING:

```
class Main {
   static int numberOfPaths(int m, int n){
       if (m == 1 || n == 1)
          return 1;
       return numberOfPaths(m-1,n)+numberOfPaths(m,n-1);
      // + numberOfPaths(m-1, n-1);
   }

   public static void main(String args[])
   {
      System.out.println(numberOfPaths(3, 3));
   }
}
```

OUTPUT:

Time complexity: O(2^(m+n))

Space complexity: O(m+n)

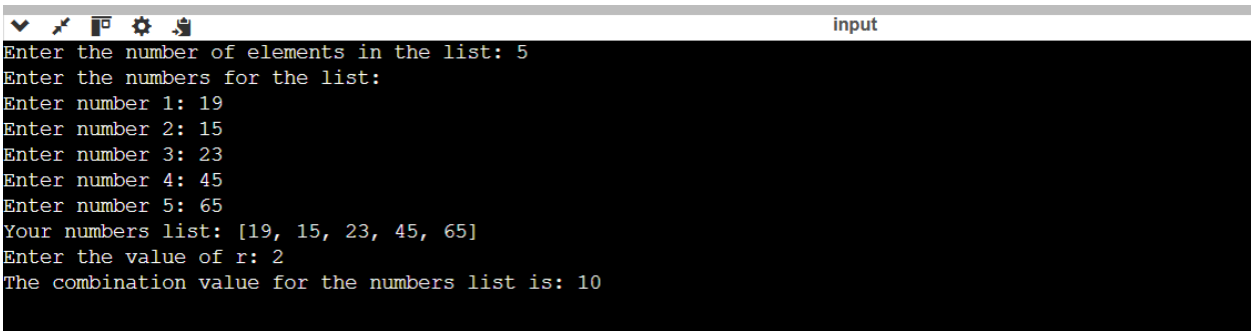
COMBINATION:

```
import java.util.*;

public class CombinationExample {
```

```
static int fact(int number) {
    int f = 1;
    int j = 1;
    while (j <= number) {
        f = f * j;
        j++;
    }
    return f;
}

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

    // Get the size of the list
    System.out.print("Enter the number of elements in the list: ");
    int n = scanner.nextInt();

    // Get the list of numbers from the user
    List<Integer> numbers = new ArrayList<>();
    System.out.println("Enter the numbers for the list:");
    for (int i = 0; i < n; i++) {
        System.out.print("Enter number " + (i + 1) + ": ");
        numbers.add(scanner.nextInt());
    }

    // Display the list
    System.out.println("Your numbers list: " + numbers);
```

```
        // Get the value of r from the user
        System.out.print("Enter the value of r: ");
        int r = scanner.nextInt();

        // Validate and calculate the combination value
        if (r > n) {
            System.out.println("The value of r cannot be greater than the size of the list.");
        } else {
            int result = fact(n) / (fact(r) * fact(n - r));
            System.out.println("The combination value for the numbers list is: " + result);
        }

        scanner.close();
    }
}
```

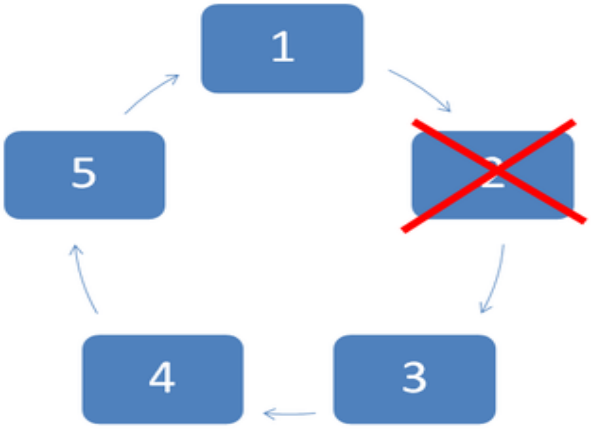
OUTPUT:



Time complexity:O(n)

Sapce complexity:O(n)

Josephus Trap:



Start = 0
K = 1
Size = 5

Start = (start + k) % size
= (0 + 1) % 5
= 1
The person at 1-index will be killed
(follow 0-indexing)

```
import java.io.*;
class Main {
    static int josephus(int n, int k){
        if (n == 1)
            return 1;
        else
return (josephus(n - 1, k) + k - 1) % n + 1;
    }

public static void main(String[] args){
        int n = 5;
        int k = 1;
        System.out.println("The chosen place is "
                           + josephus(n, k));
    }
}
```

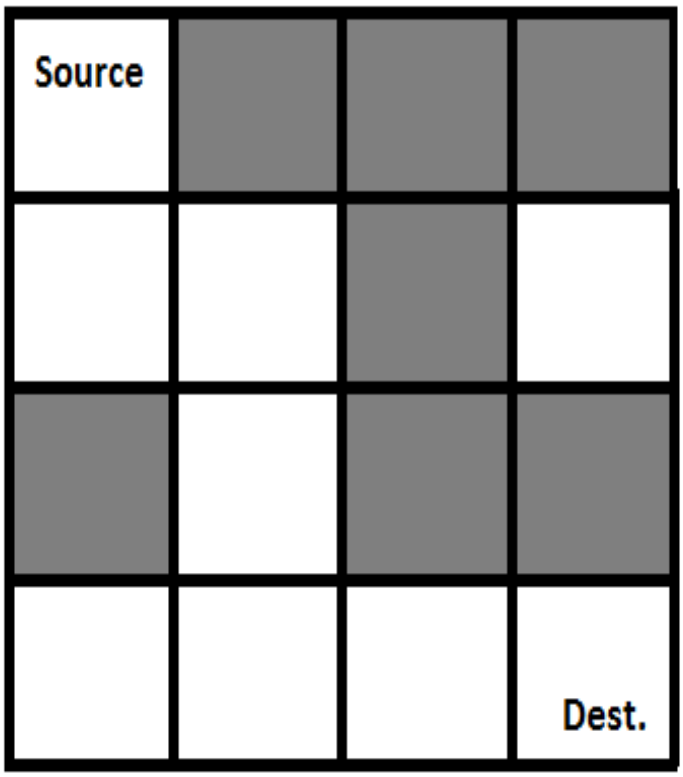
OUTPUT:

```
input
The chosen place is 5
```

Time complexity:O(n)

Space complexity:O(n)

MAZE SOLVING:



Gray blocks are dead ends (value = 0).

binary matrix representation of the above maze.

{ 1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Output:

```
1  0  0  0
1  1  0  0
0  1  0  0
0  1  1  1
```

```
public class RatMaze {
final int N = 4; // Define the size of the maze
```

```
// Print the solution path (if found)
void printSolution(int sol[][]) {
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++)
        System.out.print(" " + sol[i][j] + " ");
System.out.println();
    }
  }

  // Check if maze[x][y] is a valid move (inside bounds and not blocked)
  boolean isSafe(int maze[][], int x, int y) {
    return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
}

// Solve the maze and return true if there's a solution, false otherwise
boolean solveMaze(int maze[][]) {
int sol[][] = new int[N][N]; // Create a solution matrix initialized to 0

    // If solveMazeUtil returns false, there's no solution
if (!solveMazeUtil(maze, 0, 0, sol)) {
        System.out.print("Solution doesn't exist");
return false;
}

printSolution(sol); // Print the solution matrix
return true;
  }
```

```
// Utility function to solve the maze using backtracking
    boolean solveMazeUtil(int maze[][], int x, int y, int sol[][]) {
// If we've reached the destination (bottom-right corner), mark it as part of the solution
        if (x == N - 1 && y == N - 1) {
            sol[x][y] = 1;
return true;
}

        // Check if the current position is safe
        if (isSafe(maze, x, y)) {
// Mark the current cell as part of the solution path
            sol[x][y] = 1;

// Move forward in the x direction (down)
if (solveMazeUtil(maze, x + 1, y, sol))
                return true;

// If moving down didn't work, try moving right
if (solveMazeUtil(maze, x, y + 1, sol))
            return true;

// If neither move works, backtrack: unmark this cell
sol[x][y] = 0;
return false;
}

return false; // If not a valid move, return false
```

```
}

public static void main(String[] args) {
RatMaze ratMaze = new RatMaze();
int maze[][] = {
{ 1, 0, 0, 0 },
{ 1, 1, 0, 1 },
{ 0, 1, 0, 0 },
{ 1, 1, 1, 1 }
};

ratMaze.solveMaze(maze); // Solve the maze and print the solution
}
}
```

OUTPUT:

Time complexity: O(2^(N^2))

Space complexity: O(N^2).

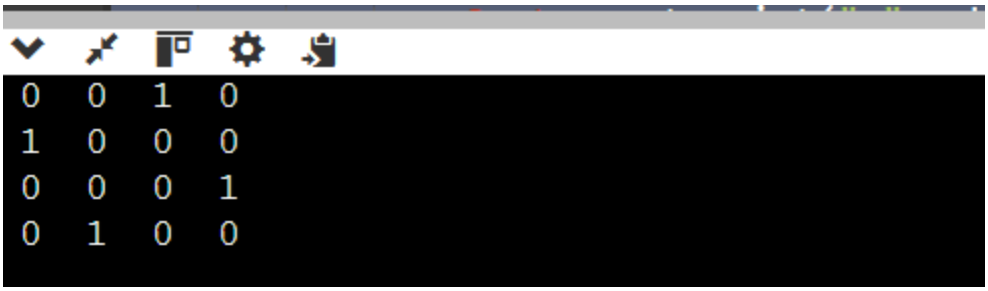
N QUEENS:

- The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

```
public class NQueenProblem {
final int N = 4;
void printSolution(int board[][]){
for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
           System.out.print(" " + board[i][j]
                    + " ");
        System.out.println();       }
   }
   boolean isSafe(int board[][], int row, int col){
      int i, j;
      for (i = 0; i < col; i++)
         if (board[row][i] == 1)
            return false;
      for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
if (board[i][j] == 1)
            return false;
      for (i = row, j = col; j >= 0 && i < N; i++, j--)
         if (board[i][j] == 1)
            return false;
      return true;
   }
   boolean solveNQUtil(int board[][], int col){
      if (col >= N)
         return true;
      for (int i = 0; i < N; i++) {
```

```
            if (isSafe(board, i, col)) {
                board[i][col] = 1;
if (solveNQUtil(board, col + 1) == true)
                    return true;
                board[i][col] = 0; // BACKTRACK
            }       }
        return false;
    }
    boolean solveNQ()
    {
        int board[][] = { { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 } };
        if (solveNQUtil(board, 0) == false) {
            System.out.print("Solution does not exist");
            return false;
        }
        printSolution(board);
        return true;
    }

    public static void main(String args[]){
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }}
```

OUTPUT:



Time complexity: O(N!)

Space complexity: O(N)

WARNSDROFF ALGORITHM:

Warnsdorff's rule is a heuristic for finding a single knight's tour.

The knight is moved so that it always proceeds to the square from which the knight will have the fewest onward moves.

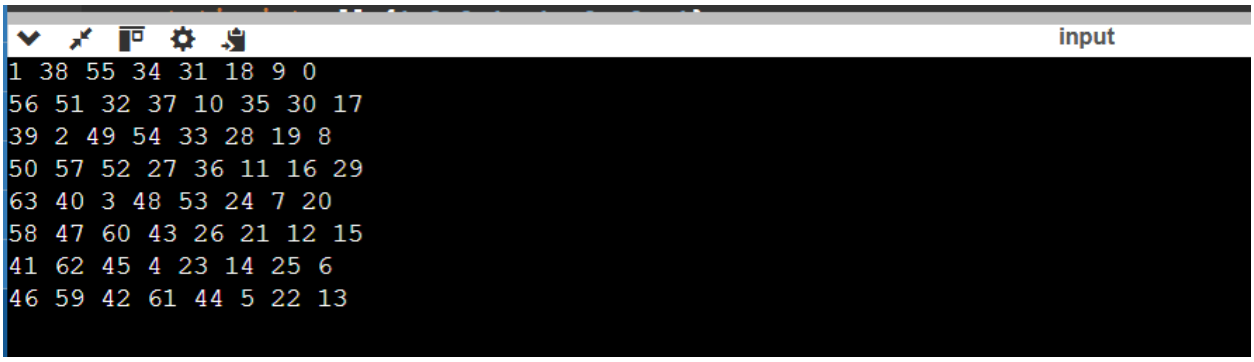
When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited.

```
import java.io.*;
import java.util.*;
public class Main {
   static int n=8;
   static int x[]={2,1,-1,-2,-2,-1,1,2};
   static int y[]={1,2,2,1,-1,-2,-2,-1};
   public static boolean tour( int [][] arr, int count, int r, int c)
   {
      if(count==(n*n))
      {
         return true;
      }
      for( int i=0;i<n;i++)
      {
```

```
        int x1=x[i]+r;
        int y1=y[i]+c;
        if(x1>=0 && y1>=0 && x1<n && y1<n && arr[x1][y1]==0)
        {
            arr[x1][y1]=count;
            if(tour(arr,count+1,x1,y1))
                return true;
            arr[x1][y1]=0;
        }
    }
    return false;

}
static void print(int[][]arr)
{
    for( int i=0;i<n;i++)
    {
        for( int j=0;j<n;j++)
        {
            System.out.print(arr[i][j]+" ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
   Scanner sc=new Scanner(System.in);
    int n=8;
```

```
        int arr[][]=new int[n][n];
        arr[0][0]=1;
       if( tour(arr,2,0,0))
          print(arr);
    }
}
```

OUTPUT:



Time complexity: O(8^(n*n))

Space complexity: O(n*n)

HAMILTONIAN CYCLE:

- Hamiltonian cycle is a path in a graph that visits each vertex exactly once and back to starting vertex.
- This program is to determine if a given graph is a hamiltonian cycle or not.
- This program assumes every vertex of the graph to be a part of hamiltonian path.

```
import java.util.*;

public class Main2 {
   static int node;
   static int[][] graph;
```

```
static int[] path;

static void displayCycle() {
    for (int i = 0; i < node; i++)
        System.out.print(path[i] + " ");
    System.out.println(path[0]);
}

static boolean isValid(int v, int k) {
    if (graph[path[k - 1]][v] == 0)
        return false;
    for (int i = 0; i < k; i++)
        if (path[i] == v)
            return false;
    return true;
}

static boolean cycleFound(int k) {
    if (k == node)
        return graph[path[k - 1]][path[0]] == 1;
    for (int v = 1; v < node; v++) {
        if (isValid(v, k)) {
            path[k] = v;
            if (cycleFound(k + 1))
                return true;
            path[k] = -1;
        }
    }
```

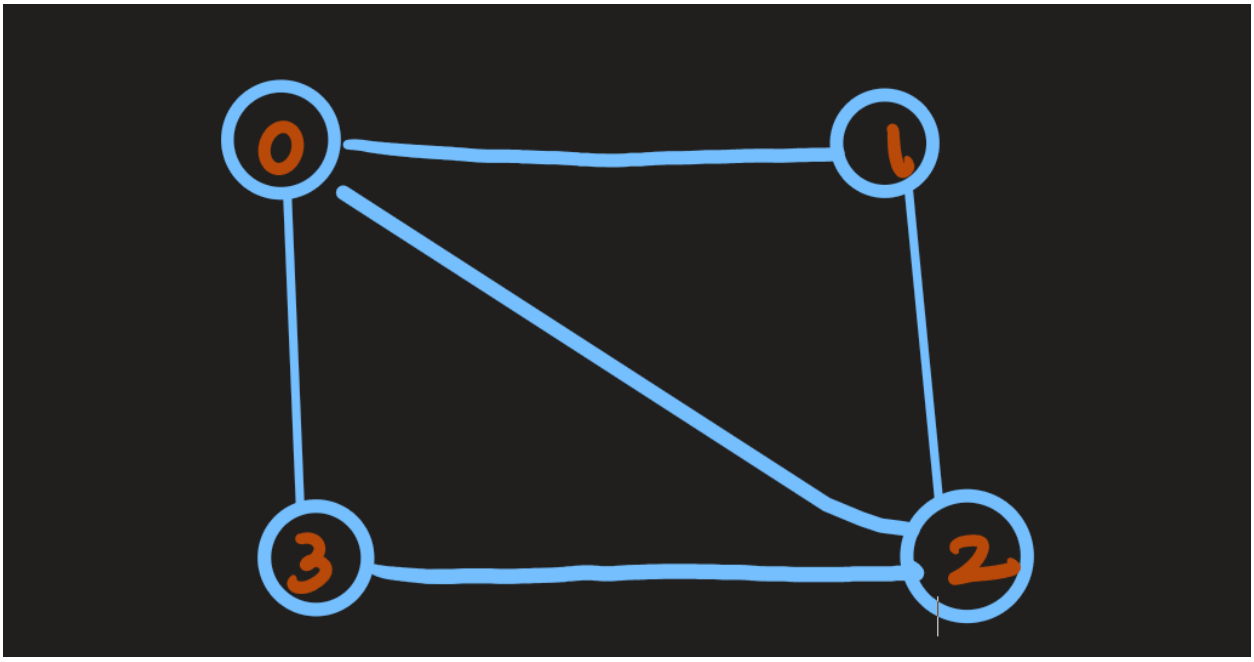
```
        return false;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of nodes: ");
        node = sc.nextInt();
        graph = new int[node][node];
        path = new int[node];
        System.out.println("Enter adjacency matrix:");
        for (int i = 0; i < node; i++)
            for (int j = 0; j < node; j++)
                graph[i][j] = sc.nextInt();

        path[0] = 0;
        for (int i = 1; i < node; i++)
            path[i] = -1;

        if (cycleFound(1))
            displayCycle();
        else
            System.out.println("No Hamiltonian Cycle found.");
    }
}
```

Time complexity: O(n!)

Space complexity: O(n^2).

Output:

```
input
Enter number of nodes: 4
Enter adjacency matrix:
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0
0 1 2 3 0
```



How to get different inputs from user:

Reading an Integer

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter an integer: ");
int intValue = sc.nextInt();
System.out.println("You entered: " + intValue);
```

Reading a Float

```
System.out.print("Enter a float: ");
```

```
float floatValue = sc.nextFloat();
System.out.println("You entered: " + floatValue);
```

Reading an Array

```
System.out.print("Enter size of array: ");
int size = sc.nextInt();
int[] array = new int[size];
System.out.println("Enter array elements:");
for (int i = 0; i < size; i++) {
   array[i] = sc.nextInt();
}
System.out.println("Array: " + Arrays.toString(array));
```

Reading a Matrix

```
System.out.print("Enter number of rows: ");
int rows = sc.nextInt();
System.out.print("Enter number of columns: ");
int cols = sc.nextInt();

int[][] matrix = new int[rows][cols];
System.out.println("Enter matrix elements:");
for (int i = 0; i < rows; i++) {
   for (int j = 0; j < cols; j++) {
      matrix[i][j] = sc.nextInt();
   }
}
System.out.println("Matrix:");
for (int i = 0; i < rows; i++) {
   System.out.println(Arrays.toString(matrix[i]));
```

```
}
Reading a Binary Number
System.out.print("Enter a binary number: ");

String binaryStr = sc.next();
int decimalValue = Integer.parseInt(binaryStr, 2); // Convert binary to decimal
System.out.println("Binary: " + binaryStr + ", Decimal: " + decimalValue);
```

Reading a String

```
System.out.print("Enter a string: ");
String str = sc.nextLine(); // Reads a whole line including spaces
System.out.println("You entered: " + str);
```

Reading a Character Array

```
System.out.print("Enter a word: ");
String word = sc.next(); // Reads a single word (no spaces)
char[] charArray = word.toCharArray();
System.out.println("Character Array: " + Arrays.toString(charArray));
```

Handling Comma-Separated Input:

```
System.out.print("Enter numbers separated by commas: ");
String input = sc.next(); // Reads the input as a single string
String[] parts = input.split(","); // Splits by commas
int[] numbers = new int[parts.length];

// Convert each part to an integer
for (int i = 0; i < parts.length; i++) {
    numbers[i] = Integer.parseInt(parts[i]);
}
```

```
System.out.println("Array of numbers: " + Arrays.toString(numbers));
```