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 = 1000000;
   static boolean arr[] = new boolean[N+1];

   static void simpleSieve() {
      Arrays.fill(arr, true);
      arr[0] = arr[1] = false;

      for (int i = 2; i * i <= N; i++) {
         if (arr[i]) {
            for (int j = i * i; j <= N; j += i) {
               arr[j] = false;
```

```
            }
        }
    }
}

static ArrayList<Integer> generatePrimes(int n) {
    ArrayList<Integer> al = new ArrayList<>();
    for (int i = 2; i <= n; i++) {
        if (arr[i]) {
            al.add(i);
        }
    }
    return al;
}

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

    simpleSieve();

    ArrayList<Integer> primes = generatePrimes(high);

    boolean[] dummy = new boolean[high - low + 1];
    Arrays.fill(dummy, true);


    for (int prime : primes) {
        int firstMultiple = (low / prime) * prime;
        if (firstMultiple < low) {
            firstMultiple += prime;
        }

        for (int j = Math.max(firstMultiple, prime * prime); j <= high; j += prime) {
            dummy[j - low] = false;
        }
    }
```

```
        for (int i = low; i <= high; i++) {
            if (dummy[i - low] && i != 1) {
                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 $\phi(n)$, 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 $\phi(n)$ 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, $\phi(p)=p-1$ for any prime ppp.
- **Multiplicative Property:**Euler's Totient function is multiplicative for coprime numbers, meaning if mmm and n are coprime, then $\phi(m\times n)=\phi(m)\times\phi(n)$.
- **Formula for $\phi(n)$:**For any integer n, the value of $\phi(n)$\phi(n) can be calculated using the formula: $\phi(n)=n\times(1-1/p1)\times(1-1/p2)\times\cdots\times(1-1/pk)$ where p1,p2,...,pk are the distinct prime factors of n.
- **Reduction of Non-Coprime Numbers**:The idea behind the formula is that for each prime factor phi of n, numbers that are multiples of phi are not coprime with n. The function reduces the count by eliminating these non-coprime numbers.
- **Applications:**
    - $\phi(n)$ is crucial in the RSA encryption algorithm, where the security relies on the difficulty of determining $\phi(n)$ for large composite numbers.
    - It's also used in problems involving modular inverses, where $\phi(n)$ 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:
    - $0 \leftrightarrow 0$
    - $1 \leftrightarrow 1$
    - $6 \leftrightarrow 9$
    - $8 \leftrightarrow 8$
    - $9 \leftrightarrow 6$
- **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: $x \equiv a1 \pmod{m1}x, x \equiv a2 \pmod{m2}x$ ..... $x \equiv ak \pmod{mk}x$
  - Here, m1,m2,...,mk are the moduli, and a1,a2,...,ak are the remainder.
- **Coprime Moduli:**
  - The theorem requires that the moduli m1,m2,...,mk be pairwise coprime, meaning the greatest common divisor (GCD) of any pair of moduli is 1 (i.e., $\gcd(mi,mj)=1$ for $i \neq j$).
  - When this condition is met, there exists a unique solution x modulo $M=m1 \times m2 \times \cdots \times mk$.
- **Constructive Solution:**
  - The CRT not only guarantees the existence of a solution but also provides a method to construct it:
    1. Compute the product $M=m1 \times m2 \times \cdots \times mk$.
    2. For each congruence, compute $Mi=M/mi$.
    3. Find the modular inverse of Mi modulo mi, denoted as yi, such that $Mi \times yi \equiv 1 \pmod{mi}M\_i$.
    4. The solution xxx is given by: $x=\sum i=1kai \times Mi \times yi \pmod{M}$.
- **Uniqueness:**
  - The solution x is unique modulo M. This means that any other solution will be congruent to x modulo M, so x 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);
  }
}
```

Euclidean Algorithm

Intuition:-

The **Euclidean Algorithm** is an efficient method for computing the greatest common divisor (GCD) of two integers. The GCD of two numbers is the largest positive integer that divides both numbers without leaving a remainder. The algorithm is based on the principle that the GCD of two numbers does not change if the larger number is replaced by its remainder when divided by the smaller number. This process is repeated until the remainder is zero, at which point the other number is the GCD.

Key Points:

- **Basic Idea:** The Euclidean Algorithm relies on the observation that if a>b, then:
  gcd(a,b)=gcd(b,a%b)
  The process continues until a%b=0, and the result will be the GCD of a and b.
- **Recursive and Iterative Approaches:** The algorithm can be implemented both recursively and iteratively:
  - In the recursive form, the algorithm continues calling itself with the smaller number and the remainder of the division until the remainder becomes zero.
  - In the iterative form, a loop replaces the recursive call, performing the same steps until the remainder becomes zero.
- **Efficiency:** The Euclidean Algorithm is very efficient, with a time complexity of O(log(min(a,b))). This is due to the fact that the remainder decreases in each step, which limits the number of iterations.
- **Handling Large Numbers:** The algorithm scales well even for large numbers, as each step significantly reduces the size of the numbers involved. This is why it is commonly used in applications involving cryptography, number theory, and computer science problems that involve large integers.
- **Properties of the GCD:**
  - If gcd(a,b)=1, then a and b are said to be coprime.

    - The Euclidean algorithm can also be extended to find solutions to Diophantine equations using the Extended Euclidean Algorithm.
- **Example:** To find the GCD of 252 and 105 using the Euclidean Algorithm:
    - Step 1: 252%105=42
    - Step 2: 105%42=21
    - Step 3: 42%21=0
    - Thus, the GCD of 252 and 105 is 21.
- **Applications:**
    - Cryptography: The Euclidean Algorithm is fundamental in algorithms like RSA for computing the GCD and modular inverses.
    - Number Theory: It is used to solve problems involving divisibility and Diophantine equations.
    - Simplifying Fractions: It helps in reducing fractions to their simplest form by dividing both the numerator and denominator by their GCD.

Iterative Code:

```
import java.util.*;
public class Main {
   public static int findGcd(int a, int b) {
      while(a !=0 && b != 0) {
         if(a > b) {
            a = a % b;
         }
         else {
            b = b % a;
         }
      }
      if(a == 0) {
         return b;
      }
      return a;
   }
```

```
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        int n1 = sc.nextInt();
        int n2 = sc.nextInt();
        System.out.println(findGcd(n1, n2));
    }
}
```

Recursive Code:

```
import java.util.*;
class Main {
    public static int gcd(int a, int b) {
        if (a == 0)
            return b;

        return gcd(b % a, a);
    }
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter two integers -");
        int n1 = scan.nextInt();
        int n2 = scan.nextInt();
        int g = gcd(n1, n2);
        System.out.println(g);
    }
}
```

Karatsuba Algorithm

Intuition:-

The **Karatsuba Algorithm** is an efficient method for multiplying large integers. It reduces the complexity of multiplying two nnn-digit numbers by breaking them down into smaller parts. Unlike the traditional long multiplication method, which has a time complexity of O(n^2), Karatsuba's algorithm reduces this complexity to O(n log 3 base 2), which is approximately O(n^1.585). This is achieved by recursively splitting numbers and performing fewer multiplications.

Key Points:

- **Divide-and-Conquer Strategy:** Karatsuba's algorithm splits two large numbers into smaller parts and multiplies those parts. Specifically, it breaks down each number into two halves:
  $X = x1 \cdot 10^{n/2} + x0, Y = y1 \cdot 10^{n/2} + y0$
  Where x1 and x0 represent the higher and lower halves of X, and similarly for Y.
- **Three Multiplications Instead of Four:** Traditional multiplication of these parts would require four separate multiplications:
  $X \cdot Y = (x1 \cdot y1) \cdot 10^n + ((x1 \cdot y0) + (x0 \cdot y1)) \cdot 10^{n/2} + (x0 \cdot y0)$
- **Karatsuba reduces this to only three multiplications:**
  $z0 = x0 \cdot y0$
  $z2 = x1 \cdot y1$
  $z1 = (x1 + x0)(y1 + y0) - z0 - z2$
  Thus, the result is:
  $X \cdot Y = z2 \cdot 10^n + z1 \cdot 10^{n/2} + z0$
- **Recursive Nature:** Each of the three products z0, z1, and z2 can be computed using the same Karatsuba method if the numbers are still large, making the algorithm recursive. The base case occurs when the numbers become small enough to use simple multiplication.
- **Efficiency:** By reducing the number of multiplicative steps from four to three, Karatsuba improves the time complexity. Traditional multiplication requires O(n^2) operations, but Karatsuba reduces this to approximately O(n^{1.585}), making it more efficient for large numbers.
- **Example:** Let's multiply X=1234 and Y=5678 using Karatsuba's algorithm:

- Split both numbers: $X=12 \cdot 10^2+34, Y=56 \cdot 10^2+78$
- Perform the three multiplications:
  $z0=34 \cdot 78=2652$
  $z2=12 \cdot 56=672$
  $z1=(12+34)(56+78)-z0-z2=46 \cdot 134-2652-672=6164$
- Combine the results: $X \cdot Y=672 \cdot 10^4+6164 \cdot 10^2+2652=7006652$
- Thus, $1234 \times 5678=7006652$.

- **Applications:**
  - Cryptography: Used in public-key algorithms like RSA where large integer multiplication is common.
  - Large Integer Arithmetic: In scientific computations requiring precise calculations with very large numbers.
  - Fast Multiplication Algorithms: Extends into faster algorithms like Toom-Cook and Schönhage-Strassen algorithms.

Code:

```
import java.util.*;
public class Main
{
   public static int karatsuba(int x, int y)
   {
      if (x < 10 || y < 10)
      {
         return x * y;
      }
      int m = Math.max(getNumDigits(x), getNumDigits(y));
      int halfM = m / 2;
      int powerOf10 = (int) Math.pow(10, halfM);
      int a = x / powerOf10;
      int b = x % powerOf10;
      int c = y / powerOf10;
      int d = y % powerOf10;
      int ac = karatsuba(a, c);
      int bd = karatsuba(b, d);
      int abcd = karatsuba(a + b, c + d);
      int result = ac * (int) Math.pow(10, 2 * halfM) + (abcd - ac - bd) * powerOf10 + bd;
      return result;
```

```
    }
    private static int getNumDigits(int x)
    {
        if (x == 0)
        {
            return 1;
        }
        int count = 0;
        while (x > 0)
        {
            count++;
            x /= 10;
        }
        return count;
    }
    public static void main(String[] args)
    {
        Scanner s1=new Scanner(System.in);
        int x = s1.nextInt();
        int y = s1.nextInt();
        int product = karatsuba(x, y);
        System.out.println(product);
    }
}
```

Longest Sequence of 1s after flip

Intuition:-

The problem of finding the **longest sequence of 1s after flipping** exactly one 0 to 1 in a binary array requires identifying how flipping a single 0 can maximize the length of a contiguous sequence of 1s. The idea is to find a 0 that, when flipped, merges two adjacent sequences of 1s into a longer sequence. The key challenge is efficiently determining how to merge the sequences by analyzing the array.

Key Points:

- **Divide into Blocks of 1s:** A binary array consists of blocks of consecutive 1s and 0s. The key is to analyze how flipping a single 0 can connect two blocks of 1s on either side.
- **Sliding Window Approach:** A sliding window approach can be used to traverse the array and keep track of sequences of 1s, handling each 0 and determining the possible length of the sequence if that 0 were flipped to 1.
- **Maintaining Left and Right Sequences:**
    - Maintain two variables to track the number of consecutive 1s on the left and right side of a 0.
    - When a 0 is encountered, check if flipping it can merge the left and right sequences of 1s.
    - Continue this process for each 0 in the array.
- **Edge Case Handling:**
    - If the array contains no 0s, return the length of the array.
    - If the array has only one 0, flipping it should result in the entire array being filled with 1s.
- **Example:** For the binary array: `[1, 1, 0, 1, 1, 1, 0, 1, 1]`
    - The first 0 is between two blocks of 1s ([1, 1] and [1, 1, 1]). Flipping the 0 merges them into a sequence of 6 consecutive 1s.
    - The second 0 is between two blocks of 1s ([1, 1, 1] and [1, 1]). Flipping the 0 merges them into another sequence of 6 consecutive 1s.
    - The maximum sequence length is 6.

- **Algorithm:**
    - Initialize variables to track the maximum length, the left 1s count, and the right 1s count.
    - Traverse through the array and when a 0 is encountered:
    - Calculate the potential new sequence length by summing the left and right 1s around the 0.
    - Track the maximum sequence length across all flips.
    - Return the maximum length found.
- **Applications:**
    - Data Compression: In algorithms that involve binary encoding and require manipulation of bit sequences.
    - Network Packet Analysis: Helps analyze patterns of transmitted or lost data in binary sequences.
    - Binary Image Processing: In tasks involving the identification of contiguous regions in binary images.

Code:

```
import java.util.Scanner;
class Main
{
   private static int findMaxConsecutiveOnes(int a, int k)
   {
      String str=Integer.toBinaryString(a);
      int maxOnes = Integer.MIN_VALUE;
      int numReplacements = 0;
      int windowStart = 0;
      for(int windowEnd = 0; windowEnd < str.length(); windowEnd++)
      {
         if(str.charAt(windowEnd) == '0')
         {
            numReplacements++;
         }
         while(numReplacements > k)
         {
            if(str.charAt(windowStart) == '0')
            {
               numReplacements--;
```

```
            }
            windowStart++;
          }
          maxOnes = Math.max(maxOnes, windowEnd-windowStart+1);
        }
        return maxOnes;
    }
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int result = findMaxConsecutiveOnes(n, 1);
        System.out.println(result);
    }
}
```

Swap Two Nibbles in a Byte

Intuition:-

Swapping two nibbles in a byte involves rearranging the first four bits (the high nibble) and the last four bits (the low nibble) of an 8-bit byte. A nibble consists of 4 bits, so the task is essentially about shifting these 4-bit groups within the byte. By isolating each nibble and then shifting them to their new positions, we can achieve the desired swap.

Key Points:

- **Nibble Definition:** In an 8-bit byte, the left half (high nibble) contains the most significant 4 bits, and the right half (low nibble) contains the least significant 4 bits.
- **Bitwise Manipulation:** To swap the nibbles, bitwise operations like shifting (<< and >>) and bit masking (&) are used to isolate and reposition the bits.
- **Masking:** Use bit masks to isolate the high nibble (11110000 or 0xF0) and the low nibble (00001111 or 0x0F).
- **Algorithm:**
    1. Isolate High Nibble:
    - Perform a bitwise AND with 0xF0 (binary: 11110000) to extract the high nibble.
    - Right shift this nibble by 4 bits to move it to the position of the low nibble.
    2. Isolate Low Nibble:
    - Perform a bitwise AND with 0x0F (binary: 00001111) to extract the low nibble.
    - Left shift this nibble by 4 bits to move it to the position of the high nibble.
    3. Combine Nibbles:
    - Use the bitwise OR operation to combine the two nibbles, which have now swapped positions.
    4. Return the Result:
    - The result is the byte with swapped nibbles.
- **Example:**

  For an input byte 10101100 (in binary), we would:

    - Isolate the high nibble 1010 and shift it to the right: 00001010.
    - Isolate the low nibble 1100 and shift it to the left: 11000000.
    - Combine them: 11000000 | 00001010 = 11001010.
- **Edge Case Handling:**
    - Input as 0: If the byte is 00000000, swapping the nibbles will still result in 00000000.
    - Maximum Value: For a byte with the maximum value 11111111, swapping the nibbles will also result in 11111111.
- **Applications:**
    - Data Encoding: Useful in data encoding schemes where nibbles need to be swapped for compression or encryption purposes.
    - Microcontroller Programming: In low-level hardware programming, swapping nibbles can be used for byte reordering when communicating between systems with different endian architectures.
    - Memory Optimization: Helps in reducing memory usage by rearranging data for better cache performance.

Code:

```
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      int num = scanner.nextInt();
      int swapNum;
      swapNum = ((num & 0x0F) << 4 | (num & 0xF0) >> 4);
      System.out.println( swapNum);
      scanner.close();
   }
}
```

Max Sum of Hourglass in a 2D Array

Intuition:-

An hourglass in a 2D array is a specific 3x3 pattern that resembles an hourglass shape:

a b c
  d
e f g

The problem of finding the maximum sum of an hourglass involves identifying all possible hourglasses in the 2D array, calculating their sums, and selecting the maximum. The array must have at least 3 rows and 3 columns to form an hourglass, and the task is to slide this hourglass pattern across the array, summing the values at the positions of the hourglass, and keeping track of the largest sum.

Key Points:

- **Hourglass Definition:** The hourglass pattern covers 7 elements in the 3x3 subarray. For each hourglass, the sum is calculated by summing the elements in the top row, middle of the middle row, and the bottom row.
- **Sliding Window:** A sliding window approach is used to traverse the 2D array. Starting from the top-left corner, we extract hourglass sums from all possible 3x3 subarrays and keep track of the maximum sum.
- **Boundary Conditions:** The hourglass can only form where the indices allow a 3x3 block, so for an array of dimensions n x m, valid hourglasses exist only within the bounds n - 2 and m - 2.
- **Algorithm:**
    1. Initialize Variables:
    - Use a variable maxSum to store the maximum hourglass sum found, initializing it to a very small value (e.g., INT_MIN).
    2. Traverse the Array:
    - Loop through the array using nested loops, such that the hourglass fits within the current bounds of the 2D array. The outer loop should run up to n - 2 (rows) and the inner loop up to m - 2 (columns).
    3. Calculate Hourglass Sum:

- For each valid starting point (i, j), extract the 7 values of the hourglass:

  a = arr[i][j], b = arr[i][j+1], c = arr[i][j+2]

  d = arr[i+1][j+1]

  e = arr[i+2][j], f = arr[i+2][j+1], g = arr[i+2][j+2]

4. Calculate the sum:

   hourglass_sum = a + b + c + d + e + f + g.

5. Update Maximum:

- Compare the hourglass_sum to maxSum, and update maxSum if the current sum is larger.

6. Return the Maximum:

- Once all hourglasses have been considered, return maxSum.

- **Example:**
  - Consider the following 2D array:

    1 1 1 0 0 0
    0 1 0 0 0 0
    1 1 1 0 0 0
    0 0 2 4 4 0
    0 0 0 2 0 0
    0 0 1 2 4 0

  - There are multiple hourglasses, but the one with the maximum sum is:

    2 4 4
    2
    1 2 4

  - The sum is 2 + 4 + 4 + 2 + 1 + 2 + 4 = 19.
- **Applications:**
  - Image Processing: This problem can be used to detect certain shapes in an image represented by a binary or grayscale matrix.
  - Terrain Analysis: Identifying patterns in elevation data in geospatial analysis.
  - Heatmaps/Grids: Can be applied to analyze heat maps or 2D grids where certain patterns are significant.

Code with i&j are 0:

```
import java.util.Scanner;

class Main {
   static int findMaxSum(int[][] mat, int row, int col) {

      if (row < 3 || col < 3) {
         System.out.println("Not possible to calculate hourglass sum for given matrix
size.");
         return Integer.MIN_VALUE;
      }

      int max_sum = Integer.MIN_VALUE;

      for (int i = 0; i < row - 2; i++) {
         for (int j = 0; j < col - 2; j++) {
            int 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;
   }

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

      int row = input.nextInt();

      int col = input.nextInt();
      int[][] mat = new int[row][col];
      for (int i = 0; i < row; i++) {
         for (int j = 0; j < col; j++) {
            mat[i][j] = input.nextInt();
         }
      }
```

```
        int res = findMaxSum(mat, row, col);
        if (res != Integer.MIN_VALUE) {
            System.out.println(res);
        }
    }
}
```

Code with i&j are 1:

```
import java.util.Scanner;

public class Main {

    public static int MaxSum(int[][] matrix) {
        int curr = 0, max = Integer.MIN_VALUE;
        int m = matrix.length;
        int n = matrix[0].length;
        for (int i = 1; i < m - 1; i++) {
            for (int j = 1; j < n - 1; j++) {
                curr = (matrix[i - 1][j - 1] + matrix[i - 1][j] + matrix[i - 1][j + 1]
                        + matrix[i][j]
                        + matrix[i + 1][j - 1] + matrix[i + 1][j] + matrix[i + 1][j + 1]);
                max = Math.max(max, curr);
            }
        }
        return max;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int rows = scanner.nextInt();
        int cols = scanner.nextInt();

        int[][] matrix = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = scanner.nextInt();
            }
```

```
        }
        int result = MaxSum(matrix);
        System.out.println(result);

        scanner.close();
    }
}
```

BlockSwap Algorithm

Intuition:-

The **Block Swap Algorithm** is used for rotating an array by d positions. This algorithm works by dividing the array into two parts, swapping them in blocks, and then recursively adjusting the position of the blocks until the array is rotated.

Key Points:

- **Array Rotation:** The goal is to rotate an array of size n by d elements to the left. This means the elements from index d to n-1 will shift left, and the first d elements will move to the end.
- **Block Swap:** The array is split into two parts: one of size d (first block) and the other of size n - d (second block). These two blocks are swapped in parts until the entire array is rotated.
- **Algorithm:**

  1. Initialize Variables:

     - Let A be the array of size n.
     - Set the rotation size d (where d <= n).
     - Divide the array into two parts:
       - First block: A[0...d-1]
       - Second block: A[d...n-1]

  2. Base Condition:

     - If d == 0 or d == n, no rotation is needed; simply return the array as it is.

  3. Recursive Swapping:

     - Case 1: If the size of the first block d is equal to the size of the second block n - d, swap them directly.
     - Case 2: If d is smaller than n - d, recursively swap the first block with the corresponding portion of the second block.
     - Case 3: If d is larger than n - d, recursively swap the second block with the corresponding portion of the first block.

4. Swap Function:

    - For each swap, exchange elements between the two blocks of the array, ensuring that the correct portion is moved in each step.

- **Steps:**
    - Divide the array A into two blocks.
    - If the size of the first block (d) matches the second block (n-d), swap them.
    - If the sizes are unequal:
        - If d < n - d, swap A[0...d-1] with A[n-d...n-1] and recursively rotate the remaining part of the array.
        - If d > n - d, swap A[0...n-d-1] with A[d...n-1] and recursively rotate the remaining part of the array.
    - Continue this process until the entire array is rotated.
- **Example:**
    - Consider an array A = [1, 2, 3, 4, 5, 6, 7] and d = 3. We want to rotate the array by 3 positions to the left.
    - Divide the array into two parts:
    - First block: [1, 2, 3]
    - Second block: [4, 5, 6, 7]
    - Since the first block's size is smaller than the second block's size, swap the first block with a corresponding portion of the second block:
        - [4, 5, 6, 1, 2, 3, 7]
    - Now, we need to rotate the remaining portion [4, 5, 6] by 3 positions, which involves swapping it:
        - [1, 2, 3, 4, 5, 6, 7] (final rotated array).
- **Applications**:
    - Data Shifting: Useful in scenarios where data needs to be shifted cyclically, such as rotating elements in buffer arrays or cyclic shift operations.
    - Memory Management: Can be used to optimize in-place array rotation without using extra memory.
    - Circular Buffers: In systems where buffers are cyclic in nature, the block swap method helps in managing data movement efficiently.

Recursive Code:

```
import java.util.Scanner;

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

   public static void leftRotateRec(int arr[], int i, int d, int n) {

      if (d == 0 || d == n)
         return;

      if (n - d == d) {
         swap(arr, i, n - d + i, d);
         return;
      }

      if (d < n - d) {
         swap(arr, i, n - d + i, d);
         leftRotateRec(arr, i, d, n - d);
      }
      else {
         swap(arr, i, d, n - d);
         leftRotateRec(arr, n - d + i, 2 * d - n, d); // This is tricky
      }
   }
   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);

        int n = scanner.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        int d = scanner.nextInt();
        leftRotate(arr, d, n);
        printArray(arr, n);
        scanner.close();
    }
}
```

Iterative Code Left Rotate:

```
import java.util.*;

class Main{
    static int[] blockSwap(int arr[],int k, int n){
        if(k==0 || k==n)
            return arr;
        if(k>n)
            k%=n;
        int res[]=new int[n];
        for(int i=0;i<n;i++){
            res[i]=arr[(i+k)%n];
        }
        return res;
    }
    public static void main (String[] args) {
        Scanner sc=new Scanner(System.in);
```

```
        int n=sc.nextInt();
        int k=sc.nextInt();
        int arr[]=new int[n];
        for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
        }
        arr=blockSwap(arr,k,n);
        for(int i=0;i<n;i++){
            System.out.print(arr[i]+" ");
        }
    }
}
```

Iterative Code Right Rotate:

```
import java.util.*;

class Main{
    static int[] blockSwap(int arr[],int k, int n){
        if(k==0 || k==n)
            return arr;
        if(k>n)
            k%=n;
        int res[]=new int[n];
        for(int i=0;i<n;i++){
            res[i]=arr[(n-k+i)%n];
        }
        return res;
    }
    public static void main (String[] args) {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int k=sc.nextInt();
        int arr[]=new int[n];
        for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
        }
        arr=blockSwap(arr,k,n);
        for(int i=0;i<n;i++){
```

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

Max Product Subarray

Intuition:-

The **Maximum Product Subarray** problem involves finding a contiguous subarray within a 1D array that has the largest product. Unlike the maximum sum subarray, where we simply accumulate the sum of elements, the product of elements can fluctuate dramatically, especially when negative numbers or zeros are involved.

Key Points:

- **Product Behavior:** When calculating the product of a subarray, multiplying by a negative number flips the sign of the product, and multiplying by zero resets the product to zero.
- **Two Key Variables:**
    - maxProduct: Tracks the maximum product encountered so far.
    - minProduct: Tracks the minimum product encountered so far (because a negative number can turn the minimum product into the maximum when multiplied).
- **Dynamic Subarrays:** As you iterate through the array, you keep track of both the maximum and minimum products for the subarray that ends at the current index, considering both positive and negative numbers.
- **Algorithm**:

  1. Initialize Variables:

    - maxProduct to store the maximum product encountered so far.
    - minProduct to store the minimum product encountered so far (to handle negative numbers).
    - globalMax to store the final result (initialized to the first element of the array).

  2. Traverse the Array:

    - Loop through the array starting from the first element. For each element:
    - Update maxProduct and minProduct by considering the current element alone, the current element multiplied by the

previous maxProduct, or the current element multiplied by the previous minProduct.
    - Since multiplication by a negative number flips the product, ensure that after updating maxProduct, you also update minProduct.
    - Keep track of the overall globalMax, which stores the maximum product found so far.

3. Return the Maximum Product:

    - After traversing the entire array, return globalMax as the maximum product of any subarray.
- **Example**:
    - Consider the array:
        - [2, 3, -2, 4]
    - Step-by-Step Calculation:
    - First element (2):
        - maxProduct = 2, minProduct = 2, globalMax = 2
    - Second element (3):
        - maxProduct = 2 * 3 = 6, minProduct = 3, globalMax = 6
    - Third element (-2):
        - maxProduct = max(-2, -2 * 6) = -2, minProduct = min(-2, -2 * 6) = -12, globalMax = 6
    - Fourth element (4):
        - maxProduct = max(4, 4 * -2) = 4, minProduct = min(4, 4 * -12) = -48, globalMax = 6
    - Thus, the maximum product subarray is 6.
- **Applications**:
    - Stock Market Analysis: Can be used to analyze stock price movements, where finding the largest product of price changes can indicate the most profitable trading period.
    - Signal Processing: Applied in time series analysis where maximizing the product of consecutive signals is critical.
    - Genetic Algorithms: This problem can appear in computational biology, where maximizing certain subsequences of genetic code is important.

Kadane's Code:

```
import java.util.Scanner;

public class Main {
   static int maxProductSubArray(int arr[]) {
      int maxProduct = arr[0], minProduct = arr[0], globalMax = arr[0];

      for (int i = 1; i < arr.length; i++) {
         int temp = Math.max(arr[i], Math.max(maxProduct * arr[i], minProduct * arr[i]));
         minProduct = Math.min(arr[i], Math.min(maxProduct * arr[i], minProduct * arr[i]));
         maxProduct = temp;

         globalMax = Math.max(globalMax, maxProduct);
      }
      return globalMax;
   }

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      int[] nums = new int[n];
      for (int i = 0; i < n; i++) {
         nums[i] = sc.nextInt();
      }
      int answer = maxProductSubArray(nums);
      System.out.println(answer);

      sc.close();
   }
}
```

Alternative Code:

```
import java.util.Scanner;

public class Main {
    public static int maxProductSubArray(int[] arr) {
        int n = arr.length;
        int pre = 1, suff = 1;
        int ans = Integer.MIN_VALUE;

        for (int i = 0; i < n; i++) {
            if (pre == 0) pre = 1;
            if (suff == 0) suff = 1;

            pre *= arr[i];
            suff *= arr[n - i - 1];
            ans = Math.max(ans, Math.max(pre, suff));
        }
        return ans;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        int answer = maxProductSubArray(arr);
        System.out.println(answer);

        sc.close();
    }
}
```

Maximum Equilibrium Sum

Intuition:-

The **Maximum Equilibrium Sum** problem involves finding an index in a 1D array where the sum of elements on the left side is equal to the sum of elements on the right side, known as the equilibrium index. The goal is to maximize this sum, i.e., find the equilibrium index that yields the largest sum of either side.

Unlike the maximum subarray sum, this problem focuses on balancing the left and right side sums and tracking the largest possible equilibrium sum.

Key Points:

- **Equilibrium Index:**
  - An index i is an equilibrium index if:
    sum(arr[0] + arr[1] + ... + arr[i]) == sum(arr[i] + arr[i+1] + ... + arr[n-1])
  - The sum of elements to the left of index i must be equal to the sum of elements to the right.
- **Two Key Variables:**
  - Prefix Sum: Keeps track of the sum of elements from the start of the array up to the current index.
  - Total Sum: Stores the sum of all elements in the array, and it is updated dynamically as the algorithm checks for equilibrium.
- **Dynamic Updates:**
  - As you iterate through the array, dynamically update the total sum and prefix sum.
  - For each element, check if the current prefix sum equals the remaining sum on the right.
  - If so, update the maximum equilibrium sum found so far.
- **Algorithm:**

  1. Initialize Variables:

     - prefixSum: Starts at 0, representing the sum of elements on the left.
     - totalSum: Initially set to the sum of the entire array.

- maxEquilibriumSum: Stores the maximum equilibrium sum found so far (initially negative infinity).

2. Traverse the Array:

- Loop through each element of the array. For each element:
  - Add the current element to the prefixSum.
  - Subtract the current element from totalSum to represent the right-side sum.
  - If the prefixSum equals the remaining totalSum, update the maxEquilibriumSum to track the maximum equilibrium sum found so far.

3. Return the Maximum Equilibrium Sum:

- After iterating through the array, return the maxEquilibriumSum. If no equilibrium index is found, return an appropriate message.

- **Example:**
  - Consider the array:
    - [3, 1, 2, 5, 6, 3, 2]
- **Step-by-Step Calculation**:
    - Total sum: 3 + 1 + 2 + 5 + 6 + 3 + 2 = 22
  - First element (3):
    - prefixSum = 3, remaining sum = 22 - 3 = 19
    - No equilibrium.
  - Second element (1):
    - prefixSum = 3 + 1 = 4, remaining sum = 19 - 1 = 18
    - No equilibrium.
  - Continue for all elements until the appropriate equilibrium index is found or all elements are checked.
  - Thus, the **maximum equilibrium sum** is the largest value found at an equilibrium index, or no equilibrium is returned if none exists.
- **Applications:**
  - Financial Analysis: Can be used to find balanced points in a sequence of financial transactions or investments.

- Supply Chain Management: Helps in finding balance points in logistics and production chains.
- Data Analysis: Useful in identifying equilibrium points in various datasets where equal left and right contributions matter.

Code:

```
import java.util.*;

class Main {
        static int findMaxSum(int arr[], int n){
           int sum=0;
                for (int num : arr) {
        sum += num;
      }

                int prefix_sum = 0;
                int res = Integer.MIN_VALUE;

                for (int i = 0; i < n; i++)
                {
                        prefix_sum += arr[i];

                        if (prefix_sum == sum)
                                res = Math.max(res, prefix_sum);
                        sum -= arr[i];
                }
                return res;
        }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      int n = scanner.nextInt();
      int[] arr = new int[n];
      for (int i = 0; i < n; i++) {
         arr[i] = scanner.nextInt();
      }
      int result = findMaxSum(arr,n);
      if (result != Integer.MIN_VALUE) {
```

```
            System.out.println(result);
        }
        else {
            System.out.println("No equilibrium sum found.");
        }
    scanner.close();
            }
}
```

Leaders in Array

Intuition:-

In the **Leaders in an Array** problem, a leader is defined as an element of the array that is greater than or equal to all the elements to its right. The rightmost element of the array is always considered a leader since there are no elements to its right.

Key Points:

- **Rightmost Leader:** The last element of the array is always a leader.
- **Backward Traversal:** By traversing the array from right to left, we can efficiently find leaders by maintaining the maximum element encountered so far. If an element is greater than or equal to this maximum, it is a leader.
- **Optimal Approach:** A single pass from right to left ensures that we find all leaders in O(n) time.
- **Algorithm:**

  1. Initialize Variables:

     - Start from the last element (rightmost) and consider it as the first leader.
     - Keep track of the maximum element encountered so far (maxFromRight).

  2. Traverse the Array:

     - Move from the second last element to the first, checking if the current element is greater than or equal to maxFromRight.
     - If it is, mark it as a leader and update maxFromRight to the current element.

  3. Return the Leaders:

     - Collect the leaders found during the traversal and return them.
- **Example:**
  - Consider the array: [16, 17, 4, 3, 5, 2]

- **Step-by-Step Calculation:**
    - Start from the rightmost element: 2 (leader).
    - Move left to 5: 5 >= 2, so 5 is a leader.
    - Move left to 3: 3 < 5, so not a leader.
    - Move left to 4: 4 < 5, so not a leader.
    - Move left to 17: 17 > 5, so 17 is a leader.
    - Move left to 16: 16 < 17, so not a leader.
  - Thus, the leaders in the array are [17, 5, 2].
- **Applications**:
  - Stock Market Analysis: Identify moments where stock prices rise compared to future prices.
  - Competitive Gaming: Analyze winning strategies based on moves or scores that outperform subsequent actions.
  - Peak Detection in Data: Used in time series analysis to find moments of peak performance or activity.

Code:

```
import java.util.*;

public class Main {
   public static ArrayList<Integer> findLeaders(int[] arr) {
      ArrayList<Integer> leaders = new ArrayList<>();
      int n = arr.length;
      int maxFromRight = arr[n - 1];
      leaders.add(maxFromRight);
      for (int i = n - 2; i >= 0; i--) {
         if (arr[i] >= maxFromRight) {
            leaders.add(arr[i]);
            maxFromRight = arr[i];
         }
      }
      Collections.reverse(leaders);;
      return leaders;
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      int n = scanner.nextInt();
      int[] arr = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        ArrayList<Integer> leaders = findLeaders(arr);
        for(int x:leaders){
            System.out.print(x+" ");
        }
        scanner.close();
    }
}
```

Majority Element in an Array

Intuition:-

The **Majority Element** problem involves finding an element in an array that appears more than ⌊n/2⌋ times, where n is the total number of elements in the array. In other words, the majority element is the element that appears in more than half of the positions in the array.

Key Points:

- **Majority Element Definition:**
    - An element is considered the majority element if it appears more than ⌊n/2⌋ times in the array, where n is the total size of the array.
    - For example, in an array of size 7, the majority element must appear at least 4 times.
- **Boyer-Moore Voting Algorithm:**
    - The Boyer-Moore algorithm is an efficient way to solve the majority element problem in linear time (O(n)) with constant space (O(1)).
    - It works by maintaining two variables:
        - Candidate: A potential majority element.
        - Count: The number of votes the candidate has at the current stage of the iteration.
    - As we iterate through the array, we update these two variables in a way that ensures that if a majority element exists, it will be the candidate when the iteration is complete.
- **Algorithm:**

    1. Step 1: Find a Candidate:

    - Initialize candidate to None and count to 0.
    - Traverse through the array:
        - If count == 0, set the current element as the candidate and set count = 1.
        - If the current element is the same as the candidate, increment count.
        - Otherwise, decrement count.

- After the first pass, the candidate will be the potential majority element (if one exists).

2. Step 2: Verify the Candidate:

- After finding the candidate, make a second pass through the array to verify that the candidate appears more than $\lfloor n/2 \rfloor$ times.
- Count the occurrences of the candidate. If its count is greater than $\lfloor n/2 \rfloor$, return the candidate. Otherwise, there is no majority element.

- **Example:**
  - Consider the array:
    - [2, 2, 1, 1, 2, 2, 2]
- **Step-by-Step Calculation:**

First Pass (Finding the Candidate):

- Initialize candidate = None and count = 0.
- Traverse the array:
  - First element: 2. count = 0, so set candidate = 2 and count = 1.
  - Second element: 2. candidate = 2, so increment count to 2.
  - Third element: 1. candidate != 1, so decrement count to 1.
  - Fourth element: 1. candidate != 1, so decrement count to 0.
  - Fifth element: 2. count = 0, so set candidate = 2 and count = 1.
  - Sixth element: 2. candidate = 2, so increment count to 2.
  - Seventh element: 2. candidate = 2, so increment count to 3.

After the first pass, the candidate is 2.

- Second Pass (Verifying the Candidate):

Count the occurrences of 2 in the array:

- 2 appears 5 times.
- Since 5 is greater than $\lfloor 7/2 \rfloor = 3$, 2 is the majority element.

- **Applications:**
    - Voting Systems: Determine the candidate that gets the majority of votes.
    - Network Communication: Identify the most frequent packet type in a stream of network data.
    - Fault Tolerance: In distributed systems, the majority element can help in consensus protocols for error detection and correction.

Code:

```
import java.util.*;

class Main {
   static int majorityElement(int[] nums) {
      int candidate = 0, count = 0;

      for (int num : nums) {
         if (count == 0) {
            candidate = num;
         }
         count += (num == candidate) ? 1 : -1;
      }

      count = 0;
      for (int num : nums) {
         if (num == candidate) {
            count++;
         }
      }

      if (count > nums.length / 2) {
         return candidate;
      } else {
         return -1;
      }
   }

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

```
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        int result = majorityElement(arr);
        if (result != -1) {
            System.out.println(result);
        } else {
            System.out.println("No Majority Element.");
        }
        scanner.close();
    }
}
```

SelectionSort Algorithm

Intuition:-

Selection Sort is a simple comparison-based sorting algorithm. In each iteration, it selects the smallest (or largest) element from the unsorted portion of the array and places it in its correct position in the sorted portion. The idea is to divide the array into two parts: the sorted portion on the left and the unsorted portion on the right. The sorted portion grows as the algorithm proceeds.

Key Points:

- **Selection Process:** In each pass, find the smallest (or largest) element from the unsorted portion.
- **Swap:** Once the smallest element is found, swap it with the first element of the unsorted portion.
- **In-Place Sorting:** Selection Sort works in-place and requires no extra space beyond the input array.
- **O(n²) Time Complexity:** Due to nested loops, the algorithm has O(n²) time complexity, making it inefficient for large datasets but easy to understand and implement.
- **Algorithm:**

  1. Initialize Variables:

     - Divide the array into a sorted and unsorted portion.
     - Set the sorted portion initially to be empty and the unsorted portion as the entire array.

  2. Traverse the Array:

     - For each element in the array, find the smallest element in the unsorted portion.
     - Swap the smallest element with the first unsorted element to move it to the sorted portion.

3. Repeat:

    - Continue this process, shrinking the unsorted portion and growing the sorted portion, until all elements are sorted.

4. Return the Sorted Array.

- **Step-by-Step Example:**
    - Consider the array: [29, 10, 14, 37, 13]
    - First Pass:
        - The smallest element in [29, 10, 14, 37, 13] is 10.
        - Swap 10 with 29, resulting in the array: [10, 29, 14, 37, 13].
    - Second Pass:
        - The smallest element in [29, 14, 37, 13] is 13.
        - Swap 13 with 29, resulting in the array: [10, 13, 14, 37, 29].
    - Third Pass:
        - The smallest element in [14, 37, 29] is 14.
        - No swap needed as 14 is already in the correct place.
    - Fourth Pass:
        - The smallest element in [37, 29] is 29.
        - Swap 29 with 37, resulting in the final sorted array: [10, 13, 14, 29, 37].
- **Applications:**
    - Educational Purposes: Selection Sort is often used to teach sorting algorithms due to its simplicity.
    - Small Data Sets: Efficient for sorting small arrays where simplicity is more important than speed.
    - Hardware-Level Sorting: Due to its in-place nature, Selection Sort is sometimes used in embedded systems with limited memory.

Code:

```
import java.util.*;

class Main{
   static int[] selectionSort(int arr[], int n) {
      for (int i = 0; i < n - 1; i++) {
         int min = i;
         for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min]) {
               min = j;
            }
         }

         int temp = arr[min];
         arr[min] = arr[i];
         arr[i] = temp;
      }
      return arr;
   }
   public static void main(String args[]) {
      Scanner sc=new Scanner(System.in);
      int n=sc.nextInt();
      int arr[]=new int[n];
      for(int i=0;i<n;i++){
         arr[i]=sc.nextInt();
      }
      arr=selectionSort(arr, n);
      for (int i = 0; i < n; i++) {
         System.out.print(arr[i] + " ");
      }
   }
}
```

QuickSort Algorithm

Intuition:-

QuickSort is a highly efficient, comparison-based, divide-and-conquer sorting algorithm. It works by selecting a "pivot" element and partitioning the array such that all elements less than the pivot are on its left and all elements greater than the pivot are on its right. This process is repeated recursively for the subarrays. QuickSort is popular for its average-case time complexity of O(n log n).

Key Points:

- **Pivot Selection:** The pivot can be chosen in several ways (e.g., first element, last element, random, or median). A good pivot ensures balanced partitioning.
- **Partitioning:** Divide the array into two parts—one with elements smaller than the pivot and one with elements larger. This allows sorting to be done in-place.
- **Recursion:** Recursively apply the same process to the left and right subarrays until the entire array is sorted.
- **Average Time Complexity:** O(n log n), but in the worst case (already sorted or poorly partitioned array), it can degrade to $O(n^2)$.
- **Algorithm:**
    - Choose a Pivot: Select a pivot element from the array. For simplicity, we can choose the last element as the pivot.
    - Partition the Array: Rearrange the array so that elements smaller than the pivot are on the left and elements larger than the pivot are on the right.
    - Recursively Sort Subarrays: Apply the same logic recursively to the left and right subarrays until all subarrays have only one element.
    - Base Case: Arrays with one or zero elements are inherently sorted, which serves as the base case for recursion.
- **Example:**
    - Consider the array: [10, 80, 30, 90, 40, 50, 70]
    - Choose Pivot:
        - Choose 70 as the pivot (last element).

    - Partition:
        - Rearrange the array such that elements less than 70 are on the left and greater are on the right.
    - After partitioning: [10, 30, 40, 50, 70, 90, 80]
        - Pivot 70 is in its correct place at index 4.
    - Recursion:
        - Apply QuickSort on the left subarray [10, 30, 40, 50] and the right subarray [90, 80].
    - Repeat:
        - For [10, 30, 40, 50], choose 50 as the pivot and partition. After partitioning: [10, 30, 40, 50].
        - For [90, 80], choose 80 as the pivot and partition. After partitioning: [80, 90].
    - Final Sorted Array: The final sorted array is [10, 30, 40, 50, 70, 80, 90].
- **Applications**:
    - General-Purpose Sorting: QuickSort is often the default choice in many programming languages for its efficiency on average.
    - Databases and File Systems: Used in systems where fast, in-place sorting is required.
    - Memory-Constrained Environments: QuickSort requires minimal additional memory due to its in-place partitioning mechanism.

Code:

```
import java.util.*;

class Main {
   static int partition(int[] arr, int low, int high) {
      int pivot = arr[low];
      int i = low;
      int j = high;

      while (i < j) {
         while (arr[i] <= pivot && i <= high - 1) {
            i++;
         }
```

```
            while (arr[j] > pivot && j >= low + 1) {
                j--;
            }
            if(i<j){
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[low];
        arr[low] = arr[j];
        arr[j] = temp;

        return j;
    }
    static int[] quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivot = partition(arr, low, high);
            quickSort(arr, low, pivot - 1);
            quickSort(arr, pivot + 1, high);
        }
        return arr;
    }

    public static void main(String args[]) {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int arr[]=new int[n];
        for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
        }
        arr = quickSort(arr, 0, n-1);

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

Lexicographically First Palindromic String

Intuition:-

We want to form the possible palindrome in **lexicographic order** using a given set of characters in a String. A palindrome reads the same forward and backward, so half of the palindrome can determine its lexicographical order.

To form a valid palindrome:

1. If **all characters appear an even number of times**, we can simply divide them into two halves and mirror them to form a palindrome.
2. If **there is one character that appears an odd number of times**, that character will be placed in the center, while the rest of the characters are split evenly between the two halves.

The goal is to construct this palindrome while keeping the left half of the string lexicographically smallest.

Key Points:

- **Frequency counting:** First, count the occurrences of each character in the string.
- **Half Construction:** Use the characters with even frequencies to construct the first half of the palindrome.
- **Middle Character:** If there's one character with an odd frequency, it will be placed in the center of the palindrome.
- **Mirroring:** Once the first half is constructed, the second half is simply the reverse of the first half.
- **Algorithm:**
    - Step 1: Count the frequency of each character in the given string.
    - Step 2: Check if more than one character has an odd frequency. If yes, forming a palindrome is impossible.
    - Step 3: Form the first half of the palindrome using characters that have even frequencies.
    - Step 4: If any character has an odd frequency, place that character in the middle.
    - Step 5: Mirror the first half to complete the palindrome.

- **Example:**

  1. Let's take the input string: "aabbccdde"

     - Step 1: Frequency count:
       - a: 2, b: 2, c: 2, d: 2, e: 1
     - Step 2: Only e has an odd frequency, so it will be the center of the palindrome.
     - Step 3: Construct the first half using half of the characters with even frequencies:
       - First half: "abcd"
     - Step 4: Place the odd-frequency character in the middle:
       - Middle character: "e"
     - Step 5: Mirror the first half to get the second half:
       - Second half: "dcba"
     - Resulting Palindrome: The lexicographically first palindrome is "abcdedcba".
- **Applications**:
  - **Palindrome Generation:** This approach can be used in applications where the smallest palindromic arrangement of characters is needed, such as certain string manipulation problems or challenges.
  - **Lexicographic Order:** It ensures the palindrome is the smallest possible lexicographically, which is useful in sorting-based applications or optimizations.

Code:

```
import java.util.*;

class Main
{
   static String makePalindrome(TreeMap<Character, Integer> hm)
   {
      int count = 0;
      String  prefix = "", oddStr = "";
      for(Map.Entry<Character, Integer> e: hm.entrySet()){
         int itr = e.getValue()/2;
```

```
            for(int i=0;i<itr;i++)
                prefix = prefix + e.getKey();

            if(e.getValue() % 2 == 1)
            {
                oddStr += e.getKey();
                count++;
            }

            if(count > 1)
                return "Can't make a palindrome";
        }

        String suffix = new StringBuilder(prefix).reverse().toString();
        return prefix + oddStr + suffix;
    }

        public static void main(String[] args) {
            Scanner sc=new Scanner(System.in);
            String s=sc.nextLine();
            char ch[] = s.toCharArray();

            TreeMap<Character, Integer> hm = new TreeMap();
        for(char ele: ch)
            hm.put(ele, hm.getOrDefault(ele,0)+1);

            System.out.println(makePalindrome(hm));
        }
}
```

Move Hyphens to Beginning

Intuition:-

The problem can be solved by counting the number of hyphens and then appending them at the beginning of the string while keeping the rest of the characters in the same order.

Key Points:

- Separate the hyphens and the non-hyphen characters.
- Append the hyphens at the beginning and retain the relative order of the remaining characters.
- **Problem Statement:**
  You are given a string that contains lowercase alphabets and hyphens ('-'). Your task is to move all the hyphens to the beginning of the string while keeping the relative order of the alphabets unchanged.
- **Input:**
  - A string S consisting of lowercase alphabets and hyphens.
- **Output:**
  - A string with all hyphens moved to the beginning, followed by the original order of alphabets.
- **Example:**

  1. Input: "move-hyphens-to-beginning"
     - Output: "----movehyphenstobeginning"
  2. Input: "a-b-c-d-e"
     - Output: "-----abcde"
  3. Input: "--hello--world-"
     - Output: "-----helloworld"

Code:

```
import java.util.*;

class Main{
   static String moveHyphens(String str){
   String ans1 = "";
      for (int i = 0; i < str.length(); i++) {
         if (str.charAt(i) == '-') {
            ans1 = '-' + ans1;
         }
         else {
            ans1 += str.charAt(i);
         }
      }
      return ans1;
   }
   public static void main(String[] args){
      Scanner sc=new Scanner(System.in);
      String str = sc.nextLine();
      System.out.println(moveHyphens(str));
   }
}
```

Weighted Substring

Intuition:-

The problem requires identifying all unique substrings of string P whose sum of character weights (derived from string Q) is less than or equal to a given value K. We need to iterate through all possible substrings of P, calculate their weight by mapping the characters to the values in Q, and count the number of unique substrings that satisfy the condition.

Key Points:

- **Character Weights:** The string Q provides the weight of each English letter, where Q[i] is the weight of the letter corresponding to its position in the alphabet (i.e., Q[0] gives the weight of 'a', Q[1] gives the weight of 'b', etc.).
- **Sliding Window Approach:** To efficiently check substrings, a sliding window or similar approach can be used to compute the sum of weights for substrings starting at each index of P and track unique substrings.
- **Boundary Conditions:** The sum of weights of characters in each substring should not exceed the given value K.
- **Uniqueness:** We need to ensure that each valid substring is counted only once, meaning we need to store substrings and verify that they haven't been counted previously.
- **Algorithm:**
    - Map Characters to Weights: Create a mapping of each character in the alphabet ('a' to 'z') to its corresponding weight from Q.
    - Initialize a Set: Use a set to store unique substrings that satisfy the weight condition.
    - Traverse through Substrings:
        - For each starting position i in the string P, iterate over the possible end positions j to form substrings.
        - Calculate the sum of the weights for each substring.
        - If the sum is less than or equal to K, add the substring to the set.

    - Return the Result: After all substrings have been considered, return the size of the set, which represents the total number of unique substrings with the sum of weights ≤ K.
- **Example Walkthrough:**
    - P = “acbacbacaa”
    - Q = “12300045600078900012345000”
    - K = 2
    - The valid substrings with sum of weights ≤ 2 are:
        - "a" (1)
        - "b" (2)
        - "aa" (1+1 = 2)
    - Output: 3 unique substrings.
- **Applications:**
    - Text Processing: Useful in problems where constraints are applied to substrings based on weights or scores.
    - Pattern Matching: Can be applied in searching for patterns in a string that satisfy certain conditions based on external mappings.
    - Game Design: Similar logic could be used to identify valid word patterns with restricted scores in games or challenges.

Code:

```
import java.util.*;

class Main {
   static int distinctSubString(String P, String Q, int K, int N) {
      HashSet < String > S = new HashSet < String > ();
      for (int i = 0; i < N; ++i) {
         int sum = 0;
         String s = "";
         for (int j = i; j < N; ++j) {
            int pos = P.charAt(j) - 'a';
            sum += Q.charAt(pos) - '0';
            s += P.charAt(j);
            if (sum <= K) {
               S.add(s);
```

```
            }
            else {
               break;
            }
         }
      }
      return S.size();
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String P = sc.nextLine();
      String Q = sc.nextLine();
      int K = sc.nextInt();
      int N = P.length();
      System.out.print(distinctSubString(P, Q, K, N));
   }
}
```

Maneuvering

Intuition:-

The problem requires finding the number of unique paths from the top-left corner to the bottom-right corner of a grid. Each movement is restricted to certain directions (usually right or down), and we need to calculate how many different ways the destination can be reached. The goal is to iterate over all possible paths, applying movement rules, and count each unique valid path.

Key Points:

- **Movement Directions:** At each point in the grid, movement is typically allowed either to the right or down. This creates a limited set of options to explore from each grid cell.
- **Dynamic Programming (DP) Approach:** By using a DP table, we can store the number of ways to reach each cell. Instead of recalculating paths for each cell repeatedly, we compute it once and reuse the result.
- **Boundary Conditions:** Movements are restricted by the grid's size. If an obstacle exists, paths must avoid those grid cells. Additionally, if the grid is empty or has no valid movements, the number of paths is 0.
- **Optimal Substructure:** The number of ways to reach any cell (i, j) is the sum of the number of ways to reach the cell above it and the number of ways to reach the cell to the left of it.
- **Algorithm**:
    - Initialize DP Table: Create a 2D DP array where each cell dp[i][j] represents the number of unique paths to cell (i, j).
    - Base Case: Set dp[0][0] to 1, as there's exactly one way to be in the starting position.
    - Iterate through the Grid: For each cell (i, j), calculate the number of paths to it using the formula:
        - dp[i][j] = dp[i-1][j] + dp[i][j-1], where dp[i-1][j] is the number of paths from the top, and dp[i][j-1] is the number of paths from the left.

    - Handle Boundary Cases: If a cell is on the first row or first column, it can only be reached from one direction (either from the left or the top).
    - Return the Result: After populating the DP table, return dp[m-1][n-1] as the number of unique paths to the bottom-right corner of an m x n grid.
- **Example Walkthrough:**
    - Grid Size: 3x3
    - Possible movements: right or down.
    - DP Table (step-by-step construction):
    - Initialize the table with base case (1 way to be at the start).

      1 0 0
      0 0 0
      0 0 0
    - Fill first row and column:

      1 1 1
      1 0 0
      1 0 0
    - Fill the rest of the table:

      1 1 1
      1 2 3
      1 3 6
    - The number of unique paths to reach the bottom-right corner is 6.
    - Output: 6 unique paths.
- **Applications**:
    - Robot Movement: In robotics, calculating the number of ways a robot can navigate a grid, avoiding obstacles or minimizing movement costs, is a common problem.
    - Maze Solving: The number of paths in a maze with specific movement restrictions (right, down) can be computed similarly.
    - Network Routing: Finding all possible paths in a network with certain constraints, like bandwidth limits or movement rules, can be modeled similarly.

Recursion Code:

```
import java.util.*;

public 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);

   }
   public static void main(String args[])
   {
      Scanner sc = new Scanner(System.in);
      int m=sc.nextInt();
      int n=sc.nextInt();
      System.out.println(numberOfPaths(m, n));
   }
}
```

DP Code:

```
import java.util.*;

public class Main
{
   static int numberOfPaths(int m, int n)
   {
      int count[][] = new int[m][n];
      for (int i = 0; i < m; i++)
         count[i][0] = 1;
      for (int j = 0; j < n; j++)
         count[0][j] = 1;
      for (int i = 1; i < m; i++)
      {
         for (int j = 1; j < n; j++)
```

```
            count[i][j] = count[i - 1][j] + count[i][j - 1];
        }
        return count[m - 1][n - 1];
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        int n = sc.nextInt();
        System.out.println(numberOfPaths(m, n));
    }
}
```

DP Space optimized Code:

```
import java.util.*;

public class Main
{
    static int numberOfPaths(int m, int n)
    {
        int[] dp = new int[n];
        dp[0] = 1;

        for (int i = 0; i < m; i++)
        {
            for (int j = 1; j < n; j++)
                dp[j] += dp[j - 1];
        }
        return dp[n - 1];
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        int n = sc.nextInt();
        System.out.println(numberOfPaths(m, n));
    }
}
```

Sorted Unique Permutation

Intuition:-

The problem requires generating all unique permutations of a given string and printing them in lexicographically sorted order. Since there may be duplicate characters in the input string, we need to ensure that each unique permutation is counted once and that the final output is sorted. The key idea is to recursively generate permutations of the string while storing them in a data structure that guarantees both uniqueness and sorted order.

Key Points:

- **TreeSet for Sorting and Uniqueness:** A TreeSet<String> is used to store the permutations. This ensures that:
    - Sorted Order: The permutations are automatically stored in lexicographical order.
    - Uniqueness: The set does not allow duplicates, so only unique permutations are stored.
- **Backtracking Approach:** The permutations are generated using backtracking. At each step, characters are swapped to explore new permutations.
- **Swap Function:** The characters of the string are swapped to rearrange the elements and form different permutations.
- **Boundary Condition:** Once we reach the end of the string (start == end), the current arrangement of the characters is considered a valid permutation and added to the set.
- **Algorithm**:
    - TreeSet Initialization: Initialize a TreeSet<String> to store the unique permutations in sorted order.
    - Recursive Permutation Generation:
        - The generatePermutation function generates all possible permutations by recursively swapping characters.
        - When the recursion reaches the base case (start == end), the current permutation is converted into a string and added to the TreeSet.
    - Swap Characters: Use the swap function to swap characters at different positions to explore different permutations.

    - Return the Result: After generating all permutations, print each unique permutation in lexicographical order (since the TreeSet is inherently sorted).
- **Example Walkthrough:**
    - Input: "aabc"
    - Step-by-step generation of sorted unique permutations:
        - First permutation: "aabc".
        - Second permutation: "aacb".
        - Continue recursively generating permutations, skipping duplicates.
    - Result:
        - aabc aacb abac abca acab acba baac baca bcaa caab caba cbaa
    - Output: 12 unique sorted permutations.
- **Applications:**
    - Password Generation: Useful for generating unique, sorted passwords from a set of characters.
    - Anagram Finder: Finding all unique anagrams of a word in sorted order.
    - Combinatorial Search Problems: Useful in exploring unique configurations in optimization and search problems where order matters.

Code:

```
import java.util.*;
class Main
{
   static TreeSet<String> ts = new TreeSet<String>();
   static void generatePermutation(char[] arr, int start, int end)
   {
      if(start == end)
      {
         String out = "";

         for(int i=0;i<arr.length;i++)
         {
            out = out + arr[i];
```

```
            }

            ts.add(out);
        }
        else
        {
            for(int i=start;i<=end;i++)
            {
                swap(arr,start,i);
                generatePermutation(arr,start+1,end);
                swap(arr,start,i);
            }
        }
    }
    static void swap(char[] arr, int i,int j)
    {
        char temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main (String[] args)
    {
        Scanner sc=new Scanner(System.in);
        String s =sc.next();
        generatePermutation(s.toCharArray(),0,s.length()-1);
        for(String ele: ts)
        {
            System.out.print(ele+ " ");
        }
    }
}
```

Josephus Problem

Intuition:-

The Josephus Problem (or Josephus Trap) is a theoretical problem related to a group of people standing in a circle and eliminating every k-th person until only one remains. The problem asks for the position of the last person standing. The key challenge is to find a way to efficiently determine the position of the survivor rather than simulating the entire elimination process.

Key Points:

- **Recursive Structure:** The Josephus problem has a recursive nature. After eliminating one person, the problem reduces in size, with the same elimination step applied to the remaining people.
- **Base Case:** When there is only one person left, the position of that person is trivially 0 (or 1, depending on how we index the positions).
- **Adjustment of Positions:** As people are eliminated, the positions of the remaining individuals are adjusted. The recursion accounts for this shift in positions as the group shrinks.
- **Algorithm:**
    - Recursive Definition:
        - If there is only one person, the position of the survivor is 0 (if 0-based indexing) or 1 (if 1-based indexing).
        - For n > 1 people, the position of the survivor can be found by recursively solving the Josephus problem for n-1 people and then adjusting the result to account for the elimination of one person in every k-th step.
        - Formula (0-based): J(n, k) = (J(n - 1, k) + k) % n
        - Formula (1-based): J(n, k) = (J(n - 1, k) + k - 1) % n + 1
- **Example Walkthrough:**
    - Josephus Problem with 7 people, eliminating every 3rd person:
        - Initial circle: 1 2 3 4 5 6 7
        - First, the 3rd person is eliminated (position 3).
        - The circle becomes: 1 2 4 5 6 7
        - Next, the 6th person is eliminated (position 6).
        - The circle becomes: 1 2 4 5 7
        - This process continues until only one person remains.
    - Recursive Formula:

        - For n = 7, k = 3: Use J(7, 3) = (J(6, 3) + 3) % 7.
        - This continues until J(1, 3) is reached, which is the base case.
- **Applications:**
    - Game Elimination Strategies: The Josephus problem can model scenarios in games or puzzles where participants are eliminated in a structured sequence.
    - Resource Allocation in Distributed Systems: It can be used in load balancing or scheduling tasks where resources are distributed cyclically.
    - Mathematical Puzzles and Algorithm Challenges: The Josephus problem is commonly found in algorithmic challenges, involving recursion and modular arithmetic.

Code:

```
import java.util.*;

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)
        {
           Scanner sc = new Scanner(System.in);
           int n = sc.nextInt();
           int k = sc.nextInt();
           System.out.println(josephus(n, k));
        }
}
```

Combination

Intuition:-

The code is designed to generate all combinations of a given size r from an array of integers. A combination is a subset of items selected from a larger set, where the order of selection does not matter. The algorithm leverages recursion to explore all possible subsets of the array that have exactly r elements.

Key Points:

- **Combinations vs. Permutations:** The difference between combinations and permutations is that in combinations, the order of elements doesn't matter. For example, [1, 2] is considered the same as [2, 1].
- **Recursive Approach:** The function combinationUtil recursively builds combinations by selecting or skipping each element in the array. If an element is included in the combination, the function proceeds with the next index.
- **Backtracking:** The algorithm builds combinations incrementally and backtracks when necessary to explore all valid combinations.
- **Stopping Conditions:** The recursion stops either when the combination has reached the desired size r or when there are no more elements to consider in the array.
- **Algorithm**:

  1. Recursive Function combinationUtil:

     - The recursive function explores two possibilities for each element: either include it in the combination or skip it.
     - Base Case: If index == r, a valid combination of size r has been formed and is printed. The function then returns.
     - Recursive Case: At each recursive call, the function makes two recursive calls:
       - One to include the current element in the combination (index + 1).
       - One to exclude the current element and move to the next one (i + 1).

2. Helper Function printCombination:

   - This function initializes a temporary array data[] to store the current combination being formed.
   - It then calls combinationUtil to recursively generate and print all valid combinations.
   - The printCombination function also checks if there are any duplicate consecutive elements in the input array and skips them to ensure combinations remain unique.

3. Main Function:

   - Reads the input array arr[] and the size r of combinations.
   - Calls printCombination to generate all possible combinations of size r.

- **Example Walkthrough:**
  - Let's walk through the example with arr = [1, 2, 3] and r = 2:
  - Initially, combinationUtil is called with index = 0 and i = 0. It considers two cases for each element:
    - Include element arr[0] = 1 in the combination.
    - Exclude element arr[0] and move to the next one.
  - The function continues recursively, including and excluding elements until it forms all valid combinations of size 2.
  - The valid combinations of size 2 from [1, 2, 3] are: [1, 2], [1, 3], and [2, 3].
  - Example Input and Output:
    - Input:

      ```
      3
      1 2 3
      2
      ```

    - Output:

      ```
      1 2
      1 3
      2 3
      ```

- **Applications:**
  - Subset Selection: Useful in problems where a subset of a specific size needs to be selected from a larger set.

- Lottery and Raffle Simulations: In scenarios where combinations of items or numbers need to be drawn without regard to the order.
- Optimization Problems: Combinatorial algorithms are frequently used in optimization problems, where exploring all possible combinations helps find the best solution.

Code:

```
import java.util.*;

public class Main {
   static void combinationUtil(int arr[], int n, int r, int index, int data[], int i) {
      if (index == r) {
         for (int j = 0; j < r; j++) {
            System.out.print(data[j] + " ");
         }
         System.out.println("");
         return;
      }

      if (i >= n) {
         return;
      }

      data[index] = arr[i];
      combinationUtil(arr, n, r, index + 1, data, i + 1);

      while (i + 1 < n && arr[i] == arr[i + 1]) {
         i++;
      }

      combinationUtil(arr, n, r, index, data, i + 1);
   }

   static void printCombination(int arr[], int n, int r) {
      Arrays.sort(arr);

      int data[] = new int[r];
```

```
        combinationUtil(arr, n, r, 0, data, 0);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int arr[] = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        int r = sc.nextInt();
        printCombination(arr, n, r);
    }
}
```

Manacher’s Algorithm

Intuition:-

The problem focuses on finding the longest palindromic substring in a given string S. Manacher's algorithm is efficient for this task, providing a solution in linear time, O(n). Unlike a brute-force method that would require checking every possible substring, Manacher's algorithm cleverly transforms the string to find palindromes around each center by leveraging symmetry properties.

Key Points:

- **String Transformation:** To handle both odd- and even-length palindromes uniformly, the algorithm inserts special characters (e.g., #) between each character of the string and at its ends. This ensures that all palindromic substrings become odd-length.
- **Array for Palindromic Lengths:** An array P is used to store the length of the palindrome radius (half-length) around each character in the transformed string.
- **Symmetry and Mirror Property:** For a palindrome centered at i, if we know the radius of a palindrome around its mirror position j (relative to the current palindrome’s center), we can often directly infer the radius of the palindrome at i, reducing redundant calculations.
- **Center and Right Boundary Update:** The variables center and right help track the farthest palindrome detected so far, speeding up calculations by leveraging existing palindromic information without rescanning.
- **Result Extraction:** After processing the transformed string, the longest palindrome length and its position are used to retrieve the original substring from S.
- **Algorithm:**

  1. Transform the String:

     - Add special characters to S to handle palindromes uniformly.
     - For example, abba becomes #a#b#b#a#.

  2. Initialize Variables:

- Array P to store palindrome radii, initially all zeros.
- Variables center and right to track the center and boundary of the farthest-reaching palindrome.

3. Traverse the Transformed String:

- For each character i in the transformed string:
  - Calculate the mirror position j of i.
  - If i lies within the current palindrome boundary (i < right), initialize P[i] to the minimum of P[j] and (right - i).
  - Expand around i while the characters on both sides match, increasing P[i].
  - If the palindrome around i extends beyond right, update center and right.

4. Extract Result:

- Find the maximum value in P, which gives the radius of the longest palindrome.
- Convert this index and length back to the original string format to obtain the longest palindromic substring.

- **Example Walkthrough:**
  - Input: S = "babad"
  - Transformed string: #b#a#b#a#d#
  - After running Manacher's algorithm, we get P = [0, 1, 0, 3, 0, 1, 0, 3, 0, 1, 0].
  - The maximum value in P is 3, corresponding to the palindrome "aba" or "bab".
  - Output: "aba" (or "bab" as they are both longest palindromic substrings).
- **Applications:**
  - Text Analysis: Useful in identifying palindromic patterns in DNA sequences or language processing.
  - Pattern Recognition: Efficient in problems requiring detection of symmetrical or palindromic structures.
  - Optimization in Search Problems: The linear time complexity makes it practical for large-scale applications, like searching for palindromic substrings in massive datasets.

Code:

```
import java.util.*;

public class Main
{
   public static String longestPalindromicSubstring(String s) {
      StringBuilder t = new StringBuilder();
      t.append("#");
      for (int i = 0; i < s.length(); i++)
      {
         t.append(s.charAt(i)).append("#");
      }

      char[] chars = t.toString().toCharArray();
      int n = chars.length;
      int[] p = new int[n];
      int center = 0, right = 0;

      int maxLen = 0;
      int centerIndex = 0;

      for (int i = 0; i < n; i++)
      {
         int mirror = 2 * center - i;

         if (i < right)
         {
            p[i] = Math.min(right - i, p[mirror]);
         }

         while (i + p[i] + 1 < n && i - p[i] - 1 >= 0 && chars[i + p[i] + 1] == chars[i - p[i] - 1])
         {
            p[i]++;
         }

         if (i + p[i] > right)
         {
            center = i;
```

```
                right = i + p[i];
            }

            if (p[i] > maxLen)
            {
                maxLen = p[i];
                centerIndex = i;
            }
        }

        int start = (centerIndex - maxLen) / 2;
        return s.substring(start, start + maxLen);
    }

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

N Queens

Intuition:-

The N-Queens problem is a classic problem where the task is to place N queens on an N x N chessboard such that no two queens threaten each other. A queen can attack another queen if they share the same row, column, or diagonal. The goal is to find all valid configurations of the N queens on the board.

The problem can be efficiently solved using backtracking, which explores potential solutions by placing queens row by row while pruning invalid placements early. The algorithm recursively places queens on the board, checks for validity, and backtracks when necessary.

Key Points:

- **Backtracking Approach:**Place queens one row at a time, trying all columns in each row.For each row and column combination, validate that placing a queen doesn't result in a conflict with any previously placed queens.If no valid positions are found for a row, backtrack to the previous row and try the next possible position.
- **Validation:**The validate() method ensures that a queen can be placed at a given position without conflicting with queens already placed on the board.
- **It checks:**The column for conflicts.The two diagonals for conflicts (both descending and ascending).
- **State Representation:**The board is represented as a 2D character array, where each cell is initially '.' (empty) and is changed to 'Q' when a queen is placed there.
- **Recursive Search:**The function dfs() recursively tries to place queens in each column of each row. When all queens are placed successfully, the board configuration is stored in the result list.
- **Solution Construction:**After a valid configuration is found, the board is converted from a 2D array to a list of strings where each string represents a row of the board.
- **Algorithm:**

  1. Initialize the Board:

    - Create an n x n board, initialized to '.'.

2. Backtracking with DFS:

    - Begin placing queens starting from the first column.
    - For each column, attempt placing a queen in each row. If the placement is valid, recursively place queens in the next column.
    - If placing queens in all columns is successful, store the configuration.

3. Validation:

    - For each possible queen placement, check whether the placement conflicts with any previously placed queens by validating the column and both diagonals.

4. Result Extraction:

    - Once a valid arrangement is found, convert the board configuration into a list of strings and add it to the result.

- **Example Walkthrough:**
    - Input:
        - N = 4
    - Steps:
        - Initialize the Board:
        - The board is initialized as [['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']].
    - Recursive Backtracking:
        - Place queens row by row. In the first step, try placing a queen in each column of the first row, and for each placement, check for conflicts.
        - When all queens are placed, store the configuration.
    - Validation:
        - Each time a queen is placed, the validate() function ensures there are no conflicts in the current column and diagonals.
    - Backtracking:

        - If a conflict is detected at any point, the algorithm backtracks by removing the queen and trying the next possible position.
    - Construct Solution:
        - When a valid configuration is found, the 2D array board is converted into a list of strings.
    - Output:
        - For N = 4, the two possible valid solutions are:

          Arrangement 1:
          . Q . .
          . . . Q
          Q . . .
          . . Q .

          Arrangement 2:
          . . Q .
          Q . . .
          . . . Q
          . Q . .

- **Applications**:
    - Constraint Satisfaction Problems (CSPs):The N-Queens problem is a classical example of constraint satisfaction, where the task is to place queens subject to certain constraints (no two queens in the same row, column, or diagonal).
    - AI and Search Algorithms:Backtracking is an essential technique used in many AI algorithms for solving problems where a solution involves searching through a space of potential configurations.
    - Optimization Problems:The N-Queens problem is a well-known optimization problem, though the task here is to find all valid configurations, not to optimize for a particular objective.
    - Parallel Computing:Solutions to the N-Queens problem are independent of each other, making it a good candidate for parallel computation, where multiple configurations can be explored simultaneously.

Code 1:

```
import java.util.*;
class Main
{
   public static List < List < String >> solveNQueens(int n)
   {
      char[][] board = new char[n][n];
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            board[i][j] = '.';
      List < List < String >> res = new ArrayList < List < String >> ();
      dfs(0, board, res);
      return res;
   }

   static boolean validate(char[][] board, int row, int col)
   {
      int duprow = row;
      int dupcol = col;
      while (row >= 0 && col >= 0)
      {
         if (board[row][col] == 'Q') return false;
         row--;
         col--;
      }

      row = duprow;
      col = dupcol;
      while (col >= 0)
      {
         if (board[row][col] == 'Q') return false;
         col--;
      }

      row = duprow;
      col = dupcol;
      while (col >= 0 && row < board.length)
      {
```

```
        if (board[row][col] == 'Q') return false;
        col--;
        row++;
    }
    return true;
}

static void dfs(int col, char[][] board, List < List < String >> res)
{
    if (col == board.length)
    {
        res.add(construct(board));
        return;
    }

    for (int row = 0; row < board.length; row++)
    {
        if (validate(board, row, col))
        {
            board[row][col] = 'Q';
            dfs(col + 1, board, res);
            board[row][col] = '.';
        }
    }
}



static List < String > construct(char[][] board)
{
    List < String > res = new LinkedList < String > ();
    for (int i = 0; i < board.length; i++)
    {
        String s = new String(board[i]);
        res.add(s);
    }
    return res;
}
public static void main(String args[])
```

```
    {
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        List < List < String >> queen = solveNQueens(N);
        int i = 1;
        for (List < String > it: queen)
        {
            System.out.println("Arrangement " + i);
            for (String s: it)
            {
                System.out.println(s);
            }
            System.out.println();
            i += 1;
        }

    }
}
```

Code 2:

```
import java.util.*;
class Main
{
    public static List < List < String >> solveNQueens(int n)
    {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                board[i][j] = '.';
        List < List < String >> res = new ArrayList < List < String >> ();
        int leftRow[] = new int[n];
        int upperDiagonal[] = new int[2 * n - 1];
        int lowerDiagonal[] = new int[2 * n - 1];
        solve(0, board, res, leftRow, lowerDiagonal, upperDiagonal);
        return res;
    }
```

```
   static void solve(int col, char[][] board, List < List < String >> res, int leftRow[], int
lowerDiagonal[], int upperDiagonal[])
   {
      if (col == board.length)
      {
         res.add(construct(board));
         return;
      }

      for (int row = 0; row < board.length; row++)
      {
         if (leftRow[row] == 0 && lowerDiagonal[row + col] == 0 &&
upperDiagonal[board.length - 1 + col - row] == 0)
         {
            board[row][col] = 'Q';
            leftRow[row] = 1;
            lowerDiagonal[row + col] = 1;
            upperDiagonal[board.length - 1 + col - row] = 1;
            solve(col + 1, board, res, leftRow, lowerDiagonal, upperDiagonal);
            board[row][col] = '.';
            leftRow[row] = 0;
            lowerDiagonal[row + col] = 0;
            upperDiagonal[board.length - 1 + col - row] = 0;
         }
      }
   }


   static List < String > construct(char[][] board)
   {
      List < String > res = new LinkedList < String > ();
      for (int i = 0; i < board.length; i++)
      {
         String s = new String(board[i]);
         res.add(s);
      }
```

```
        return res;
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        List < List < String >> queen = solveNQueens(N);
        int i = 1;
        for (List < String > it: queen) {
            System.out.println("Arrangement " + i);
            for (String s: it) {
                System.out.println(s);
            }
            System.out.println();
            i += 1;
        }

    }
}
```

Maze Solving

Intuition:-

The problem is about finding all possible paths from the top-left corner to the bottom-right corner of a maze, where you can only move through cells that are open (represented by 1). The moves can be made in any of the four directions: Down (D), Left (L), Right (R), and Up (U). The goal is to return all possible paths in the form of strings, where each string represents a sequence of moves taken from start to end.

The problem is solved using a Backtracking approach, where we recursively explore all possible paths, marking cells as visited and backtracking when no valid move is possible.

Key Points:

- **Maze Representation:**
  - The maze is represented as a 2D matrix where:
  - 1 represents an open path (can move through).
  - 0 represents a blocked path (cannot move through).
  - We are given the starting point (0, 0) and the destination point (n-1, n-1).
- **Backtracking:**We start at the top-left corner (i.e., (0, 0)) and recursively explore all possible directions (Down, Right).If we reach the bottom-right corner (n-1, n-1), the current path is added to the result list.We maintain a visited matrix to avoid revisiting cells and to prevent infinite loops.After exploring a move, we backtrack, unmarking the current cell as visited, and continue exploring other directions.
- **Edge Cases:**
  - If the starting point (0, 0) is blocked (0), no valid path exists.
  - If there is no valid path from start to end, return -1.
- **Algorithm:**

  1. Initialize Visited Matrix:

     - A 2D matrix vis is used to track visited cells. Initially, all cells are unvisited (set to 0).

2. Recursive Search:Start from the top-left corner (0, 0) and try all 4 possible directions:

    - Down (D)
    - Right (R)
    - For each valid move (i.e., the move is within bounds, the cell is open, and not visited), mark the cell as visited and recursively explore from the new position.
    - If the destination (n-1, n-1) is reached, store the current path.

3. Backtrack:

    - Once all possible moves from a cell are explored, backtrack by marking the cell as unvisited and return to the previous state.

4. Path Reconstruction:

    - If the end is reached, reconstruct the path by appending the corresponding move (D, L, R, U) to the string.

5. Result:

    - Return the list of all paths found. If no path is found, return -1.

- **Example Walkthrough:**
    - Input:
        - 5 5

          1 0 0 0 0
          1 1 0 1 1
          0 1 0 0 1
          0 1 1 1 0
          0 0 0 1 1
    - The maze has 1 representing an open path and 0 representing a blocked cell.
    - Start: (0, 0)
    - End: (4, 4)
    - The possible moves are:
        - From (0, 0) → (1, 0) → (1, 1) → (2, 1) → (3, 1) → (4, 1) → (4, 2) → (4, 3) → (4, 4).
    - Output:

        - DDRRRRR
        - This is the shortest path from the start to the end, represented as a sequence of directions: Down (D), Right (R).
- **Applications:**
    - Robotics Pathfinding:This algorithm can be used in robotic navigation to find paths through a grid-based environment while avoiding obstacles.
    - Game Development:Many games that involve maze-solving, pathfinding, or exploration use this kind of backtracking approach to navigate the game environment.
    - Network Routing:Similar pathfinding problems are used in network routing algorithms to find paths between two points in a network while avoiding blocked nodes (congested nodes).
    - AI-based Pathfinding:AI-controlled characters in video games, like in puzzle-solving or maze-solving tasks, often utilize pathfinding algorithms like this one to determine possible routes.

Code:

```
import java.util.*;


class Solution
{
   static void solve(int i, int j, int a[][], int n, ArrayList < String > ans, String move,
   int vis[][])
   {
      if (i == n - 1 && j == n - 1)
      {
         ans.add(move);
         return;
      }

      if (i + 1 < n && vis[i + 1][j] == 0 && a[i + 1][j] == 1)
      {
         vis[i][j] = 1;
         solve(i + 1, j, a, n, ans, move + 'D', vis);
```

```
        vis[i][j] = 0;
    }


    if (j - 1 >= 0 && vis[i][j - 1] == 0 && a[i][j - 1] == 1)
    {
        vis[i][j] = 1;
        solve(i, j - 1, a, n, ans, move + 'L', vis);
        vis[i][j] = 0;
    }


    if (j + 1 < n && vis[i][j + 1] == 0 && a[i][j + 1] == 1)
    {
        vis[i][j] = 1;
        solve(i, j + 1, a, n, ans, move + 'R', vis);
        vis[i][j] = 0;
    }


    if (i - 1 >= 0 && vis[i - 1][j] == 0 && a[i - 1][j] == 1)
    {
        vis[i][j] = 1;
        solve(i - 1, j, a, n, ans, move + 'U', vis);
        vis[i][j] = 0;
    }
}
public static ArrayList < String > findPath(int[][] m, int n)
{
    int vis[][] = new int[n][n];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            vis[i][j] = 0;
        }
    }

    ArrayList < String > ans = new ArrayList < > ();
```

```
        if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis);
        return ans;
    }
}
class Main
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int m = sc.nextInt();
        int[][] a = new int[m][m];

        for(int i=0;i<m;i++)
        {
            for(int j=0;j<m;j++)
            {
                a[i][j]=sc.nextInt();
            }
        }

        Solution obj = new Solution();
        ArrayList < String > res = obj.findPath(a, m);
        if (res.size() > 0)
        {
            for (int i = 0; i < res.size(); i++)
                System.out.print(res.get(i) + " ");
        System.out.println();
        }
        else
        {
        System.out.println(-1);
        }
    }
}
```

Activity Selection Problem

Intuition:-

The problem requires selecting the maximum number of non-overlapping activities from a given list, where each activity has a defined start and end time. The key is to prioritize activities that finish earlier, as they leave the maximum time for subsequent activities.

Key Points:

- **Sorting by Finish Time:** To maximize the count of activities, we sort them by their finish times in ascending order. This allows us to always choose the next activity that finishes the earliest, making room for more activities afterward.
- **Greedy Approach:** Starting with the activity that has the earliest finish time, we iteratively add activities that start after or at the finish of the last chosen activity.
- **Boundary Conditions:** If there is only one activity or all activities overlap, the solution will either be the single activity or no valid selections, respectively.
- **Algorithm**:

  1. Sort Activities: Arrange all activities in ascending order by their finish times.

  2. Select Activities:

     - Start with the first activity (after sorting).
     - For each subsequent activity, if its start time is greater than or equal to the finish time of the last selected activity, select it and update the last selected finish time.

  3. Count and Return: The total number of selected activities gives the maximum number of non-overlapping activities that can be performed.

- **Example Walkthrough:**
  - Given a list of activities with start and finish times:

        - Activities: A1, A2, A3, A4, A5
        - Start Times: 1, 3, 0, 5, 8
        - Finish Times: 2, 4, 6, 7, 9
    - Steps:
    - Sort activities by finish times: [(A1: 1-2), (A2: 3-4), (A4: 5-7), (A3: 0-6), (A5: 8-9)]
    - Select activities:
        - Start with A1 (1-2).
        - A2 (3-4) starts after A1 finishes, so select A2.
        - A4 (5-7) starts after A2 finishes, so select A4.
        - A5 (8-9) starts after A4 finishes, so select A5.
    - Output: 4 activities can be selected (A1, A2, A4, A5).
- **Applications**:
    - Scheduling: Ideal for scheduling non-overlapping meetings, tasks, or events in limited time slots.
    - Resource Allocation: Useful in resource allocation problems where resources are limited to non-overlapping usage intervals.
    - Event Planning: In event management, optimizing time slots to fit in maximum events without conflict.

Code:

```
import java.util.*;

class Activity {
   int start;
   int finish;

   public Activity(int start, int finish) {
      this.start = start;
      this.finish = finish;
   }

   @Override
   public String toString() {
      return "(" + start + ", " + finish + ")";
   }
```

```
}

public class Main {

    public static ArrayList<Activity> selectActivities(ArrayList<Activity> activities) {
        Collections.sort(activities, Comparator.comparingInt(a -> a.finish));

        ArrayList<Activity> selectedActivities = new ArrayList<>();
        int lastFinishTime = -1;

        for (Activity activity : activities) {
            if (activity.start >= lastFinishTime) {
                selectedActivities.add(activity);
                lastFinishTime = activity.finish;
            }
        }
        return selectedActivities;
    }

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

        int n = scanner.nextInt();

        ArrayList<Activity> activities = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            int start = scanner.nextInt();
            int finish = scanner.nextInt();
            activities.add(new Activity(start, finish));
        }

        ArrayList<Activity> selectedActivities = selectActivities(activities);

        System.out.println(selectedActivities);
        scanner.close();
    }
}
```