**Batch- T5**

**Practical No. 8**

**Name: Siya Pondkule**

**PRN: 23520008**

Assignment No.8

Dynamic Programming Algorithms

1. You are given an array containing n integers. Your task is to determine the longest increasing subsequence in the array, i.e., the longest subsequence where every element is larger than the previous one.

A subsequence is a sequence that can be derived from the array by deleting some elements without changing the order of the remaining elements.

# Input:

The first line contains an integer n: the size of the array.

After this there are n integers x1, x2,….., xn: the contents of the array.

# Output:

Print the length of the longest increasing subsequence.

# Constraints:

1 ≤ n ≤ 2 \* 105

1 ≤ xi ≤109 Example Input:

8

7 3 5 3 6 2 9 8

Output:

4

### Step-by-Step Algorithm

1. Create an empty list lis to hold the elements of the subsequence.
2. Loop through each element xi in the array.
3. Use binary search to find the appropriate position in lis where xi can be placed.
4. If xi is larger than all elements in lis, append it to the list.
5. If there’s an element in lis larger than or equal to xi, replace that element with xi (this maintains the smallest possible elements for each subsequence length).
6. At the end, the size of the lis list will give the length of the longest increasing subsequence.

import java.util.*\**;

public class LongestIncreasingSubsequence {

    public static int findLIS(int[] arr) {

        ArrayList<Integer> lis = new ArrayList<>();

        for (int num : arr) {

            int pos = Collections.binarySearch(lis, num);

            if (pos < 0) {

                pos = -(pos + 1);

            }

            if (pos < lis.size()) {

                lis.set(pos, num);

            } else {

                lis.add(num);

            }

        }

        return lis.size();

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of elements:");

        int n = sc.nextInt();

        System.out.println("Enter elements in array:");

        int[] arr = new int[n];

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

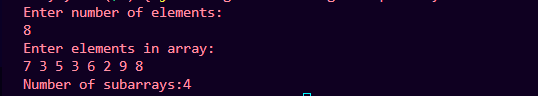
        System.out.println("Number of subarrays:"+findLIS(arr));

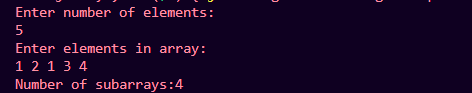
        sc.close();

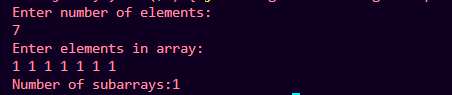
    }

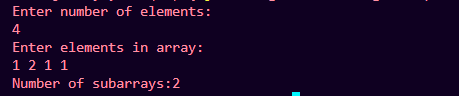
}

Output:









1. There are n people who want to get to the top of a building which has only one elevator. You know the weight of each person and the maximum allowed weight in the elevator. What is the minimum number of elevator rides?

# Input:

The first input line has two integers n and x: the number of people and the maximum allowed weight in the elevator.

The second line has n integers w1,w2…….,wn: the weight of each person.

# Output:

Print one integer: the minimum number of rides.

# Constraints:

1 ≤ n ≤ 20

1 ≤ x ≤ 109 1 ≤ wi ≤ x

Example Input:

4 10

4 8 6 1

Output:

2

### Algorithm Steps:

1. **Initialize DP Array**:
   1. Create a DP array dp where dp[mask] stores (rides, current\_weight) for the given subset of people.
2. **Iterate Over Each Mask**:
   1. For each subset of people (represented by a bitmask), calculate the optimal number of rides required by adding a new person to the current ride if possible, or starting a new ride.
3. **Update the DP Array**:
   1. For each possible person not in the current subset (using bitmasking operations), try adding them to the current ride or starting a new ride.
4. **Result**:
   1. After evaluating all possible subsets, the value dp[(1 << n) - 1] will give the minimum number of rides required to transport all people.

import java.util.*\**;

public class ElevatorRides {

    public static int minElevatorRides(int n, int maxWeight, int[] weights) {

        int INF = n + 1;

        int[][] dp = new int[1 << n][2];

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

            dp[i][0] = INF;

            dp[i][1] = 0;

        }

        dp[0][0] = 1;

        dp[0][1] = 0;

        for (int mask = 0; mask < (1 << n); mask++) {

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

                if ((mask & (1 << i)) == 0) {

                    int newMask = mask | (1 << i);

                    if (dp[mask][1] + weights[i] <= maxWeight) {

                        if (dp[mask][0] < dp[newMask][0] ||

                            (dp[mask][0] == dp[newMask][0] && dp[mask][1] + weights[i] < dp[newMask][1])) {

                            dp[newMask][0] = dp[mask][0];

                            dp[newMask][1] = dp[mask][1] + weights[i];

                        }

                    } else {

                        if (dp[mask][0] + 1 < dp[newMask][0]) {

                            dp[newMask][0] = dp[mask][0] + 1;

                            dp[newMask][1] = weights[i];

                        }

                    }

                }

            }

        }

        return dp[(1 << n) - 1][0];

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of peoples:");

        int n = sc.nextInt();

        System.out.println("Enter max allowed weight in the elevator:");

        int x = sc.nextInt();

        System.out.println("max allowed weight in the elevator");

        int[] weights = new int[n];

        for (int i = 0; i < n; i++) {

            weights[i] = sc.nextInt();

        }

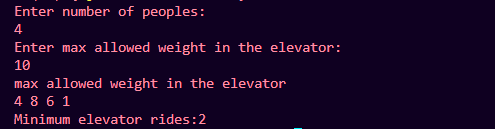
        System.out.println("Minimum elevator rides:"+minElevatorRides(n, x, weights));

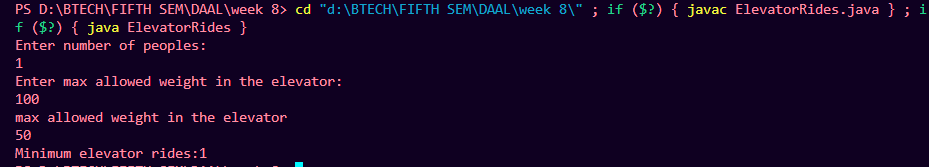
        sc.close();

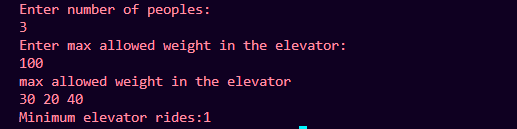
    }

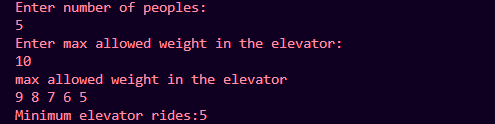
}

Output:





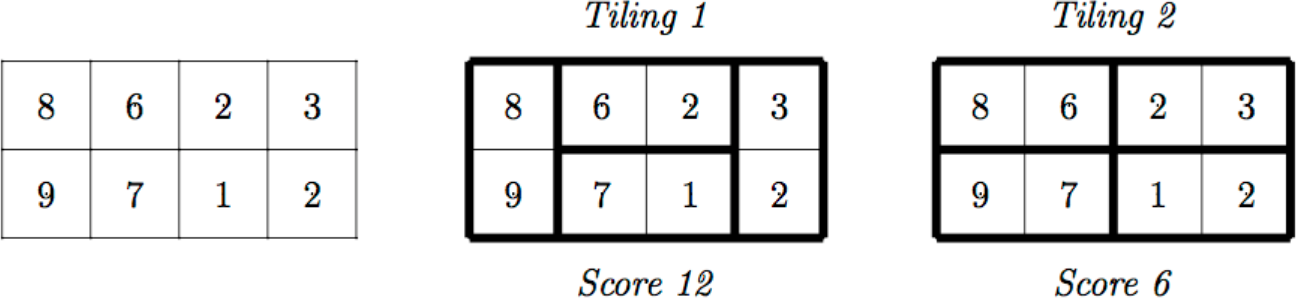




1. In Domino Solitaire, you have a grid with two rows and many columns. Each square in the grid contains an integer. You are given a supply of rectangular 2 × 1 tiles, each of which exactly covers two adjacent squares of the grid. You have to place tiles to cover all the squares in the grid such that each tile covers two squares and no pair of tiles overlap.

The score for a tile is the difference between the bigger and the smaller number that are covered by the tile. The aim of the game is to maximize the sum of the scores of all the tiles.

Here is an example of a grid, along with two different tilings and their scores.



The score for Tiling 1 is 12 = (9−8)+(6−2)+(7−1)+(3−2) while the score for Tiling 2 is 6 = (8−6)+(9−7)+(3−2)+(2−1). There are other tilings possible for this grid, but you can check that Tiling 1 has the maximum score among all tilings.

Your task is to read the grid of numbers and compute the maximum score that can be achieved by any tiling of the grid.

# Solution hint

Recursively find the best tiling, from left to right. You can start the tiling with one vertical tile or two horizontal tiles. Use dynamic programming to evaluate the recursive expression efficiently.

# Input format

The first line contains one integer N, the number of columns in the grid. This is followed by 2 lines describing the grid. Each of these lines consists of N integers, separated by blanks.

# Output format

A single integer indicating the maximum score that can be achieved by any tiling of the given grid.

# Test Data:

For all inputs, 1 ≤ N ≤ 105. Each integer in the grid is in the range {0,1,...,104}. Sample Input:

4

8 6 2 3

9 7 1 2

Sample Output:

12

Sample Test Cases

|  |  |  |
| --- | --- | --- |
|  | Input | Output |
| Test | 4 |  |
| 12 |
| Case | 8 6 2 3 |
| 1 | 9 7 1 2 |
|  |
|  |  |  |
| Test | 10 | 31597 |
| Case | 8789 7959 4809 5257 4592 9455 6462 5855 6399 9569 |
| 2 | 4977 5499 7329 2997 9599 5445 2412 9838 6252 6577 |
|  |

### ****Step-by-Step Algorithm:****

1. **Define the DP State:**
   * Let dp[i] be the maximum score achievable by covering the grid up to the ith column.
   * The base case is dp[0] = 0 since no tiles have been placed at the start.
2. **Transition:**
   * For each column i, you have two options:
     + **Place a vertical tile** at column i, covering both rows. The score will be the difference between the two values in the column.
     + **Place horizontal tiles** starting from column i, covering two consecutive columns (i and i+1).
3. **Recursion:**
   * Recursively calculate the best tiling for the remaining grid.
   * Use memoization to store the results of subproblems to avoid recomputation.
4. **Base Case:**
   * If all columns are processed, return 0 because no further score can be achieved.
5. **Optimization:**
   * Use memoization or a bottom-up DP approach to efficiently compute the solution for large inputs (N can be up to 10^5).

import java.util.*\**;

public class DominoSolitaire {

    static int N;

    static int[][] grid;

    static int[] dp;

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter number of columns:");

        N = scanner.nextInt();

        grid = new int[2][N];

        System.out.println("Input grid values:");

        for (int i = 0; i < 2; i++) {

            for (int j = 0; j < N; j++) {

                grid[i][j] = scanner.nextInt();

            }

        }

        dp = new int[N];

        Arrays.fill(dp, -1);

        int result = findMaxScore(0);

        System.out.println(result);

    }

    static int findMaxScore(int col) {

        if (col >= N) return 0;

        if (dp[col] != -1) return dp[col];

        int option1 = Math.abs(grid[0][col] - grid[1][col]) + findMaxScore(col + 1);

        int option2 = Integer.MIN\_VALUE;

        if (col + 1 < N) {

            option2 = Math.abs(grid[0][col] - grid[0][col + 1])

                    + Math.abs(grid[1][col] - grid[1][col + 1])

                    + findMaxScore(col + 2);

        }

        dp[col] = Math.max(option1, option2);

        return dp[col];

    }

}

Output:

