**Batch : T7**

**Practical No. 8**

**Title of Assignment : Dynamic Programming**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement :**

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 \* 10^5

1 ≤ xi ≤10^9

Example

Input:

8

7 3 5 3 6 2 9 8

Output:

4

1. Algorithm :

**Step 1: Understanding the Problem**

* **Subsequence**: A subsequence is a sequence that can be derived from the array by deleting some elements without changing the order of the remaining elements.
* **Increasing**: Each element in the subsequence must be strictly greater than the previous one.

**Step 2: Approach Overview**

We will use a **dynamic programming with binary search** approach to solve this efficiently. The idea is to maintain a list where we store the smallest possible tail value for subsequences of different lengths, and we will use **binary search** to determine where to place each element.

**Why binary search?**

* For each element, instead of linearly searching for its position in the subsequence, we can use binary search to insert it in the correct position. This improves the time complexity from O(n2)O(n^2)O(n2) to O(nlog⁡n)O(n \log n)O(nlogn).

**Step 3: Algorithm Steps**

1. **Input the array**:
   * First, take the input of the array size nnn and the array itself.
2. **Create an auxiliary list (lis) to store the potential subsequence**:
   * This list will store the smallest possible tail values for all increasing subsequences of different lengths.
   * Initialize an empty list lis.
3. **Iterate through the input array**:
   * For each element in the array:
     + Use **binary search** (using bisect\_left from Python's bisect module) to find the position in the lis list where the current element should go.
     + If the element can extend the largest subsequence, append it to lis.
     + If the element can replace a larger value in the lis (i.e., a value that is not maintaining the smallest subsequence), update that position.
4. **Return the length of the lis list**:
   * The length of the lis list represents the length of the longest increasing subsequence.

Program Code :

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int lengthOfLIS(const vector<int>& nums) {

    if (nums.empty()) return 0;

    vector<int> lis;

    for (int num : nums) {

        auto it = lower\_bound(lis.begin(), lis.end(), num);

        if (it == lis.end()) {

            lis.push\_back(num);

        } else {

            \*it = num;

        }

    }

    return lis.size();

}

int main() {

    int n;

    cout << "Enter the size of the array: ";

    cin >> n;

    vector<int> nums(n);

    cout << "Enter the elements of the array: ";

    for (int i = 0; i < n; i++) {

        cin >> nums[i];

    }

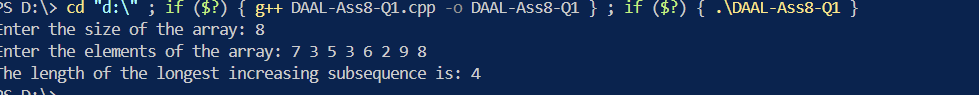
    int result = lengthOfLIS(nums);

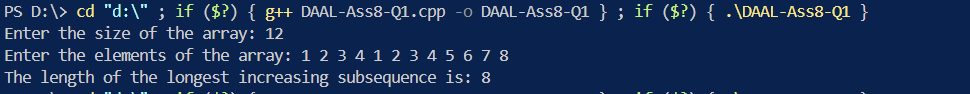
    cout << "The length of the longest increasing subsequence is: " << result << endl;

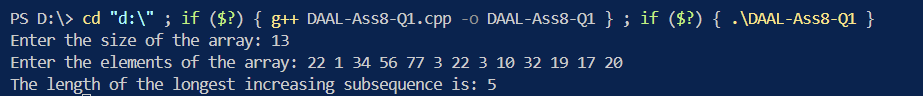
    return 0;

}

Output to verify different test cases :







**Complexity Analysis :**

**Summary**

* **Time Complexity:**
  + Best Case: O(nlogn)
  + Average Case: O(nlogn)
  + Worst Case: O(nlogn)
* **Space Complexity:** O(n)

**Problem Statement :**

2) 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 ≤ 10^9

1 ≤ wi ≤ x

Example

Input:

4 10

4 8 6 1

Output:

2

**Algorithm :**

Step 1: Problem Understanding

* We have n people, each with a specific weight.
* The elevator has a maximum capacity x (total weight it can carry in one ride).
* We need to find the minimum number of rides to take all the people to the top without exceeding the elevator’s capacity.

Step 2: Input

* The first line contains two integers n (number of people) and x (maximum weight the elevator can carry).
* The second line contains n integers w1, w2, ..., wn representing the weight of each person.

Step 3: Define the State

We can represent each possible set of people using a bitmask of size n, where each bit is either 1 (person included in the current subset) or 0 (person not included). We will use Dynamic Programming (DP) to keep track of:

* dp[mask]: The minimum number of rides required to move the people represented by the bitmask mask.
* best\_weight[mask]: The total weight in the last ride for the people represented by the bitmask mask.

Step 4: Transition

For each person, we will check whether adding them to the current ride exceeds the elevator capacity:

* If their weight can be added to the current ride (without exceeding the capacity), we update the state by considering adding them to the current ride.
* If their weight exceeds the current ride’s remaining capacity, start a new ride with this person.

Step 5: Base Case

* Initially, dp[0] = 1 (one ride for the empty set, i.e., zero people).
* The weight of the last ride in the empty set is best\_weight[0] = 0 (no one is in the elevator).

Step 6: Iterate Through All Masks

* Loop through all possible subsets (bitmasks) and update the DP values by considering every person and whether to include them in the current subset.

Step 7: Example Walkthrough

**Program Code :**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

    int n, x;

    cout << "Enter the number of persons: ";

    cin >> n;

    cout << "Enter the maximum allowed weight in the elevator: ";

    cin >> x;

    vector<int> weights(n);

    vector<int> excluded;

    cout << "Enter the weights of each person: ";

    for (int i = 0; i < n; i++) {

        cin >> weights[i];

        if (weights[i] > x) {

            excluded.push\_back(weights[i]);

        }

    }

    weights.erase(remove\_if(weights.begin(), weights.end(), [&](int w) {

        return w > x;

    }), weights.end());

    sort(weights.begin(), weights.end());

    int rides = 0;

    int left = 0;

    int right = weights.size() - 1;

    while (left <= right) {

        if (weights[left] + weights[right] <= x) {

            left++;

        }

        right--;

        rides++;

    }

    if (!excluded.empty()) {

        cout << "Excluded passengers (weight exceeds maximum): ";

        for (int weight : excluded) {

            cout << weight << " ";

        }

        cout << endl;

    } else {

        cout << "No passengers were excluded." << endl;

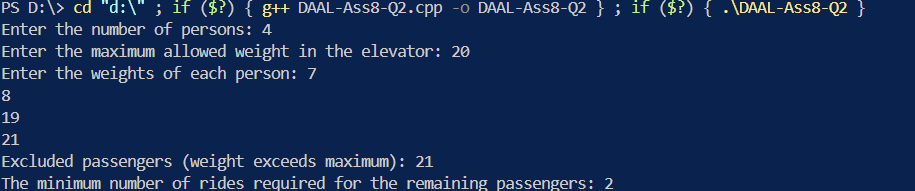
    }

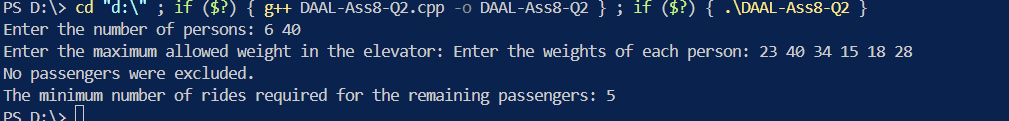
    cout << "The minimum number of rides required for the remaining passengers: " << rides << endl;

    return 0;

}

**Output to verify test cases :**





**Analysis of Complexity**

* **Time Complexity:**
  + Best Case: O(nlogn)
  + Average Case: O(nlogn)
  + Worst Case: O(nlogn)
* **Space Complexity:** O(n)

**Problem Statement :**

3) 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

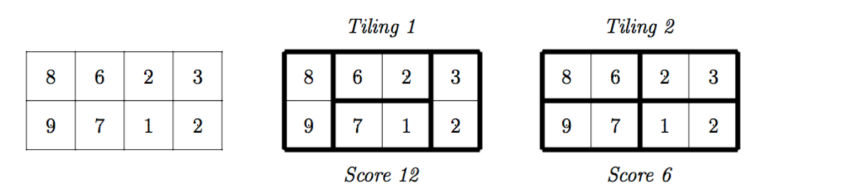
covers two adjacent squares of the grid. You have to place tiles to cover all the squares in the

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

Case

1

4

8 6 2 3

9 7 1 2

12

Test

Case

2

10

8789 7959 4809 5257 4592 9455 6462 5855 6399 9569

4977 5499 7329 2997 9599 5445 2412 9838 6252 6577

31597

Test

Case

3

100

2511 2090 9410 4226 3959 3826 2318 5356 5333 8630 9624 3155 7360 6547 503 4539 8065 6558 8119 8299 792 2046

6803 6519 9765 851 2039 2315 143 1566 141 7040 894 5713 9574 2861 1437 8254 8573 3503 2540 2862 8272 5518

9578 155 8493 9935 1672 5874 5457 3379 3689 6102 9972 4269 3263 274 8535 2766 1393 1859 2864 8412 368 6360

9530 1607 5327 6394 6831 86 7476 1983 1257 9508 5275 8492 8620 4276 800 5409 2229 6220 8377 2016 1569 1255

1554 4253 3592 8325 8073 4123 5605 7625 4737 5013 4173 2287

9668 4457 791 6609 6438 9208 9074 5723 6687 4940 3855 3866 7280 6290 3158 7736 7585 9150 5101 5567 8238 605

3218 3442 6767 7493 2552 6121 7803 9479 1702 7483 7379 9357 1309 4021 6197 2206 402 6193 5867 6284 8661

5558 3199 5171 4723 8388 9933 827 9738 7870 1030 6640 7850 249 2164 4176 4203 4686 2685 5869 9403 698 1360

1954 1818 464 9144 5064 5033 9785 2402 1599 3597 1153 5942 9486 1823 4149 3317 6659 5671 2763 753 518 9301

399 5176 3041 5035 2088 8825 8874 7437 9378 6412 9721 9874 7499

425423

**Algorithm :**

Step 1: Problem Understanding

* Input:
  + We are given a grid with 2 rows and N columns.
  + Each grid cell contains an integer, and we are allowed to tile the grid using 2x1 dominoes either vertically (one domino covering one cell in each row) or horizontally (one domino covering two adjacent cells in the same row).
* Objective:
  + Our goal is to tile the grid such that the sum of the differences between the larger and smaller values covered by each tile is maximized.

Step 2: Approach

To solve this problem efficiently, we will use a dynamic programming approach that computes the maximum score based on the choices made at each step (whether to place the tile vertically or horizontally).

Step 3: Input Format

* The first line contains an integer N (the number of columns in the grid).
* The second and third lines contain the two rows of the grid, each with N integers.

Step 4: Define DP State

* Let dp[i] represent the maximum score we can achieve from column i to column N-1 (right end).

At each column i, we can choose one of the following tiling options:

1. Vertical Tile: A 2x1 tile placed vertically covering both the top and bottom row in column i.
   * The score for this move is the absolute difference between the top and bottom elements at column i (i.e., abs(grid[0][i] - grid[1][i])).
   * Move to the next column i+1 to tile the rest of the grid.
2. Horizontal Tiles: Two horizontal tiles covering adjacent columns i and i+1:
   * One tile covers the top row at column i and column i+1, and another tile covers the bottom row at column i and column i+1.
   * The score for this move is the sum of the differences of the two horizontal tiles:
     + abs(grid[0][i] - grid[0][i+1]) for the top row,
     + abs(grid[1][i] - grid[1][i+1]) for the bottom row.
   * After this, move to column i+2 to tile the rest of the grid.

Step 5: Base Case

* The base case is when we've reached the last column i = N. In this case, no more tiles can be placed, and the score is 0.

Step 6: Recurrence Relation

We define the DP recurrence as follows:

* For a vertical tile at column i:
  + dp[i] = abs(grid[0][i] - grid[1][i]) + dp[i+1]
* For horizontal tiles at columns i and i+1 (only if i+1 is within bounds):
  + dp[i] = abs(grid[0][i] - grid[0][i+1]) + abs(grid[1][i] - grid[1][i+1]) + dp[i+2]

At each column i, we take the maximum score between these two options.

Program Code :

#include <iostream>

#include <vector>

#include <algorithm>

#include <cmath>

using namespace std;

int main() {

    int N;

    cout << "Enter the number of columns: ";

    cin >> N;

    vector<int> row1(N), row2(N);

    cout << "Enter the values for the first row (space-separated): ";

    for (int i = 0; i < N; ++i) {

        cin >> row1[i];

    }

    cout << "Enter the values for the second row (space-separated): ";

    for (int i = 0; i < N; ++i) {

        cin >> row2[i];

    }

    vector<int> dp(N + 1, 0);

    for (int i = 0; i < N; ++i) {

        dp[i + 1] = max(dp[i + 1], dp[i] + abs(row1[i] - row2[i]));

        if (i < N - 1) {

            int score = abs(row1[i] - row1[i + 1]) + abs(row2[i] - row2[i + 1]);

            dp[i + 2] = max(dp[i + 2], dp[i] + score);

        }

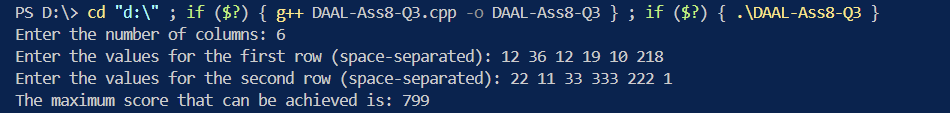
    }

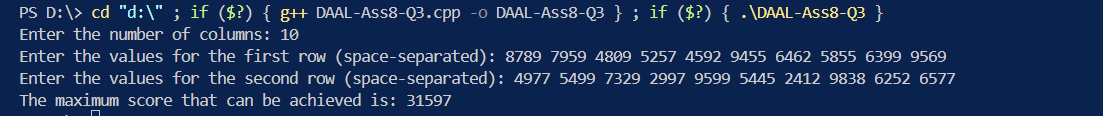
    cout << "The maximum score that can be achieved is: " << dp[N] << endl;

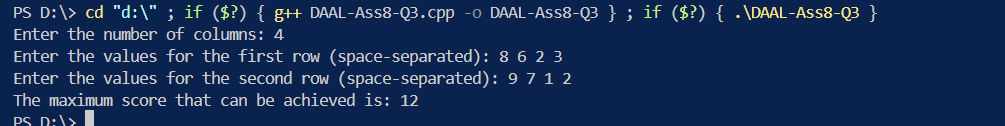
    return 0;

}

Output to verify test cases :







**Analysis of Complexity :**

* **Time Complexity:**
  + Best Case: O(N)
  + Average Case: O(N)
  + Worst Case: O(N)
* **Space Complexity:** O(N)