Batch:T6

Practical No.8

Title of Assignment: Dynamic Programming Algorithms

Student Name: Parshwa Herwade

Student PRN: 22510064

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

Pseudocode:

function LIS(arr, n):

dp = array of size n, initialized to 1

for i = 1 to n-1:

for j = 0 to i-1:

if arr[i] > arr[j]:

dp[i] = max(dp[i], dp[j] + 1)

return max value in dp

CODE:  
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int LIS(const vector<int>& arr) {

    int n = arr.size();

    vector<int> dp(n, 1);

    for (int i = 1; i < n; ++i) {

        for (int j = 0; j < i; ++j) {

            if (arr[i] > arr[j]) {

                dp[i] = max(dp[i], dp[j] + 1);

            }

        }

    }

    return \*max\_element(dp.begin(), dp.end());

}

int main() {

    int n;

    cin >> n;

    vector<int> arr(n);

    for (int i = 0; i < n; ++i) {

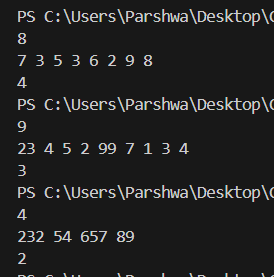
        cin >> arr[i];

    }

    cout << LIS(arr) << endl;

    return 0;

}

**OUTPUT:  
**

**Time and Space Complexity:**

* **Best case time complexity**: O(n) (when array is already sorted in increasing order)
* **Average case time complexity**: O(n²)
* **Worst case time complexity**: O(n²) (when array is sorted in decreasing order)
* **Space complexity**: O(n) (for the dp array)

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 ≤ 109

1 ≤ wi ≤ x

Example

Input:

4 10

4 8 6 1

Output:

2

Pseudocode:

function minElevatorRides(n, x, weights):

dp = array of size (1 << n), initialized to infinity

sumWeights = array of size (1 << n), initialized to 0

dp[0] = 1

for mask = 1 to (1 << n) - 1:

for i = 0 to n-1:

if mask has i-th bit set:

prevMask = mask ^ (1 << i)

if sumWeights[prevMask] + weights[i] <= x:

dp[mask] = dp[prevMask]

sumWeights[mask] = sumWeights[prevMask] + weights[i]

else:

dp[mask] = dp[prevMask] + 1

sumWeights[mask] = weights[i]

return dp[(1 << n) - 1]

CODE:

#include <iostream>

#include <vector>

#include <limits.h>

using namespace std;

pair<int, int> minElevatorRides(int n, int x, const vector<int>& weights) {

    int totalMasks = 1 << n;

    vector<pair<int, int>> dp(totalMasks, {n + 1, 0});

    dp[0] = {1, 0};

    for (int mask = 1; mask < totalMasks; ++mask) {

        for (int i = 0; i < n; ++i) {

            if (mask & (1 << i)) {

                int prevMask = mask ^ (1 << i);

                pair<int, int> option = dp[prevMask];

                if (option.second + weights[i] <= x) {

                    option.second += weights[i];

                } else {

                    option.first += 1;

                    option.second = weights[i];

                }

                dp[mask] = min(dp[mask], option);

            }

        }

    }

    return dp[totalMasks - 1];

}

int main() {

    int n, x;

    cin >> n >> x;

    vector<int> weights(n);

    for (int i = 0; i < n; ++i) {

        cin >> weights[i];

    }

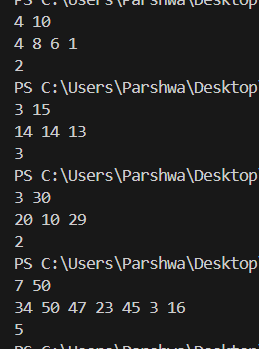
    pair<int, int> result = minElevatorRides(n, x, weights);

    cout << result.first << endl;

    return 0;

}

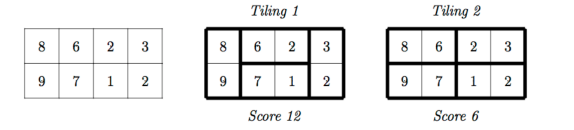
OUTPUT:



**Time and Space Complexity:**

* **Best case time complexity**: O(n \* 2ⁿ) (DP solution iterating over all subsets)
* **Average case time complexity**: O(n \* 2ⁿ)
* **Worst case time complexity**: O(n \* 2ⁿ) (always needs to iterate over all subsets)
* **Space complexity**: O(2ⁿ) (for storing dp and sumWeights arrays)

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.

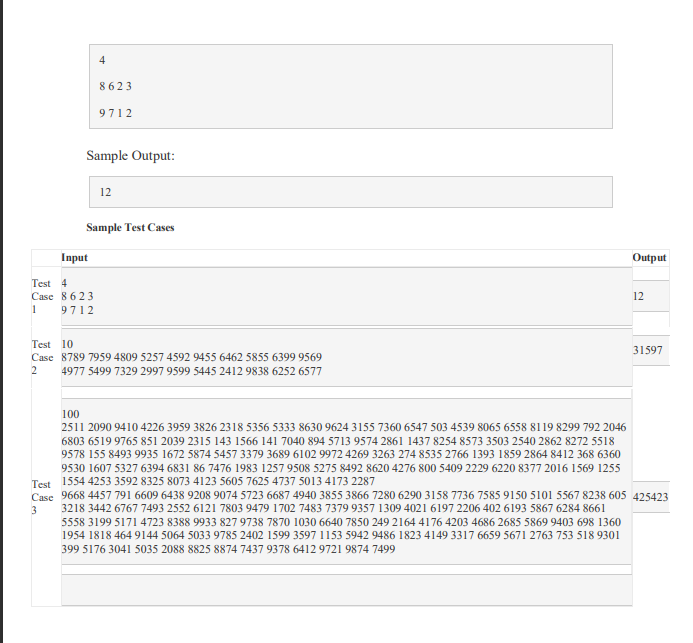
Input format

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:



Pseudocode:

function maxScore(grid, N):

    dp = array of size N+1, initialized to 0

    for i = 1 to N:

        if i > 1:

            dp[i] = max(dp[i], dp[i-2] + abs(grid[0][i-1] - grid[0][i-2]) + abs(grid[1][i-1] - grid[1][i-2]))

        dp[i] = max(dp[i], dp[i-1] + abs(grid[0][i-1] - grid[1][i-1]))

    return dp[N]

CODE:  
#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

int maxScore(const vector<vector<int>>& grid, int N) {

    vector<int> dp(N + 1, 0);

    for (int i = 1; i <= N; ++i) {

        if (i > 1) {

            dp[i] = max(dp[i], dp[i-2] + abs(grid[0][i-1] - grid[0][i-2]) + abs(grid[1][i-1] - grid[1][i-2]));

        }

        dp[i] = max(dp[i], dp[i-1] + abs(grid[0][i-1] - grid[1][i-1]));

    }

    return dp[N];

}

int main() {

    int N;

    cin >> N;

    vector<vector<int>> grid(2, vector<int>(N));

    for (int i = 0; i < 2; ++i) {

        for (int j = 0; j < N; ++j) {

            cin >> grid[i][j];

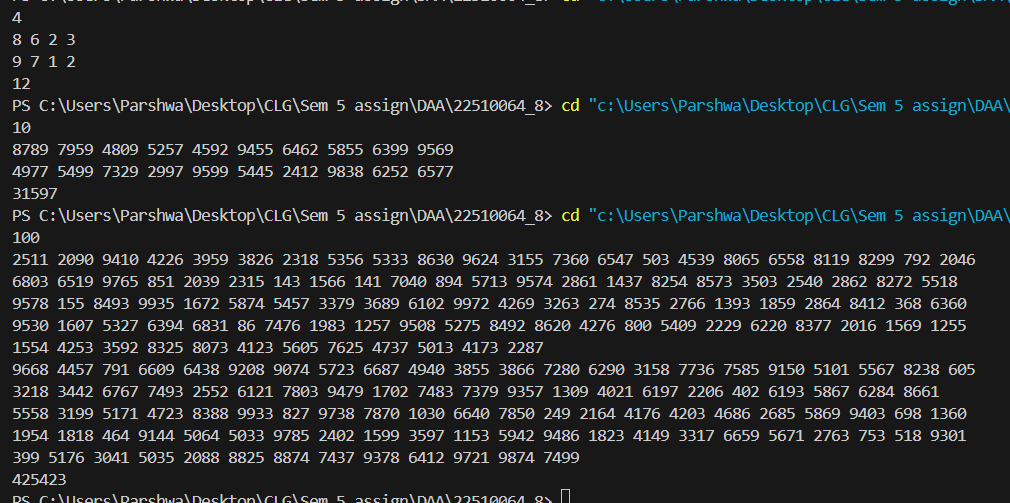
        }

    }

    cout << maxScore(grid, N) << endl;

    return 0;

}

OUTPUT:  


**Time and Space Complexity:**

* **Best case time complexity**: O(N) (as we are iterating over all columns once).
* **Average case time complexity**: O(N)
* **Worst case time complexity**: O(N)
* **Space complexity**: O(N) (for storing dp array and input grid).