Batch: T3

Assignment No.: 7

Title of Assignment: Dynamic Programming

Student Name: Piyush Dawkhare

PRN: 23510056

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 \* 105

1 ≤ xi ≤109

**Example**

Input: 8 7 3 5 3 6 2 9 8

Output: 4

1. Technologies/libraries/algorithm used:

* #include algorithms
* #include <bits/stdc++.h> //This library includes these two libraries

1. Related theory or any related information:

The **Longest Increasing Subsequence (LIS)** problem is a classic **Dynamic Programming (DP)** problem.

* A subsequence keeps the order of elements but may skip some.
* In LIS, each next element must be **strictly larger** than the previous.

Example:  
Array = [7, 3, 5, 3, 6, 2, 9, 8]

LIS = [3, 5, 6, 9]

(length = 4)

1. Algorithms

* Define dp[i] = length of LIS ending at index i.
* Formula:
  + dp[i]=1+max(dp[j])
  + for all j<i and arr[j]<arr[i]
* Initialize all dp[i] = 1 (each element alone is length 1).
* Answer = max(dp[i]).
* **DP O(n²)**: Works for n ≤ 5000.

1. Code:
2. #include <bits/stdc++.h>
3. using *namespace* std;
4. *int* main() {
5. *int* n;
6. cin >> n;
7. vector<*int*> arr(n);
8. for (*int* i = 0; i < n; i++) cin >> arr[i];
9. vector<*int*> dp(n, 1);
10. *int* lis = 1;
11. for (*int* i = 1; i < n; i++) {
12. for (*int* j = 0; j < i; j++) {
13. if (arr[j] < arr[i]) {
14. dp[i] = max(dp[i], dp[j] + 1);
15. }
16. }
17. lis = max(lis, dp[i]);
18. }
19. cout << lis << "\n";
20. return 0;
21. }

5. INPUT:

8

7 3 5 3 6 2 9 8

6. OUTPUT:

4

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 ≤ 109

1 ≤ wi ≤ x   
**Example**

Input: 4 10 4 8 6 1

Output: 2.

1. Technologies/libraries/algorithm used:

* Libraries:
  + <bits/stdc++.h> → includes all standard C++ libraries.
  + vector, pair for DP storage.
* Algorithm:
  + Dynamic Programming with Bitmasking
  + State representation = subset of people already assigned to rides.
  + Transition chooses whether to place a new person in the same ride or start a new ride.

1. Related theory or any related information:

* The **problem** is a variation of the **bin packing problem**.
* Each person is like an **item** with weight w[i].
* The elevator is like a **bin** with capacity x.
* We want the **minimum number of bins (rides)** to carry all people.
  + Since n≤20n \leq 20n≤20, we can use **bitmask DP** because 220=1,048,5762^{20} = 1,048,576220=1,048,576, which is manageable.
  + Complexity: **O(n·2^n)** → ~20 million operations at most.

1. Algorithms:

* Read n (number of people) and x (elevator capacity).
* Store all weights in an array w[].
* Initialize DP array of size 2^n with (n+1, 0).
  + dp[0] = (1, 0) → 1 empty ride, weight = 0.
* For each mask (subset of people already placed):
* For each person not in mask:
  + Create new\_mask = mask | (1<<i).
  + If last\_weight + w[i] ≤ x → stay in same ride.
  + Else → start new ride.
  + Update dp[new\_mask] with minimum rides.
* Final answer = dp[(1<<n)-1].first.

1. Code:
2. #include <bits/stdc++.h>
3. using *namespace* std;
4. *int* main() {
5. *int* n, x;
6. cin >> n >> x;
7. vector<*int*> w(n);
8. for (*int* i = 0; i < n; i++) cin >> w[i];
9. sort(w.begin(), w.end());
10. *int* i = 0, j = n - 1;
11. *int* rides = 0;
12. while (i <= j) {
13. if (w[i] + w[j] <= x) {
14. i++; *// lightest also fits*
15. }
16. j--; *// heaviest always goes*
17. rides++;
18. }
19. cout << rides << "\n";
20. return 0;
21. }

5. Input / Output:

**Input**:

4 10

4 8 6 1

**Output**:

2

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.

*\*Rest Question is NOT Shown Here*

1. Technologies/libraries/algorithm used:

* Use **Dynamic Programming (DP)**.
* At each column, you can either:
  + Place **one vertical domino**.
  + Place **two horizontal dominoes** (using two consecutive columns).

1. Related theory or any related information:

* A domino covers **two numbers**.
* **Score = |larger - smaller|**.
* We want the **maximum total score** when covering the entire 2×N grid.
* DP State:
  + dp[i] = maximum score possible up to column i.
* Transitions:
  + If we use a **vertical domino** at column i:
    - dp[i]=dp[i−1]+∣top[i]−bottom[i]∣dp[i] = dp[i-1] + |top[i] - bottom[i]|dp[i]=dp[i−1]+∣top[i]−bottom[i]∣
  + If we use **two horizontals** at columns (i-1, i):
  + dp[i]=dp[i−2]+∣top[i−1]−top[i]∣+∣bottom[i−1]−bottom[i]∣dp[i] = dp[i-2] + |top[i-1] - top[i]| + |bottom[i-1] - bottom[i]|dp[i]=dp[i−2]+∣top[i−1]−top[i]∣+∣bottom[i−1]−bottom[i]∣
* Answer = dp[N].

1. Algorithms:

* Read number of columns N.
* Read top row and bottom row values.
* Initialize DP array.
* Base cases:
  + dp[1] = |top[0] - bottom[0]|
  + dp[2] = max(verticals, horizontals)
* Loop from column 3 to N and update using recurrence.
* Print dp[N].

1. Code:

*#include <bits/stdc++.h>*

*using namespace std;*

*int main() {*

*int N;*

*cin >> N;*

*vector<int> top(N), bottom(N);*

*for (int i = 0; i < N; i++) cin >> top[i];*

*for (int i = 0; i < N; i++) cin >> bottom[i];*

*vector<int> dp(N+1, 0);*

*// Base cases*

*dp[1] = abs(top[0] - bottom[0]);*

*if (N > 1) {*

*dp[2] = max(*

*dp[1] + abs(top[1] - bottom[1]),*

*abs(top[0] - top[1]) + abs(bottom[0] - bottom[1])*

*);*

*}*

*// Fill DP*

*for (int i = 3; i <= N; i++) {*

*int vertical = dp[i-1] + abs(top[i-1] - bottom[i-1]);*

*int horizontal = dp[i-2] + abs(top[i-2] - top[i-1]) + abs(bottom[i-2] - bottom[i-1]);*

*dp[i] = max(vertical, horizontal);*

*}*

*cout << dp[N] << "\n";*

*return 0;*

*}*

6. Output:

12