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:

$1 \le n \le 2 * 105$

$1 \le xi \le 109$

Example

Input:

8

7 3 5 3 6 2 9 8

Output:

4

Pseudocode:

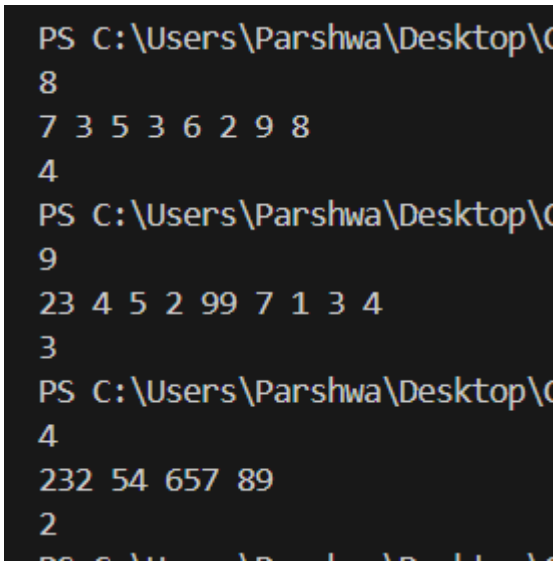
function LIS(arr, n):

```
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:

```cpp
#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**: $O(n^2)$
- **Worst case time complexity**: $O(n^2)$ (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:

$1 \le n \le 20$

$1 \le x \le 109$

$1 \le wi \le x$

Example

Input:

4 10

4 8 6 1

Output:

2

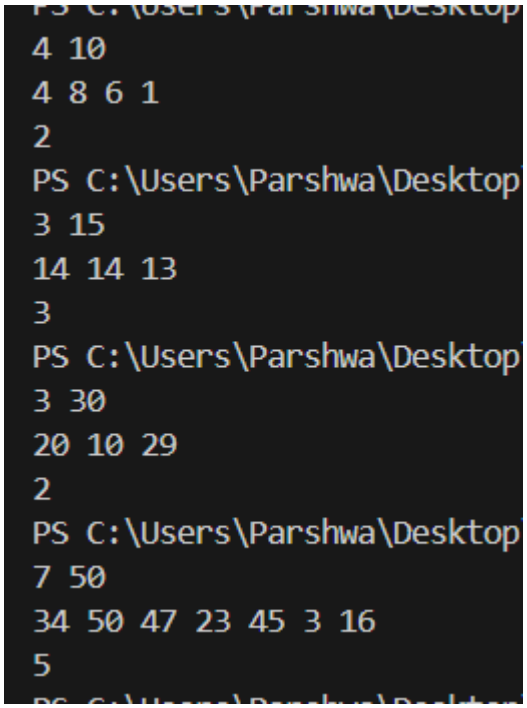
Pseudocode:

```
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:

```cpp
#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 * 2^n) (DP solution iterating over all subsets)
- **Average case time complexity**: O(n * 2^n)
- **Worst case time complexity**: O(n * 2^n) (always needs to iterate over all subsets)
- **Space complexity**: O(2^n) (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:

```
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

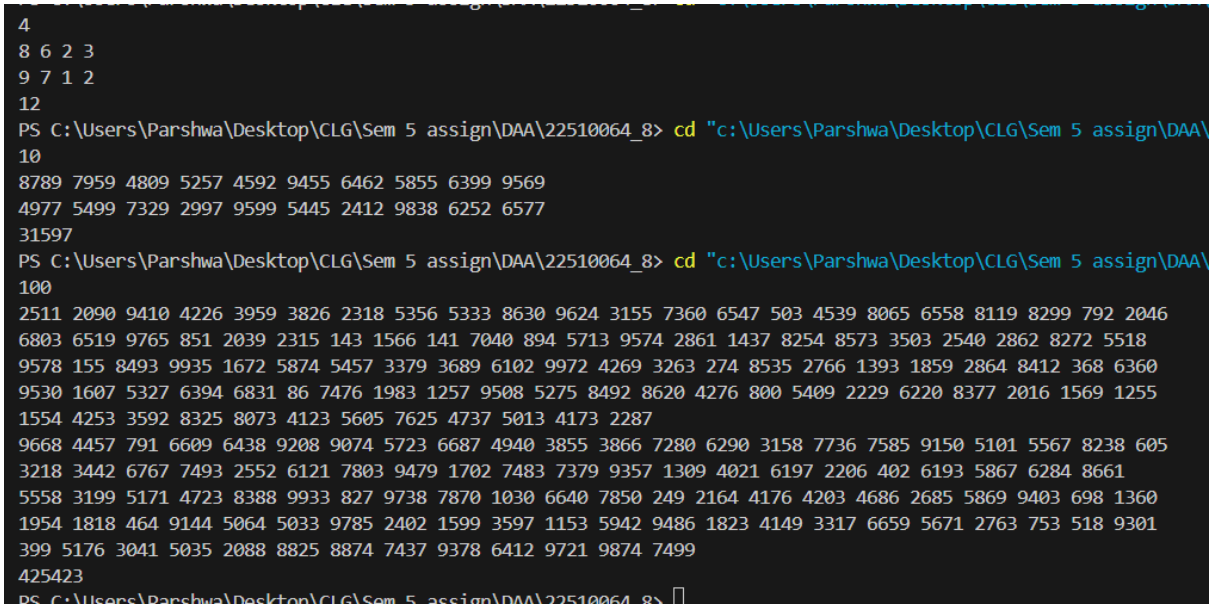
Pseudocode:

```
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).