HANDS ON 3 REPORT

Competitive Programming and Contests

University of Pisa, Academic Year 2024-2025

SALMI ABDERRAHMANE

a.salmi1@studenti.unipi.it

Matricola 704608

Problem #1: Holiday Planning

Approach

To solve this problem efficiently, we used a Dynamic Programming approach, in which we compute the maximum number of attractions that can be visited over a given number of days (D) across multiple cities (n).

State Definition:

We created a dynamic programming 2D table **dp[days][city]**
The value of **dp[d][c]** represents the maximum number of attractions we can visit using d days in the city c.

Base Case:

- **If d = 0 (no days)**, dp[0][c] = 0 for all c because we can't visit attractions without any days.
- **If c = 0 (no cities)**, dp[d][0] = 0 for all d because there are no cities to visit.

Recurrence:

For each **d** (total days) and **c** (current city), we must decide how many days to spend in city c, and we do that by choosing the max number of attractions gained by either:

- **Visiting the city**: add the number of attractions of the current city with the current number of spent to the previous best result before spending the number days
- **Skipping the city**: keep the same number of attractions already planned for the previous cities

```
for days_spent in 0..=days {
    dp[days][city] = dp[days][city].max(
        dp[days - days_spent][city - 1] + attractions[city - 1][days_spent]
    );
}
```

Result:

The value of **dp[D][n]** (all days and all cities considered) is the maximum number of attractions we can visit.

Implementation

Here are the steps of our implementation:

1. **Initialize the DP Table**: The table is initialized to zeros.
2. **Fill the DP Table**: For each city and day combination, the table is updated by checking all possible days to spend in the current city and ensuring the maximum attractions are calculated by comparing previous results (we always take the max number of attractions).
3. **Final Output:** After processing all cities and days, the result is found in **dp[D][n]**, which gives the maximum number of attractions possible for the given input.

Complexity Analysis

Time Complexity:

The time complexity of this solution is **O(n * D^2)**:

- **Outer Loop**: Iterate through all `d` days → **O(d)**.
- **Inner Loops**:
    - Iterate through **n** cities → **O(n)**.
    - Calculate prefix sums for each city → **O(d)**.
    - Spend **days_spent** days in the current city → **O(d)**.

Space Complexity:

The space complexity is **O(n * D)**, which is the size of the DP table used to store the results for each combination of days and cities, and of course we have the prefix sum array that has a complexity of O(d).

Problem #2: Design a course

Approach

Similarly to the previous problem, we used a Dynamic Programming approach to solve this problem efficiently. When reading the problem description, we can notice that it can be solved using the Longest Increasing Subsequence (LIS) solution, which involves:

1. **Sorting Topics:** The topics are sorted first by beauty in increasing order, and then by difficulty in descending order if their beauty is the same. This sorting ensures valid subsequences when calculating the Longest Increasing Subsequence (LIS).
2. **Applying LIS on Difficulty:** After sorting, the goal becomes finding the Longest Increasing Subsequence (LIS) based on difficulty, where each topic can only connect to those with a higher difficulty.
3. **Dynamic Programming:** A DP table **lis[]** is used, where **lis[i]** is the length of the LIS ending at **topic i**. For each topic **i**, we compare it with all previous topics **j**. If the difficulty of topic i is greater than that of topic j, we update lis[i] using the formula: **lis[i]** = **max(lis[i], lis[j]** + **1)**.
4. **Final Result:** The max number of selected topics is the max value in the lis array.

Implementation

Here are the steps of our implementation:

1. **Initialize the DP Table**: The table is initialized to ones (1s), because initially, each topic can at least form a subsequence of length 1 (itself).
2. **Sorting:** Sort topics by beauty in ascending order, and by difficulty if they have the same beauty.
3. **Fill the DP Table (lis)**: Iterate through each topic i and compare it with all previous topics j to compute the LIS.
4. **Final Output:** After processing all topics, return the maximum value from the lis array.

```rust
fn max_topics(topics: Vec<(usize, usize)>, n: usize) -> usize {
    let mut lis = vec![1; n];

    let mut sorted_topics = topics.clone();
    sorted_topics.sort_by(|a, b| a.0.cmp(&b.0).then_with(|| b.1.cmp(&a.1)));

    for i in 1..n {
        for j in 0..i {
            if sorted_topics[i].1 > sorted_topics[j].1 {
                lis[i] = lis[i].max(lis[j] + 1);
            }
        }
    }

    lis.iter().max().unwrap().clone()
}
```

Complexity Analysis

Time Complexity:

The time complexity of this solution is **O(n^2)**:

- **Sorting**: O(n log n).
- **LIS Calculation**:
    - Outer loop: O(n)
    - Inner loop: O(n)
    - Total: O(n^2)

Space Complexity:

The space complexity is **O(n)**, which is the size of the DP table (lis).