HandsOn 3

Studente: Luca Rizzo
Matricola: 598992

Exercise 1

I modeled the problem as a knapsack problem in which each city corresponds to an item, but with multiple choices available for each city (unlike the standard 0/1 case). For every city, I can choose it in D different ways, depending on how many days I decide to stay there. Each choice consumes a certain number of available days and yields a corresponding benefit in terms of visited attractions. The input sequence `cities_attrs` imposes an artificial ordering when processing the cities, which does not correspond to any actual constraint on the order in which the tourist must visit them. In particular, I define the DP state as dp[i][j]=maximum number of attractions achievable using the first i cities of input and j available days. To clarify, if I decide to stay k days in city i, the corresponding benefit is the sum of the first k entries of its itinerary. This value is added to the optimal solution achievable with the remaining D−k days using only the previous cities. The recurrency is the following:

$$dp[i][j] = \max\left(dp[i-1][j], \max_{\text{k=1...j}}\left(dp[i-1][j\ -k] + \sum_{t=1}^{k} attrs[i][t] \right)\right)$$

The first term corresponds to the choice of not spending any day in the current city, while the second term represents the best among the D possible choices of staying exactly k out of j days in that city. Since we fill a table of n * D cells and, for each cell, we iterate over at most D possible choices, the overall **time complexity is** $\Theta(n * D^2)$**.** To reduce the space usage from $\Theta(n * D)$ to $\Theta(D)$, I do not store the full matrix. Because the recurrence only depends on the previous row, and we do not need to reconstruct the sequence of choices, we can keep only two rows at any time: the current row and the previous one, swapping them when moving from one city to the next. In this case, the space complexity is $\Theta(D)$. This DP formulation satisfies the required time and space constraints and matches the expected solution format.

Since for each city considering all possible choices is equivalent to having D different “versions” of the same object, each with its own cost (the number of days spent) and its own benefit (the cumulative attractions obtained by staying exactly k days in that city), the problem can be interpreted as a 0/1 knapsack instance with n*D items. In this view, each city i corresponds to D items, where item j represents the choice of spending exactly j days in that city. This formulation requires enforcing the constraint that no two items belonging to the same city can be chosen simultaneously. In practice, when processing the item corresponding to j days in city i, the DP transition should refer to the DP state corresponding to exactly j fewer days (i.e., using row k−j as predecessor).
This would require keeping multiple DP rows available at once leading to a higher memory usage compared to the more compact two-row DP formulation adopted earlier.

Exercise 2

In this problem, the professor wants to select the longest sequence of topics that is strictly increasing in both beauty and difficulty. This naturally suggests computing a Longest Increasing Subsequence (LIS), but applied in a way that enforces monotonicity over two attributes rather than just one. In fact, the classical LIS algorithm operates on a one-dimensional sequence and assumes that the order of the input already encodes the structural constraint we want to enforce. In other words, LIS can only check monotonicity along a single dimension (the one we feed into the algorithm), while the ordering of the sequence must already ensure that the other dimension is consistent.
For this reason, we impose an ordering that makes the sequence "safe" with respect to difficulty: we sort the topics by increasing difficulty and, in case of ties, by decreasing beauty. This tie-breaking rule ensures that two topics with equal difficulty cannot both appear in an increasing subsequence on beauty because the greater appears first, thus enforcing strict monotonicity in difficulty without explicitly checking it. After sorting, it is sufficient to compute the LIS on the beauty values. The **correctness** follows from the fact that sorting by difficulty produces a sequence that is already strictly increasing in this attribute, and therefore contains every feasible subsequence that is increasing in both difficulty and beauty. Once this ordering is fixed, computing the LIS on the beauty values explores exactly all such feasible subsequences. Since the LIS has maximum length among all increasing subsequences, the value it returns corresponds to the maximum possible number of topics satisfying both constraints.

To compute the LIS efficiently, I applied the greedy algorithm discussed in class, which maintains an array of dominant tail values. This array, of size at most nnn, stores at position kkk the smallest possible tail of any increasing subsequence of length k+1. The correctness of this greedy algorithm relies on the fact that extending the subsequence with the smallest available tail is always optimal: a smaller tail leaves more room for future extensions. By a standard exchange argument, any optimal solution using a larger tail can be transformed into one using the greedy choice without reducing its length, which justifies maintaining the dominant array.

For each beauty value of the sorted topics, we perform a binary search (via `partition_point`) on this array. If we find an index `p < dominant.len()`, we replace `dominant[p]` with the new value, improving the tail of subsequences of length p+1. If p=dominant.len, we append the value, thereby extending the longest subsequence found so far. At the end, the length of the dominant array equals the length of the LIS, which corresponds to the maximum number of topics the professor can include in the course.

This solution requires $\Theta(n * log(n))$ time to sort the n input topics by difficulty. After sorting, the greedy LIS algorithm performs n binary searches on an array `dominant` that is always sorted and has size at most n. Each binary search takes O(logn) time, for an additional $\Theta(n * log(n))$ term.

Therefore, the overall time complexity of the algorithm is $\Theta(n * log(n))$. An additional O(n) space is required to store a copy of the input array, so that the topics can be sorted without modifying the original input.