Hands On 3 Report

This brief report outlines the solution for the first hands-on assignment of the Competitive Programming course, focused on dynamic programming.

Requirements

The assigment required us to solve 2 problems:

Holiday Planning

We're tasked to plan an holiday of d days in n cities. For each city we know the number of attractions $a_{i,d}$ that we can visit in that city i, spending d days there. We want to know whats the maximum number of attractions we can visit.

Design a Course

We're given a list of n topics, each one having an interest and difficulty factor: $t.i, t.d$. We wan't to know what's the maximum number of topics that we can order so that each one has increasing difficulty and interest than the precedent.

Holiday Planning Solution

From the requirements, we see that this is a maximization problem, solving it by a bruteforce approach would require an exponential number of steps. Generally a dynamic programming approach helps us solving these kinds of problems by decomposing them in subinstances that overlap with each other. So we start by the smallest sub-instance (generally a base case) and we construct the solutions "from the ground up", forming the ones for sub-instance i from a general composition and update from solutions $i-1$.

Maximizations problems often come with constraints that the solution must respect in order to be accepted. We start by identifying constraints.

In this case, we have one main constraint in the way we compose our sub-instances. This is the fact that, given a city c and a holiday of d days, we can't "cherry pick" the days with the maximum $a_{i,d}$ but we're forced to process them from left to right.

With this constrain we have to find a way to express the number of attraction $A_i[\,k\,]$ that we can visit in the city c_i spending there exactely $k \in [0, d]$ days. Since we're given for each c_i a list with the number of attraction $a_{i,j}$ for each day j, we find $A_i[\,k\,]$ as the prefix-sum of all items a_i up to j.

We can represent this as:

$$A_i[\,k\,] = \sum_{j=0}^{k} a_{i,j}$$

To solve the problem, we apply dynamic programming by computing, the maximum number of attractions visitable considering the fist $i \in [0, n]$ cities in exactely $j \in [0, d]$ days.

Programmatically we use a matrix $M_{(n+1)\times(d+1)}$. Each cell of the matrix will be filled with the solution of the corresponding sub-instance of the problem.

We start by initializing the first row and column of the matrix with 0 since:

- $\forall i \in [0, d] \parallel M[0][i]$ is the maximum attractions visitable in i days considering 0 cities, which would be 0.
- $\forall j \in [0, n] \parallel M[j][0]$ is the maximum attractions visitable considering j cities, in 0 days, which would be 0.

Defined the base case, we have to choose how to compute a general solution $M[i, j]$. We remember we want to maximize the number of attractions, by starting from the solutions $M[i-1]$, and try adding the new city c_i. We find the new maximum by considering that we can choose to spend the first $j-k$ (with $k \in [0, j]$) days in the previous c_{i-1} cities (visitin $M[i-1][j-k]$ attractions), while spending the remaining k days in the new city c_i, visiting $A_i[k]$.

This allows us to assemble the general dynamic programming recurrence relation as:

$$M[i,j] = \begin{cases} 0 & (i = 0 \ \vee \ j = 0) \\ \max\limits_{k \in [0,j]} \left(M[i-1][j-k] + A_i[k] \right) & (\text{otherwise}) \end{cases}$$

After computing each cell of the matrix, we can find the solution to the problem instance as $\max\limits_{i \in [0,d]}(M[i,d])$.

Complexity

To employ this algorithm we basically require 2 matrices: $A_{c \times d}$ to store the prefix sums for each city c_i up to index d and the memoization matrix $M_{(n+1)\times(d+1)}$. We find the overall additional space needed as:

$$S(n,d) = \Theta(nd)$$

Where n is the number of cities set and d is the number of days of the holiday.

For the time complexity we consider that for each cell $M[i,j]\ \ (i,j \neq 0)$ of the memoization matrix we need to scan the the first j items of the row $i-1$ in order to compute the maximum. We find the time complexity of compiling the matrix as:

$$T_M(n,d) = n\left(\sum_{j=1}^{d} j\right) = n\frac{d^2+d}{2} \sim O(nd^2)$$

Other than that we take $\Theta(nd)$ to compute the prefix sum matrix A, so the total time complexity is:

$$T(n,d) = \Theta(nd) + O(nd^2) \sim O(nd^2)$$

Implementation and Test Suite

Regarding implementation specifics I produced the following:

- `plan(&[usize],usize,usize) -> usize` that takes a flattened matrix (`&[usize]`) (row-major-order) where each row contains the ordered list of attraction for city i, and two `usize`'s, respectively the number of cities and days, and returns an `usize` with the maximum numbers of attraction visitable.
- `parse_input(&str) -> (Vec<usize>,usize,usize)`: that given a static string, returns a tuple containing the parameters in order for above described `plan(...)` method. The method is compliant with the test format defined in the requirements paper.
- `parse_output(&str) -> usize`: that given a static string, returns an `usize` consisting in the expected output for an input, compliant with the test format defined in the requirements paper.

As for validation, the base crate directory, contains a `test_holiday` directory, with all input and output files given in the requirements page. A test method was defined, that, for each input-output couple of files compares the output of the `plan(...)` method with the expected output.

Course Design Solution

A key insight in the solution of this problem, was to see the similarities between the requirements and a `longest_increasing_subsequence` (LIS from now on) instance dependant on two parameters.

A general version of LIS is a standard problem that, given a list of values (generally numerical), requires to determine the maximum length of an increasing subsequence in the list.

An increasing subsequence can be defined as an ordered sequence of list items where:

$$l_i < l_j \parallel \forall i,j \in [1,n]^2 . i < j$$

This looks really similar as the requirements of the problem.

$$\forall i,j \in \{1,\dots,n\},\ i<j \Rightarrow (t_i.\text{interest} < t_j.\text{interest} \wedge t_i.\text{difficulty} < t_j.\text{difficulty})$$

From now on we'll refer, as the problem to find the longest 2 parameter increasing subsequence as LIS-2D, and we'll use LIS-1D for the normal LIS instance.

Reducing LIS-2D to LIS-1D

Since we know some performant ways to solve LIS-1D, we try to find a ways to reduce the LIS-2D instance into something equivalent.

Let's consider a list l_1 of simple numerical values (that can constitute a LIS-1D input). We can construct a list l_2 of 2-dimensional values, so that the length of the LIS of l_1 is the same as the LIS-2D of l_2.

$$\{3,2,4,5,4,7\} \overset{\text{LIS}}{\equiv} \{(0,\underline{3}),(1,\underline{2}),(3,\underline{4}),(4,\underline{5}),(5,\underline{7})\}$$

We use $L_1 \overset{\text{LIS}}{\equiv} L_2$ to say that 2 lists have the same longest increasing subsequence length.

The equivalence also works in reverse; if we implicitly suppose an ordering so that we can construct a LIS-2D solution by a LIS-1D solution, we can enforce an ordering on the 2 dimensional input so that 1 parameter can be taken out of the equation.

As the example suggest, we can sort the input list in ascendent order by the first parameter. While this might seem true it's only half the answer. In the example before, I explicitly supposed an ordering that would make it easy to see the strategy of reduction.

In practice, we have to put attention to the fact that pairs can have duplicate sort parameters, so we have to define how the sorting happens for such pairs.

Let's consider a sequence of pairs (a, b) that has already been sorted in ascendent order by parameter a. If (a_1, b_1) comes before (a_2, b_2) in the sorted sequence, then it means that $a_1 \leq a_2$. For a valid LIS-2D solution we need $a_1 < a_2$ which means we have to find a way to prior exclude invalind LIS extension that would allow for tuples where $a_1 = a_2$.

In order to do this we have to enforce a decreasing ordering on the second pair parameter. This prevents choosing both pairs so that $a_1 = a_2$, since we would be choosing a pair with smaller second parameter violating the LIS-1D property on the second coordinate.

To show how the ordering affects the reduction, let's consider a list L where the LIS-2D has length of 1, and two sorting alternatives L_{s_1}, L_{s_2}.

$$L = \{(3,1),(3,2),(2,3)\},\ \mathrm{LIS_{2D}}\,(L) = 1 \| L \rightarrow \begin{cases} L_{s_1} = \{(2,3),\underline{(3,1),(3,2)}\} \\ L_{s_2} = \{(2,3),\underline{(3,2),(3,1)}\} \end{cases}$$

Now let's reduce the lists to LIS-1D instances by considering only the second parameters of each pair.

$$L_{s_1} = \{3,\underline{1,2}\},\ \mathrm{LIS(L}_{s_1}) = 2$$

$$L_{s_2} = \{3,2,1\},\ \mathrm{LIS(L}_{s_1}) = 1$$

$$L \overset{\mathrm{LIS}}{\equiv} L_{s_2}$$

We showed a way to reduce the longest increasing subsequence from 2 to 1 dimension.

Solving LIS

Now we're able to tackle the LIS problem in one dimension. This instance is a common problem that gets solved efficiently using dynamic programming.

We start by defining the LIS length of the list truncated at the first index as 1. With that as base case, we compute the solution for the list truncated at increasing index by using the previous solutions. For each item at index $j < i$, we can update our current solution if $L[j]$ is less than $L[i]$. We use the current element as extension for a previously computed LIS.

This is the reasoning behind the quadratic approach that requires us to visit each intex $j < i$ to compute each solution. This requires a quadratic number of iterations in relation to input length.

A more efficient approach relies on additional storage initialized as empty to accumulate items, keeping them sorted, to use binary search to update the LIS for every index i.

We define binary seach as a method (that executes in a logarithmic number of steps in relation to input size), that given a list and and a target, returns the index of apperance of the target in the list (for this particular problem we don't care about left-right-most apperance). If the item is not present in the list, then it returns the first index where we can insert the element in order to keep the list sorted.

We can use this, to replace items in the current LIS, and to extend the list if the index returned is out of bounds.

Complexity

We find asymptotic complexity for this solution to be:

$$T(n) = \Theta(n \log(n)) \qquad S(n) = \Theta(n)$$

Implementation and Test Suite

Regarding implementation specifics I produced the following:

- `design(&[(i64,i64)]) -> usize` that takes a slice of topics as pairs of `i64` and returns an `usize` with the LIS length.
- `parse_input(&str) -> (Vec<(i64,i64)>)`: that given a static string, returns a vector of pairs. The method is compliant with the test format defined in the requirements paper.
- `parse_output(&str) -> usize`: that given a static string, returns an `usize` consisting in the expected output for an input test, compliant with the test format defined in the requirements paper.

As for validation, the base crate directory, contains a `test_course` directory, with all input and output files given in the requirements page. A test method was defined, that, for each input-output couple of files compares the output of the `design(...)` method with the expected output.