COS 330: Great Ideas in Theoretical Computer Science

Fall 2025

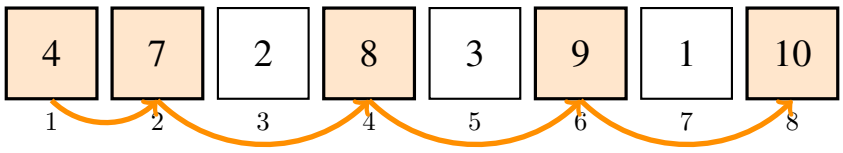
Precept 2

My Dynamic Precept

Learn

Let's recall the *Longest Increasing Subsequence (LIS)* problem from class. We are given a sequence of integers $a_1, a_2, \ldots, a_n$ and we want to find the longest increasing subsequence. An *increasing subsequence* is a sequence that can be derived from the original sequence by deleting some elements without changing the order of the remaining elements, such that each element in the subsequence is less than the next one. Another way to define is to say that an increasing subsequence is given by a set of indices $i_1 < i_2 < \ldots < i_k$ such that $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

Here is an example with $n = 8$, with its unique longest increasing subsequence highlighted (not all sequences have a single unique LIS):



We saw one way to solve this problem in $O(n^2)$ time using dynamic programming. We defined

$$L_i = \text{length of the longest increasing subsequence ending at index } i.$$

For the above instance this gives $L = [1, 2, 1, 3, 2, 4, 1, 5]$. We can compute L_i by using the following recurrence:

$$L_i = 1 + \max_{j<i, a_j<a_i} L_j,$$

with the base case $L_1 = 1$. Translating the above recurrence to English, we say that the longest increasing subsequence ending at index i is given by taking the longest increasing subsequence ending at some index $j < i$ where $a_j < a_i$ (so that we can append a_i to the subsequence ending at j) and adding 1 to account for a_i. If there is no such j, then we just have the subsequence of length 1 containing only a_i. The answer to the problem is then $\max_i L_i$.

> Note that in the definition of L_i we force it to include a_i. This is critical otherwise we would not be able to know how to extend the subsequence. Forcing an element to be included in a dynamic programming state is a common trick in dynamic programming problems where the order of elements matters in some way.
>
> A consequence of this is that the last state isn't necessarily the optimal one, since the optimal subsequence might not include the last element of the array. So the answer isn't simply L_n but

rather $\max_i L_i$.

A quick analysis of the above algorithm shows that it runs in $O(n^2)$ time since we have n states and each state takes $O(n)$ time to compute. But can we do better?

Let's define some slightly different dynamic programming states:

M_k^i = the minimum possible last element of an increasing subsequence of length k using the first i elements.

At first glance this definition might seem odd, after all it defines n^2 states, which is worse than the n states we had before, but we will make some observations that will help us be more efficient.

First note that $M^i = [-\infty, a_i, \infty, \ldots, \infty]$, where we use ∞ to denote that there is no increasing subsequence of that length using, and $-\infty$ to denote that there is a trivial increasing subsequence of length 0. Now let's see how to compute M^{i+1} from M^i.

Observation 1. M^{i+1} and M^i differ in at most one position
Let $j^\star$ be the largest index such that $M_{j^\star}^i < a_{i+1}$. Then we have the following cases:

- For all $j < j^\star$, we have $M_j^{i+1} = M_j^i$. This is because we can just take the same increasing subsequence of length j that we had before.
- For all $j > j^\star + 1$, we have $M_j^{i+1} = M_j^i$. This is because we cannot extend any increasing subsequence of length j using a_{i+1} since it is too small.
- For $j = j^\star + 1$, we have $M_j^{i+1} = \min(M_j^i, a_{i+1})$. This is because we can either take the same increasing subsequence of length j that we had before, or we can extend the increasing subsequence of length $j-1$ ending in M_{j-1}^i by adding a_{i+1} to it.

This observation gives us a way to solve the problem in $O(n^2)$ time. Keep an array M of size n initialized to $[-\infty, \infty, \ldots, \infty]$. For each i from 1 to n, update M by finding the largest j such that $M_j < a_i$ and then setting $M_{j+1} = \min(M_{j+1}, a_i)$. Basically, the value of M in the ith iteration is exactly M^i. So the answer to the problem is the largest k such that $M_k < \infty$ at the end of the algorithm. Since each iteration takes $O(n)$, and we have n iterations, the overall running time is $O(n^2)$. Let's make one more observation to speed this up.

Observation 2. M^i is a monotonically increasing sequence, i.e., $M_1^i < M_2^i < M_3^i < \ldots$
This is because if we have an increasing subsequence of length k ending in M_k^i, then we can always remove the last element to get an increasing subsequence of length $k-1$ ending in some element less than M_k^i. Since M_{k-1}^i is the minimum possible last element of an increasing subsequence of length $k-1$, it must be less than M_k^i.

Since M^i is monotonically increasing, we can find the largest j such that $M_j^i < a_{i+1}$ using binary search in $O(\log n)$ time. This means that we can update the above algorithm to perform each iteration in $O(\log n)$ time. Thus, we have n iterations, each taking $O(\log n)$ time, leading to an overall running time of $O(n \log n)$.

Here is some pseudocode for the algorithm we just described:

1. Initialize array $M[1..n]$ where:

- $M[0] = -\infty$ (dummy value for length 0)
- $M[1..n] = +\infty$ (no subsequences of these lengths yet)

2. For $i = 1$ to n:

- Find largest j such that $M[j] < a[i]$ using binary search
- Set $M[j+1] = \min(M[j+1], a[i])$

3. Return largest k such that $M[k] < +\infty$

Practice

Problem 1

You're given a sequence of integers $a_1, \ldots, a_n$. A subsequence is almost-increasing if it contains at most one index t where $a_t \geq a_{t+1}$ (i.e., at most one descent). Design an $O(n^2)$ algorithm to compute the maximum length of an almost-increasing subsequence of a.

Problem 2

You are given two integers n and k, give an algorithm to compute n^k in $O(\log k)$ time.

Note, for the sake of this problem you can assume that multiplying two integers takes $O(1)$ time even if their magnitude depends on n and k. If this feels uncomfortable, you can assume that you are computing $n^k \mod m$ for some constant integer m.

Challenge

Problem 1

n shipping containers, with weights $w_1, \ldots, w_n$, need to be transported across the Hudson River from New Jersey to New York. There is a ferry that can transport the containers, but has a maximum weight capacity of W. You want to find the minimum number of round trips the ferry needs to make in order to transport all the containers across the river.

(a) Show that the following greedy algorithm does not always find the optimal solution:

- While there are still containers to transport:
  - Load the ferry with as much weight as possible without exceeding the weight limit W. Break ties arbitrarily.
  - Make a round trip across the river and back.

(b) Design an algorithm to find the minimum number of round trips in $O(n2^n)$ time.
