CSCI 270 Problem Set 3 Solutions

Fall 2025

Problem 1. *Imagine that you are going on a road trip. You start from position 1, and will end up at position n. At each integer position along the road (so positions $\{1, 2, \ldots, n\}$), there is a gas station. For each gas station i, you are given a price per gallon $p_i \geq 0$. One gallon of gas is exactly enough to make it one position on the line. Your car has a gas tank size of $s \leq n$ gallons, which starts out empty with you at the gas station at position 1.*

You can stop at as many gas stations as you want. Every time you stop, you can decide how many gallons to put in your tank, though the total in your tank can never exceed s. The amount of gas you buy at any station will always be an integer. Your goal is to compute the minimum amount of money you can spend to get to location n. (It's ok if your tank runs empty right as you reach it.)

Give and analyze (prove correct and analyze running time) a polynomial-time algorithm for solving this problem.

Solution. Let $\mathsf{OPT}(i, j)$ denote the cost, under an optimal strategy, of driving to position i, possibly purchasing gas at position i, and concluding with exactly j gallons of gas in the car's tank. Then $\mathsf{OPT}(0, j) = p_0 \cdot j$ for all $j \in [s]$, as the only option at position 0 is to purchase all j gallons of gas at price $p_0 \cdot j$. Furthermore, for arbitrary (i, j) with $i \geq 1$, we have the recurrence

$$\mathsf{OPT}(i, j) = \min\left\{\mathsf{OPT}(i-1, j+1), \mathsf{OPT}(i, j-1) + p_i\right\}.$$

This is due to the fact that any optimal strategy for reaching position i with j gallons of fuel either 1) does not purchase any fuel at position i, and must have reached position $i-1$ with exactly $j+1$ gallons of fuel, or 2) purchased at least one gallon of fuel at position i, and thus reached position i with $j-1$ gallons of fuel (possibly after already purchasing some gallons at position i) and purchased an additional gallon of gas in order to reach j total gallons.

In either case, respectively, the optimal strategy will also attain optimality for the subproblem $\mathsf{OPT}(i-1, j+1)$ or $\mathsf{OPT}(i, j-1)$, otherwise its cost could be improved by altering its behavior on these subproblems.[1] This establishes correctness of the following code, which populates the array A with $A[i, j] = \mathsf{OPT}(i, j)$.

```
def solution(P, n, s):
    A = [[None] * (s + 1) for _ in range(n + 1)]
    A[0] = [P[0] * j for j in range(0, s + 1)]

    for i in range(1, n + 1):
```

[1] Informally, due to the memorylessness property of the problem.

```
6           best = float('inf')
7           for j in range(0, s + 1):
8               if j + 1 <= s:
9                   best = min(best, A[i - 1][j + 1])
10              if j - 1 >= 0:
11                  best = min(best, A[i][j-1] + P[i])
12              A[i][j] = best
13      return A[n][0]
```

The running time of the algorithm is $O(n \cdot s) = O(n^2)$, as each entry of A is populated in constant time on lines 8 – 12. □

Problem 2. *Recall that we have discussed the maximum non-overlapping interval problem in the lecture. Specifically, you are given a list of n tasks. For each task i, there is a start time s_i, a finish time f_i, and a weight w_i. Your goal is to find a set of tasks S such that*

- *There is no overlap in S, i.e., for every two tasks $i, j \in S$, either $s_i > f_j$ or $f_i < s_j$.*
- *The total weight $\sum_{i \in S} w_i$ is maximized.*

Please provide an algorithm with running time polynomial in n. You only need to provide the pseudo-code and the running time analysis. The correctness proof can be skipped.

Solution. We use the following approach.

```
1   def soln(s, f, w):
2       times = sorted(set(s + f))
3       T = len(times)
4       to_idx = {time:index + 1 for (index, time) in enumerate(times)}
5
6       ends_at = [[] for _ in range(T + 1)]
7       for i in range(len(s)):
8           ends_at[to_idx[f[i]]].append(i)
9
10      A = [0] * (T + 1)
11      for t in range(1, T + 1):
12          A[t] = A[t - 1]
13          for i in ends_at[t]:
14              t_start = to_idx[s[i]]
15              A[t] = max(A[t], w[i] + A[t_start - 1])
16      return A[T]
```

The running time of the algorithm is $O(n \log n)$, dominated by the time to sort all starting and ending times on line 2. In particular, lines 3 – 10 require $O(n)$ time, as they iterate

over the arrays T and s, which have length $\leq 2n$ and n, respectively. Lines $11-15$ also run in linear time, as the total work performed by the innermost `for` loop over all t is $\sum_{t\in[T]} |\text{ends\_at[t]}| = n$, and the remaining work performed by the outer `for` loop is linear in T (with $T \leq 2n$). This completes the analysis. □

Problem 3. *Given an integer array $A = \{a_1, \ldots, a_n\}$ of length n, a subsequence $\{i_1, \ldots, i_t\}$ is called a strictly increasing subsequence if*

- $i_1 < i_2 < \cdots < i_t$, *and*
- $a_{i_1} < a_{i_2} < \cdots < a_{i_t}$.

For example, if $A = \{3, 1, 2, 4, 6, 5\}$, then both $\{3, 4, 6\}$ and $\{1, 2, 4, 5\}$ are strictly increasing subsequences. Please design an algorithm to determine the length of the longest strictly increasing subsequence. Please provide the pseudocode, running time, and correctness proof.

Solution 1. We use the following code.

```
1  def soln(A):
2      n = len(A)
3      opt = [1] * n
4      for i in range(1, n):
5          for j in range(i):
6              if A[j] < A[i]:
7                  opt[i] = max(opt[i], opt[j] + 1)
8      return max(opt)
```

The algorithm's runtime is $O(n^2)$, as it has two nested `for` loops and performs constant work on line 7. To demonstrate correctness, it suffices to prove the following claim. We refer to a strictly increasing subsequence as an SIS.

> **Lemma 0.1.** *For each array A of length n and $i \in [n]$, after the innermost `for` loop has completed running for the ith time, `opt[i]` equals the length of the longest SIS whose final index is i.*
>
> *Proof.* By induction on i. When $i = 1$ the claim clearly holds, as $\{1\}$ is the longest SIS in $\{a_1\}$. Supposing the claim holds up to $i-1$, then the innermost `for` loop on lines $5-7$ sets `opt[i]` equal to the maximum of 1 and `opt[j] + 1`, where $j < i$ is the index maximizing `opt[j]` subject to the constraint that $A[j] < A[i]$. Any SIS ending at i is either $\{i\}$ or decomposes as $p + \{i\}$, where p is an SIS ending at an index $j < i$ with $A[j] < A[i]$. This completes the inductive step. □

As any SIS in A concludes at some index $i \leq n$, `max(opt)` thus equals the length of the longest SIS in A. □

Solution 2. We use the following pseudocode.

```
1  def soln(A):
2      tails = []
3      for x in A:
4          i = bisect_left(tails, x)   # first index i with tails[i] >= x
5          if i == len(tails):
6              tails.append(x)
7          else:
8              tails[i] = x
9      return len(tails)
```

The algorithm's runtime is $O(n \log n)$, as it iterates over each element in A and runs in $O(\log n)$ time for each element of A. In particular, the runtime of lines $4-8$ is $O(\log n)$ on line 4 to binary search the array `tails`, and constant work on lines $5-8$ to either append to `tails` or update one of its entries.

For correctness, it suffices to prove the following claim. We again refer to a strictly increasing subsequence as an SIS.

> **Lemma 0.2.** *Upon processing the prefix $A[:t]$ of A, `tails` is a strictly increasing array with `tails[i]` equal to the minimal possible tail value of an SIS of length $i+1$ in $A[:t]$.*
>
> *Proof.* Exercise :) □

□