CSCI 270 Problem Set 5 Solutions

Fall 2025

Problem 1. *Now we consider a variant of the Interval Scheduling problem. Instead of maximizing the number of intervals, we now want to find a set of non-overlapping intervals with the maximum total interval length.*

Consider a greedy algorithm that first selects the longest interval. *Please determine whether it is optimal. If it is, prove it. If not, provide a counterexample.*

If you do not believe the above greedy algorithm is correct, provide a dynamic programming algorithm with pseudocode, running time analysis, and a correctness proof.

Solution. The greedy algorithm is not correct. Consider the following input: $[0, 10]$, $[9, 21]$, $[20, 30]$. The greedy algorithm will select the interval $[9, 21]$, after which it cannot select any more intervals. This yields a total interval lenght of 12. The optimal strategy is to select $[0, 10]$ and $[20, 30]$, yielding a total interval length of 20.

We can instead use the following dynamic programming algorithm.

```
1  def soln(intervals):
2      n = len(intervals)
3      intervals = sorted(intervals, key=lambda x: x[1]) # sort by finish time
4      starts = [s for s, _ in intervals]
5      finishes = [f for _, f in intervals]
6      lengths = [f - s for s, f in intervals]
7
8      p = []
9      for i in range(n):
10         # binary search for latest interval ending before ith start time
11         j = bisect_right(finishes, starts[i]) - 1
12         p.append(j)
13
14     dp = [0] * (n + 1)
15     for i in range(1, n + 1):
16         dp[i] = max(dp[i - 1], lengths[i - 1] + dp[p[i - 1] + 1])
17
18     return dp[n]
```

The algorithm runs in time $O(n \log n)$, dominated by the time to sort all n intervals by finish time (line 3), as well as the time to binary search each task for the last task which ends before

it (n many binary searches each taking time $\log(n)$; lines $10-11$). The remaining operations run in time $O(n)$, i.e., creating `starts`, `finishes`, `lengths`, and populating each entry of `dp` in constant time.

For correctness, we will demonstrate by induction on i that `dp[i]` equals the greatest total interval length that can be achieved by selecting from among the first i many intervals, subject to the non-overlapping requirement (after the intervals have been sorted by finish time). The claim clearly holds for $i = 0$ at initialization of `dp` on line 14, as 0 interval length can be attained when selecting from 0 intervals. Suppose now that the entries $0, \ldots, i-1$ of `dp` have been correctly populated by the `for` loop on line 15. Then line 16 sets `dp[i]` equal to the maximum of two terms: 1) `dp[i - 1]`, the maximum interval length that can be achieved when the ith interval is *not* used, by our inductive hypothesis, and 2) `lengths[i-1] + dp[p[i-1] + 1]`, the maximum interval length that can be achieved when the ith interval *is* used, which precludes all intervals after the first `p[i - 1] + 1` many.[1] As any scheduling of the first i intervals either includes the $(i-1)$th interval or does not, this equals the maximum total interval length that can be achieved. □

Problem 2. *You are given n items, each with a weight w_i and a value v_i. You are also given a knapsack with a maximum capacity W. The goal is to determine the maximum total value that can be obtained by selecting some of these items, subject to the constraint that the total weight does not exceed W.*

You may take a fraction of each item and obtain a proportional value. Note that this is different from Problem 3 in Homework 4.

For example, if there are four items with weights and values $(2,3), (3,2), (4,4), (5,6)$ and the knapsack capacity is $W = 9$, then you can take the items $(2,3)$, $(4,4)$ and 80% of $(5,6)$, achieving a total value of $3 + 4 + 4.8 = 11.8$.

Please provide an algorithm with pseudocode, running time analysis, and a correctness proof.

Soln. Define the "density" of an item to be its value divided by its weight. We use the algorithm which greedily takes as much weight as possible from the densest items first, as formalized by the following pseudocode.

```
1  def soln(items, W):
2      items = sorted(items, key=lambda x: x[1]/x[0], reverse=True) # sort by density
3      cur_weight = 0
4      cur_value = 0
5      for item in items:
6          weight, value = item
7          density = value / weight
8          if cur_weight + weight >= W:
9              cur_value += (W - cur_weight) * density
```

[1] Note that `dp` indexes into the *quantity* of intervals are allowed, meaning `dp[i]` allows for selection of intervals from `intervals[:i]`.

```
            return cur_value
        else:
            cur_value += value
            cur_weight += weight
    return cur_value
```

The algorithm runs in time $O(n \log n)$, dominated by the time to sort all n items by density. In particular, the `for` loop on line 5 iterates over n items and performs constant work for each, requiring a total of $O(n)$ time.

For correctness, let S be the allocation of items to the knapsack produced by the above algorithm, and let $T \neq S$ be any other allocation (subject to the weight W restriction). We will demonstrate that the value attained by T is at most that of S. We may assume without loss of generality that S and T both select items with a total weight of exactly W. First note that we may represent S as $(s_1, \ldots, s_n)$ and T as $(t_1, \ldots, t_n)$, where $s_i, t_i \leq w_i$ denote the weight taken from the ith-densest item, under S and T respectively. Let i be the first index for which $t_i < s_i$, and let j be the first index for which $t_j > s_j$. The existence of these indices is guaranteed from the fact that $S \neq T$ and S, T both have total weight W. As S selects as much weight as possible from the densest items, it must be that $i < j$. Then let T' be the allocation derived from T which reduces the weight t_j as much as possible and increases the weight t_i by the same amount, subject to $t_j \geq 0$ and $t_i \leq w_i$. Let δ be the amount of weight transferred. Then, as the density d_i of the ith item is at least that of the jth item, d_j, we have

$$\begin{aligned} \text{val}(T') &= \text{val}(T) + \delta \cdot d_i - \delta \cdot d_j \\ &= \text{val}(T) + \delta \cdot (d_i - d_j) \\ &\geq \text{val}(T). \end{aligned}$$

Repeating this exchange procedure at most $2n$ times transforms T into S, as index i or j must increase in value with each exchange. As the total value of the allocation weakly increases with each exchange, we have that $\text{value}(T) \leq \text{value}(S)$, completing the argument. □

Problem 3. *In this problem, we will look at how to flip the outcome of an election. In many elections, voters can only vote for one candidate. However, this leads to a lot of speculating to avoid "wasting" one's vote on a candidate who won't win anyway. For that reason, some elections ask voters to rank all candidates from most favorite to least favorite; with this extra information, one can then do a better job picking a "good" candidate.*

Assume that there are $m \geq 2$ candidates and n voters. Each voter ranks all *m candidates from most to least favorite. For voter $v = 1, \ldots, n$, their order is a permutation π_v, and we write $\pi_v(i)$ for the candidate that voter v has in position i. Voter v's most favorite candidate is $\pi_v(1)$, and their least favorite is $\pi_v(m)$. For each vote, candidate c gets $m-1$ points for each first-place vote they get, $m-2$ for each second-place vote they get, and so on; with 0 points for each last-place vote they get. The points are added up, and the candidate with the largest total number of points wins.*

Give a polynomial-time algorithm to compute a ballot you can submit to make A win, or correctly conclude that no such ballot exists. Prove the correctness of your algorithm, and analyze its running time.

Solution. We design a ballot in which A is the top candidate and the remaining candidates are ranked in reverse order by their total number of points. We return this ballot if it leads to a victory for A, and otherwise conclude that no ballot exists for which A can win. This is formalized by the following pseudocode.

```
def soln(votes, A):
    m = len(votes[0])
    candidates = votes[0][:]

    score = {c: 0 for c in candidates}
    for ballot in votes:
        for pos, c in enumerate(ballot):
            score[c] += (m - 1 - pos)

    others = [c for c in candidates if c != A]
    others.sort(key=lambda x: score[x])

    new_ballot = [A] + others  # weakest rival first after A
    new_score = score.copy()
    for pos, c in enumerate(new_ballot):
        new_score[c] += (m - 1 - pos)

    if all(new_score[A] > new_score[c] for c in candidates if c != A):
        return new_ballot
    return None
```

The algorithm's runtime is $O(mn + m \log m)$, as calculating candidates' scores from all ballots requires time $O(mn)$ and sorting candidates by their scores requires time $O(m \log m)$.

For correctness, let b be an arbitrary ballot, and let `new_ballot` be the ballot computed by our algorithm. Recall that `new_ballot` ranks A as the first candidate, followed by all other candidates in increasing order of points (i.e., lower-point competitors ranked higher). We will demonstrate that if A will win the election with b as the last ballot, then it will win the election with `new_ballot` as the last ballot. We may assume that $b \neq$ `new_ballot`, otherwise the claim is immediate. We may also assume that b ranks A as the first candidate, otherwise A can be swapped with the first candidate in b, which increases A's points while reducing one of its competitor's points. Then in order for it to be the case that $b \neq$ `new_ballot`, it must be that there exists a pair of candidates (B, C) such that B has more points than C (before b is accounted for) and B is ranked exactly one place higher than C in b. Let b' be the ballot obtained from b by swapping the rankings of B and C. Then if b leads to a victory for A, b' also leads to a victory for A, as it transfers exactly one vote from candidate B to C.

Proceeding in this way, adjacent pairs misordered elements in b can be resolved while preserving A's victory. After a finite number of steps (less than m^2), this exchange argument transforms b into `new_ballot`, demonstrating that if there exists any ballot for which A wins the election, `new_ballot` is one such ballot. This completes the argument. □