Math 182: Problem Set 2
Kenny Guo

Question 1.

Suppose that you are in some country where there are n types of coins, which have values $v_1, v_2, \ldots, v_n$ and that you have access to an unlimited supply of each type of coin. One day, you decide to buy some item with cost v and you want to do so using as few coins as possible (assume that there is some choice of coins whose values add up to v). However, it takes you a very long time to figure out the minimum number of coins required, so you try to come up with an efficient algorithm to solve the problem.

You come up with the following algorithm: repeatedly find the coin with the highest value less than or equal to the currently remaining amount you need to pay and subtract its value from the amount left to pay until you hit 0. Pseudocode for this algorithm is given below:

```
ChooseCoins(A,v):
    result = new array of same size as A
    while v is not 0:
        i = index which maximizes A[i], subject to A[i] less than or equal to v
        v = v - A[i]
        result[i] = result[i] + 1
    return result
```

Array A contains the values of all of the coins; `result`$[i]$ is the number of the i-th coin used. Note that we are assuming that a solution is possible, i.e. that there is some way of obtaining exactly v by adding up values from A.

Is the algorithm described above correct? Prove that it is, or give an input on which it is not correct.

Example: Suppose the coins have values $1, 2, 7$, and 20. If the cost of the item is 31, then the smallest number of coins is 4: $20 + 7 + 2 + 2$.

It is not correct. Consider the counterexample: $v = 10$ with coins having values $8, 5, 1$. The greedy algorithm chooses $8, 1, 1$ which is 3 coins, however, the optimal solution is $5, 5$, which is only 2 coins.

Question 2.

Design an efficient algorithm to solve the following problem. Given two sequences of numbers $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_n$ (with duplicates allowed), determine whether $a_1, \ldots, a_m$ occurs as a subsequence of $b_1, \ldots, b_n$.

Recall that a subsequence of a sequence $b_1, b_2, \ldots, b_n$ is a sequence of the form $b_{i_1}, b_{i_2}, \ldots, b_{i_j}$, where

$$1 \leq i_1 < i_2 < \cdots < i_j \leq n.$$

You may assume that the two sequences are given as arrays A and B.

For full credit, your algorithm should run in $O(n + m)$ time.

Example: If $A = [4, 5, 2]$ and $B = [4, 3, 2, 4, 5, 3, 2]$, then the answer is yes: we can take the subsequence of B consisting of the fourth, fifth, and seventh elements. If $A = [4, 5, 2]$ and $B = [4, 3, 2, 4, 5, 3, 4]$, then the answer is no.

The idea of the algorithm is as follows, keep a pointer (i) for array A and a pointer (j) for array B, initialized at their beginnings. For each iteration, if they match, increment i and j by 1, and if not, just j. If i reaches end, return true, else, return false.

```
bool Subsequence(A, B):
    m = len(A)
    n = len(B)

    if m = 0:
        return true
    if m > n:
        return false

    i = 1
    for j = 1, 2, ..., n:
        if A[i] == B[j]:
            i++
        if i > m:
            return true
    return false
```

The runtime of this algorithm is $O(m + n)$.

Proof. The initializations and edge case checks are constant time. Each iteration of the for loop performs a comparison ($O(1)$ time), and possibly an increment of i ($O(1)$ time). There are n iterations and i is incremented at most m times before the algorithm terminates. Thus, runtime is $O(m + n)$. □

This algorithm is correct.

Proof. The edge cases of an empty sequence and a larger sequence are handled by the algorithm already. We show the loop invariant that at the start of iteration j, $A[1: i - 1]$ is a subsequence of $B[1: j - 1]$. For the base case, $j = 1$, this is trivially satisfied since the heads in question are empty. So now suppose this is true for some j. If $A[i] = A[j]$, then the algorithm increments i by 1, and we have that $A[1: i - 1]$ is a subsequence of $A[1: (j + 1) - 1]$. If they don't match, we still have that $A[1: i - 1]$ is a subsequence of $A[1: (j + 1) - 1]$ since i is not incremented. Finally, the algorithm terminates since for-loops always terminate.

If the loop terminates on some j where $i = m + 1 > m$ and returns true, by the loop invariant, we must have that $A[1: m]$ is a subsequence of $B[1: j - 1]$, and thus a subsequence of B itself. If loop finishes after $j = n$ and returns false, this means $i \leq m$. By the loop invariant, this means only at most $A[1: m - 1]$ is a subsequence of $B[1: n] = B$, so A is not a subsequence of B. Thus, the algorithm returns true iff A is a subsequence of B. □

Question 3.

Design an efficient algorithm to solve the following problem. Given some collection of intervals $[s_1, t_1], [s_2, t_2], \ldots, [s_n, t_n]$, find the smallest number of intervals whose union is equal to the union of the entire collection. (This is known as an optimal covering.) You may assume that the left endpoints of the intervals are all distinct (i.e. that for all $i \neq j$, $s_i \neq s_j$). For full credit, your algorithm should run in $O(n \log n)$ time.

Example: If the intervals are

$$[3, 6],\ [5, 10],\ [1, 4],\ [6, 10],\ [7, 12],\ [10, 12]$$

then the smallest number of intervals is four: the union of

$$[1, 4],\ [3, 6],\ [6, 10],\ [10, 12]$$

will cover every other interval.

Idea: Sort the intervals by their starts. Add the first interval, since it has (unique) earliest start. Then, employ greedy across all intervals whose starts are contained with current component, and adding the one with the furthest right endpoint (or nothing if all are subsumed). Continue by adding next interval until I is iterated through.

```
OptimalCover(I):
    n = len(I)
    Sort intervals [s_i, t_i] in I by start points s_i
    C = empty list
    i = 1

    while i <= n:
        Add I[i] to C
        covered = t_i
        i = i + 1

        while i <= n and s_i <= covered:
            best_end = covered
            best_interval = empty

            while i <= n and s_i <= covered:
                if t_i > best_end:
                    best_end = t_i
                    best_interval = I[i]
                i = i + 1

            if best_interval == empty:
                break
```

```
            Add best_interval to C
            covered = best_end

    return C
```

This algorithm runs in $O(n \log n)$ time.

Proof. Sorting takes $O(n \log n)$. After sorting, the algorithm scans across the interval list using index i, which only increases, either when an interval is added to start a new component or when an interval is checked to see if it is "best" in greedy selection. Both of these processes take constant work, so the total time spent across all the while loops is only $O(n)$. Thus, the total complexity is $O(n \log n)$. □

This algorithm returns an optimal cover.

Proof. Note that we can restrict our attention to connected components of the full union; i.e. if our greedy solution chooses intervals for each connected component optimally, it will have produced an optimal cover for the whole union.

Suppose the intervals are sorted by increasing start point. Consider an arbitrary connected component of the union, and assume the greedy algorithm chooses $G_1, \ldots, G_k \in I$ to cover. We proceed via exchange argument and show there is an optimal solution that also uses these intervals for this component.

For the base case, since the start points are distinct, every cover of the component must contain G_1, since it has the left-most point. Next, suppose for some $j \geq 1$ that there exists an optimal cover that contains $G_1, G_2, \ldots, G_r$. Let x be the rightmost point covered these r intervals. If the component ends at x, we are done since the optimal is the greedy.

Else, let O_{r+1} be the next interval in the optimal solution, since it must cover beyond x. It must be that the start point of O_{r+1} is less than or equal to x, otherwise there would be a gap. Thus, the greedy algorithm also considers this interval for G_{r+1}, meaning

$$G_{r+1}(t) \geq O_{r+1}(t).$$

This means we can replace O_{r+1} with G_{r+1}, and the optimal solution is still optimal, maintaining the same coverage with the same number of intervals, but with the first $r+1$ intervals the same as the greedy solution.

Thus, we can continue this induction up to k and show $G_1, G_2, \ldots, G_k$ is an optimal solution for this component. Applying to all components of the union let's us conclude. □

Question 4.

Design an efficient algorithm to solve the following problem. Suppose we are given n jobs, each from a different customer. As in the minimizing lateness problem from class, only one job can be completed at a time, and they cannot be split into pieces or abandoned once started. Each job i will take t_i time to complete. Given a schedule, let c_i denote the finishing time of job i. That is, if the first job is job i, $c_i = t_i$. If job j is done immediately afterwards, c_j will be $c_i + t_j$. Each job is given a weight w_i based on the importance of the customer.

The goal is to build a schedule that will minimize

$$\sum_{i=1}^{n} w_i c_i.$$

For full credit, your algorithm should run in $O(n \log n)$ time.

Example: Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would give a weighted completion time of

$$10 \cdot 1 + 2 \cdot (1 + 3) = 18,$$

while doing the second job first would give

$$2 \cdot 3 + 10 \cdot (3 + 1) = 46.$$

Idea: Intuitively, larger weights should be completed first, and smaller durations should be completed first. Thus, we sort the jobs by $\frac{t_i}{w_i}$ in increasing order, and then assign the jobs in order to the schedule (with no gaps).

```
Schedule(T[], W[]):
    n = len(T)
    S = new length n array
    A = [1, 2, ..., n]
    Sort A by T/W in increasing order
    cur = 0
    for i = 1, 2, ..., n:
        S[A[i]] = cur
        cur = cur + T[A[i]]
    return S
```

This algorithm runs in $O(n \log n)$ time.

Proof. Computing for each of the n jobs takes constant time, and sorting takes linear-log time, for a total complexity of $O(n \log n)$. Finally, the algorithm performs a linear scan and appends to the schedule, which is $O(n)$ time. Together, $O(n \log n)$. □

This algorithm creates a schedule that minimizes the weighted sum of finishing times.

Proof. We proceed via exchange argument. Let O be an optimal solution, and assume it has no idle time, without loss. Define an inversion to be a pair of jobs i, j where $s_i < s_j$ but $t_i/w_i > t_j/w_j$. We claim that a schedule with an inversion can be turned into a schedule with fewer inversions without increasing the weighted sum $\sum_i w_i c_i$.

A schedule with an inversion has a side-by-side inversion, say with indices $k, k+1$, so $t_k/w_k > t_{k+1}/w_{k+1}$. Suppose all jobs before finish at time T. Thus, these two jobs in this order contribute to the weighted sum:

$$w_k(T + t_k) + w_{k+1}(T + t_k + t_{k+1}).$$

We can swap these two, removing the inversion, and get the contribution to the weighted sum:

$$w_{k+1}(T + t_{k+1}) + w_k(T + t_{k+1} + t_k).$$

Subtracting the second from the first, we see:

$$w_k(T + t_k) + w_{k+1}(T + t_k + t_{k+1}) - w_{k+1}(T + t_{k+1}) - w_k(T + t_{k+1} + t_k) = w_{k+1}t_k - w_k t_{k+1} \geq 0$$

where the inequality follows from the fact that $k, k+1$ was an inversion. Thus, once we remove the inversion, we get a schedule with weakly lower weighted sum.

Now, suppose O has inversions. We can iterate this argument and remove all the inversions from O, and since it's optimal, its weighted sum after these removals should still be the same. Finally, we claim that schedules with no inversions and no idle time all have the same weighted sum. By no inversions, schedules can only differ in the order in which consecutive jobs with the same ratio t/w are scheduled. Consider two jobs, i, j, such that $t_i/w_i = t_j/w_j$, or equivalently, $w_j t_i = w_i t_j$. Then, if i comes first, the contribution to the weighted sum is

$$w_i(T + t_i) + w_j(T + t_i + t_j),$$

and if j comes first, its contribution is

$$w_j(T + t_j) + w_i(T + t_j + t_i).$$

Again, subtracting the two yields $w_j t_i - w_i t_j = 0$, so they both contribute the same amount, and order doesn't matter.

To conclude, note that both the optimal solution post-inversion removal and the greedy solution both have no inversions and no idle time. Thus, the greedy solution must also be optimal. □