Homework 3
Sara Pesavento
spesaven@usc.edu

1. **You have N ropes each with lengths L1, L2, . . . , LN, and we want to connect the ropes into one rope. Each time, we can connect 2 ropes, and the cost is the sum of the lengths of the 2 ropes. Develop an algorithm such that we minimize the cost of connecting all the ropes. (10 points)**

Notes:
Imagine the following ropes of increasing lengths (Li < Lj here when i < j):

___ _____ _______ _____________ ____________________
L1 L2 L3 L4 L5

If we were to start merging them in decreasing order of length:
L4 + L5
L3 + (L4 + L5)
L2 + (L3 + L4 + L5)
L1 + (L2 + L3 + L4 + L5)
As we can see, L1, the shortest length, only occurs one time, while the longest length L5 occurs four times. We can fix this by instead merging them in increasing order of length.
L1 + L2
L3 + (L1 + L2)
L4 + (L1 + L2 + L3)
L5 + (L1 + L2 + L3 + L4)

Algorithm:
Sort the N ropes in non-decreasing order of lengths

```
connected_ropes = 1st rope length
loop i from 2nd to N ropes:
        x = merge length of existing and ith rope length
        connected_ropes += x

return connected_ropes
```

Proof:
Let's say we have an optimal solution S where the cost is minimized. We can prove that our algorithm always stays ahead of this solution. First of all, our solution has no inversions.
Step 1: show that by removing any inversions in the optimal solution, the solution stays optimal.
An inversion would occur if Lj > Li. and i > j. For example, L3 > L2 but L2 comes after L3:
... _________ ____
L3 L2

If the previous sum is Lp. Then the merge with the inversion would be (Lp + L3) + L2 + (Lp + L3).
The merge without the inversion would be (Lp + L2) + L3 + (Lp + L2), which is less, since L3 > L2.
Step 2: show that if lengths are equal, their ordering doesn't matter.
… _____ _____
L2 L3
If the previous sum is Lp. Then the merge would be (Lp + L2) + L3 + (Lp + L2). If they are

reversed, the merge would be (Lp + L3) + L2 + (Lp + L3), which is the same length because L2 == L3.

We have transformed an optimal solution into a solution that looks like ours (with no inversions), and thus our solution is optimal.

2. **You have a bottle that can hold L liters of liquid. There are N different types of liquid with amount L1, L2, ..., LN and with value V1, V2, ..., VN. Assume that mixing liquids doesn't change their values. Find an algorithm to store the most value of liquid in your bottle. (10 points)**

Notes:
So we have N types of liquid, and each has a certain value. We want to store the most value of liquid. Say we have ordered the liquids in non-increasing order such that Vn >= Vn+1.
In that case we will want to start at the beginning with the liquid with the highest value and use all of it. If this fills up all L liters, we are done. Otherwise, we choose the next highest value of liquid and use all of it (or up to the 1 liter mark).

Algorithm:
Sort the N types of liquid in non-increasing order by value.

store pointer i starting at front of the list = 0
store current bottle value = 0
store amount of bottle filled = 0

while we have not filled the L liter bottle:
    record the amount we still need to fill as full mark - amount of bottle filled
    fill the bottle with the min(amount we need to fill, all of our current liquid Li of Vi)

    record the new bottle value and amount of bottle filled
    increment i to next largest bottle

return value of bottle achieved

Proof:
Let's say we have an optimal solution S where the value of the bottle is maximized. We can prove that our algorithm always stays ahead of this solution. This is very similar to the previous proof.
Step 1: show that by removing any inversions in the optimal solution, the solution stays optimal.
    An inversion would occur if Vj > Vi, but the Liquid (Li, Vi) comes before (Lj, Vj). For example:
    Li Lj
    Vi Vj
    If the amount currently in the bottle is (Lp, Vp), and the total bottle capacity is Lt, then with this inversion, the steps to add (Li, Vi) then (Lj, Vj) would be
        1. Lp + min(Lt - Lp, Li) = Lp2
        2. Lp2 + min(Lt - Lp2, Lj)
    If we removed the inversion it would be:
        1. Lp + min(Lt - Lp, Lj) = Lp2
        2. Lp2 + min(Lt - Lp, Li)

Since Vj > Vi, Lp2 in the inverted method is less than in our solution. In the second step, if all the liquid can be added, then the inverted step will be the same as our solution, otherwise, our solution will have a higher value.

Step 2: We can also show in a similar manner that if Li, Vi and Lj, Vj have the same value, it does not matter which comes first. Either way we will fill the same amount of the same value so our solution will have the same value as the optimal solution.

We have transformed an optimal solution into a solution that looks like ours (with no inversions), and thus our solution is optimal.

3. **Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let d1 < d2 ... < dn be the locations of all the gas stations along the route where di is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most p miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Give the time complexity of your algorithm as a function of n. (20 points)**

Notes:

USC --------->g2-------->g2--->g3------->g4----------->g5 ------->Santa Monica

Let's first look p miles ahead of our current position at USC. We can only travel p miles before completely running out of gas. But if that range of p miles had two gas stations, we'd want to pick the one that is furthest out. Because we want to stop a minimum number of times.

Algorithm:

Consider the next p miles from the starting point, and choose the farthest possible gas station (or your destination) within that range. Update the starting point to that chosen gas station. Then once again choose the farthest possible gas station (or your destination) within p miles from this station. Continue this process until you are able to reach Santa Monica.

Time Complexity:

O(n)

Loop once through and check gas stations until you reach a station >= p miles, or the destination. Once this happens, take the last station before reaching this point.

Proof:

To show that this is the optimal solution, we want to show that our solution always stays ahead of an optimal solution.

1. For the first gas station chosen, we know our algorithm chooses the last possible gas station in the range of p miles from our starting point. Thus, if the optimal solution started at a gas station to the right of ours, this would not work since the car would have run out of gas.
2. We also need to show that our k+1th gas station is the same or to the right of the k+1th gas station in the optimal solution (aka that we stay ahead). Say we start at the kth gas station in our solution and in the optimal solution. It is not possible for the optimal solution to end up at a gas station that

is farther along, because if there was a gas station to the right that could be travelled to in p miles, our solution would have chosen it as well.

3. Let's assume that the optimal solution has < n gas station stops, and our solution has n. We have already shown that our n-1th stop is not to the left of their n-1th stop. So if we needed one more stop to reach the destination, the optimal solution would need at least one as well.

4. **(a) Consider the problem of making change for n cents using the fewest number of coins. Describe a greedy algorithm to make change consisting of quarters(25 cents), dimes(10 cents), nickels(5 cents) and pennies(1 cent). Prove that your algorithm yields an optimal solution. (Hints: consider how many pennies, nickels, dimes and dime plus nickels are taken by an optimal solution at most.) (20 points)**

Notes:

If n = 100 cents →
1 (x100)
1 (x95) + 5 (x1)
1 (x90) + 5 (x2)
…
25 (x4)

Algorithm:

A greedy algorithm might follow how we would create change in real life. We start with the largest coin amount and add as many as possible without exceeding the total amount of change n. Deduct this sum from the existing amount, and use this remainder as our new existing amount to create change for. Move onto the next smallest coin, and repeat this process until there is no more change to create.

Proof:

- Consider if we only had pennies and nickels. We will use at most 4 pennies, because anything after pennies would be covered by nickels, which will reduce the overall amount of coins.
- Consider if we had pennies, nickels, and dimes. Then we would use at most 1 nickel, since anything covered by 2 nickels would instead be covered by a dime, which will reduce the overall amount of coins.
- Consider if we had pennies, nickels, dimes, and quarters. Then we would use at most 2 dimes, since anything over 2 dimes would be replaced by a quarter and a nickel, which will reduce the overall amount of coins.

Therefore, we can see that the greedy algorithm provides the optimal solution for this set of coin denominations.

(b) For the previous problem, give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Assume that each coin's value is an integer. Your set should include a penny so that there is a solution for every value of n. (10 points)

Let us consider the coins **[1, 3, 4]**. If we tried to make change for **n = 6** cents, then our algorithm you first choose 4, because it's the largest value. Then 2 is leftover and we'd have to use 2 pennies. → This yields 3 coins. However, the optimal solution would be to use two 3's for a total of 2 coins.

5. **Solve Kleinberg and Tardos, Chapter 3, Exercise 3. (15 points)**

The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be O(m + n) for a directed graph with n nodes and m edges.

We can run the same algorithm as in the book but with a slight modification. If in each iteration we always find a node with no incoming edges, then a topological ordering (no cycles) will be produced.
First pop one node from the list of those with no incoming edges, then look at it's adjacent nodes. If in some iteration, every node has at least one incoming edge, it follows that the graph contains a cycle.

Previous:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G-{v} and append this order after v

Current:
Find a node v with no incoming edges and order it first
If no node is found that has no incoming edges, return False because a cycle is found
Delete v from G
Recursively compute a topological ordering of G-{v} and append this order after v

6. **Solve Kleinberg and Tardos, Chapter 4, Exercise 4. (20 points)**

Some of your friends have gotten into the burgeoning field of time-series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges-what's being bought- are one source of data with a natural ordering in time. Given a long sequence s of such events, your friends want an efficient way to detect certain "patterns" in them-for example, they may want to know if the four events:

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible events (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a subsequence of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence:

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events-S' of length m and S of length n, each possibly containing an event more than once-and decides in time O(m +n) whether S' is a subsequence of S.

Algo Notes:

In our sequence S, we have s1, s2…..sn, and in S' we have s1', s2'.....sm'.
Our algorithm will search for the first event/occurrence of s1 in S', then continue the search for s2..
Example:
subsequence: D E F sequence: A B C D E D E F G delete these second D and E, we already saw them earlier so we are just looking for an F at that point.

Algorithm:
#1 - pointers

```
i = pointer in S = 0  #length of sequence
j = pointer in S' = 0 #length of subsequence

while i < n:
        if j == m:
                break   #we found all the subsequence characters!
        if we found a match (si == sj'):
                j += 1
        i += 1

#check if a match was found
if j == m:
        return True

return False
```

This algorithm will run in a single loop over the sequence of length N → O(N).
inside the loop are all O(1) operations.

#2 - greedy (less optimal)

```
#base cases
if source S' is empty (we found all the words in S):
        return True
if source S is empty (we couldn't find subsequence S):
        return False
```

Start by comparing the first two strings of source S and target S'
If they match:
        remove this word from S and S' and recursively search the remaining strings
else:
        remove this first word in S' and continue searching for a letter that matches the first word in S

This solution is greedy because when we find a match we immediately cross it out and search through the rest without considering the rest of the words.

O(N) time and O(N) space

Proof:
Let us prove that our greedy algorithm always stays ahead of an optimal solution: O. Consider when we have a match w1, which is the first matching word in S and S'. Since it is the first word in the array of words, then it must be w1 <= o1 (meaning we are at or to the right of the optimal solution's first match). Now we can consider the case where the word index i > 1. Up until this point we can assume that based on the previous point, we have found a match wi-1 where wi-1 <= oi-1 (meaning we stay ahead). Our algorithm states that we will find the first matching word after wi-1, if it exists. Therefore it must be that our next match wi <= oi. If oi found a match before our solution, then based on our algorithm we would have also chosen that match.