CS 161 (Stanford, Winter 2024) Homework 7

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Exercises. The following questions are exercises. We suggest you do these on your own. As with any homework question, though, you may ask the course staff for help.

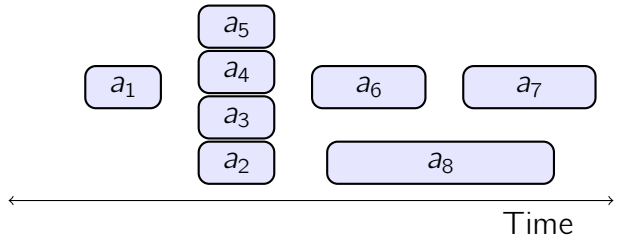
1 Greedy Class Selection (3 pt.)

You are signing up for courses next quarter. You create a schedule with n classes you want to take, each of which has a start and finish time. All classes have attendance required, so you must find a subset of the greatest possible size, subject to the constraint that none of the classes within overlap.

Below is an example schedule.

start times:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	1	3	3	3	3	5	8	5.5

finish times:	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
	2	4	4	4	4	7	12	10



Two valid solutions to this schedule are $\{a_1, a_2, a_6, a_7\}$, and $\{a_1, a_3, a_6, a_7\}$. Two invalid solutions are $\{a_1, a_2, a_6\}$ (We only include three activities when we could include four) and $\{a_1, a_2, a_6, a_8\}$ (two of the activities overlap).

Consider the following greedy algorithm for class selection. The idea is that at each step, we greedily add a valid class with the fewest conflicts with other valid classes. (A class is *valid* if it doesn't conflict with an already selected class).

The number of conflicts to begin with (represented in the array OV) are:

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
0	3	3	3	3	1	1	2

The algorithm (breaking ties arbitrarily) could choose a_1, then a_6, then a_7, then a_2.

Is this algorithm correct?

[We are expecting: Either a short English explanation for why this algorithm always succeeds, or a counterexample to show that it doesn't.**]**

Solution

This algorithm does not always succeed. Imagine a situation where we had A: (0, 2), B-D: (1, 3), E: (3, 4), F: (4, 6), G: (6, 7), H-J: (7-10), and K: (11-12). In this case, it actually makes more sense to take intermediate blocks E and G, alongside beginning and ending blocks A and K, although they each overlap with many blocks B-D and H-J. However, our algorithm would instead pick the central block F, because F only overlaps with the intermediate blocks E and G, avoiding the stacked blocks B-D and H-J. In short, an example where some optimal activities overlap with a large number of small, overlapping, focused activities will artificially remove the optimal activities from consideration, though the small, overlapping activities (irrelevant activities) should have been ignored.

2 Quagga Coffee

Quana the Quizzical Quagga has a coffee bean canister with a capacity of Q ounces. Quana heads to the campus coffee shop that has n types of coffee beans. Each coffee bean type i has a value per ounce $v_i > 0$ (measured in units of dollars per ounce) and a quantity $q_i > 0$ (measured in ounces). There are q_i ounces of coffee bean type i available to you, and for any real number $x \in [0, q_i]$, the total value that you derive from x ounces of coffee bean type i is $x \cdot v_i$.

Note that Quana can take a fractional amount of each type of bean. For example, perhaps there is 15 ounces of Arabica coffee beans. Quana can choose to put 1.61 ounces of Arabica coffee beans in their canister.

Quana wants to create the most expensive coffee bean concoction in the canister. In other words, Quana wants to choose an amount $x_i \geq 0$ of each coffee bean type i in order to maximize the total cost $\sum_i x_i v_i$ while satisfying the following:

(1) Quana doesn't overfill their canister (that is, $\sum_i x_i \leq Q$), and

(2) Quana doesn't take more of an item than is available (that is, $0 \leq x_i \leq q_i$ for all i).

Assume that $\sum_i q_i \geq Q$, so there always is some way to fill the canister.

2.0 (0 pt.)

Suppose that Quana already partially filled the canister, and there is some amount of each item left. What type of coffee bean should Quana take next, and how much?

[We are expecting: Nothing, this part is worth zero points, but it's a good thing to think about before you go on to the next part.**]**

2.1 (4 pt.)

Design a **greedy algorithm** which takes as input Q along with the tuples (i, v_i, q_i) for $i = 0, \ldots, n-1$, and outputs tuples (i, x_i) so that (1) and (2) hold and $\sum_i x_i v_i$ is as large as possible. Your algorithm should take time $O(n \log n)$.

Note: If you have a list of tuples (a_i, b_i, c_i), it is perfectly acceptable to say something like "Sort the list by c_i" in your pseudocode.

[We are expecting: Pseudocode with a short English description of your algorithm, and a justification of the running time.**]**

Solution

```
def expensiveBeans(Q, beans):
   // sort by value per ounce
   sorted = sort_by_v_desc(beans)
   bag = []
   bag_weight = 0

   for j in range(len(sorted)):
       amount_to_add = min(Q - bag_weight, sorted[j][q]
       bag_weight += amount_to_add
       bag.append((i, amount_to_add))
       if bag_weight == Q:
           return bag

   return bag
```

This question is essentially the Fractional Knapsack problem, and can be solved identically. We are trying to maximize the price of beans that have limited availability by placing them into a limited space. We first sort the beans, which is $O(nlogn)$. Then, we loop through the beans, which is $O(n)$, doing a sequence of $O(1)$ operations, causing our total runtime to be dominated by the sort and therefore to be $O(nlogn)$.

The pseudocode takes the sorted beans and simply attempts to put as much of the most expensive beans in as possible by finding the minimum of the remaining amount

of the most expensive beans and the remaining amount of space (to avoid negative issues with both limited quantities) and continuing to do this with every most expensive bean until it is exhausted or the bag is full.

2.2 (3 pt.)

Fill in the inductive step below to prove that your algorithm is correct.

- **Inductive hypothesis:** After making the t'th greedy choice, there is an optimal solution that extends the solution that the algorithm has constructed so far.
- **Base case:** Any optimal solution extends the empty solution, so the inductive hypothesis holds for $t = 0$.
- **Inductive step:** *(you fill in)*
- **Conclusion:** At the end of the algorithm, the algorithm returns a set S^* of tuples (i, x_i) so that $\sum_i x_i = Q$. Thus, there is no solution extending S^* other than S^* itself. Thus, the inductive hypothesis implies that S^* is optimal.

[We are expecting: A proof of the inductive step: assuming the inductive hypothesis holds for t, prove that it holds for $t + 1$.**]**

Solution

We have a coffee cup of t choices of coffee bean selected by our algorithm in 2.1, and we want to show that by following our algorithm in $t + 1$, there exists an optimal solution which uses the $t + 1$.

Consider an optimal solution S. Assume that at step $t + 1$, b is the most expensive bean type available and x_b is the amount of bean b the algorithm would have selected. Assume S selects less than x_b of bean type b at the $t + 1$st step, either selecting a cheaper bean c or less than x_b of b. We can exchange some amount of bean c with an equivalent amount of bean b, maintaining the total weight. This would either increase or maintain the the value because b has higher value density than c at the same weight. If this amount of b was selected later on in S, causing the value after swap to be maintained, then we are still able to create an optimal solution after step $t + 1$, satisfying the inductive hypothesis. If this amount of b was not selected later on in S, then the total value of S will have gone up after the swap, showing that S was not optimal; this is impossible.

Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

3 Smartphone Stress Testing

Plucky is an engineer at WaddleWireless, a company that makes and sells smartphones. They are testing the company's newest product, the SnowMobile! Plucky needs to report how many floors high the phone can be dropped from without breaking. If the phone breaks from floor i, it will also break when dropped from any floor j as long as $j \geq i$.

Plucky has some excess test phones to run experiments with to determine the highest floor the phone can be dropped from without breaking. Once a phone breaks, it can no longer be used to run experiments. Plucky must be entirely certain of the highest floor from which the phone can withstand being dropped before running out of test phones. Plucky's job depends on it!

Count the minimum number of drops that Plucky needs to make in the worst case, given that Plucky has k test phones.

[**Note:** If dropping the phone from any of the floors above ground level would break the phone, then the only floor that the phone can be dropped from is ground level, which is indicated as floor 0. If there is one floor above ground level and the phone won't break if dropped from this high, then the phone can withstand floor 1. You won't need to worry about this too much in the following problems.]

For $n \geq 0$ and $k \geq 1$, let $D[n, k]$ be the *optimal worst-case number of drops* that Plucky needs to determine the correct floor out of n floors using k test phones. That is, $D[n, k]$ is the number of drops that the best algorithm would use in the worst-case.

3.1 (1 pt.)

For any $1 \leq j \leq k$, what is $D[0, j]$? What is $D[1, j]$?

[Note: the phone is guaranteed not to break when dropped from the floor 0 (ground level).]

[We are expecting: Your answer. No justification required.**]**

Solution

D[0, j] = 0
D[1, j] = 1

3.2 (1 pt.)

For any $1 \leq m \leq n$, what is $D[m, 1]$?

[We are expecting: Your answer, with a brief (1 sentence) justification.**]**

Solution

D[m, 1] = m. Since Plucky has to break the phone on the exact maximum floor, we can't use any algorithm other than going sequentially from floor 0 and testing drops at every floor; if he broke it on a floor with any gaps in testing between that floor and a lower floor, we wouldn't know which floor was actually the max (the floor where Plucky dropped, or one of the gap floors).

3.3 (2 pt.)

Suppose the best algorithm drops the first phone from floor $x \in \{1, \ldots, n\}$. Write a formula for the optimal worst-case number of drops remaining in terms of $D[x-1, k-1]$ and $D[n-x, k]$.

[We are expecting: Your formula and an informal explanation of why this formula is correct.**]**

Solution

The worst case drops = 1 + max(D[x-1, k-1], D[n-x, k]). This is because if the phone breaks, then we have to keep checking the lower floors to find where exactly was the minimum (somewhere between 1 and n-1) and we have 1 fewer phone (k-1). If the phone does not break, then we have to find where it will break (somewhere between n-x) and we still have k phones.

3.4 (2 pt.)

Write a formula for $D[n, k]$ in terms of values $D[m, j]$ for $j \leq k$ and $m < n$.

Hint: Use part 3.3.

[We are expecting: Your formula and an informal explanation of why this formula is correct.**]**

Solution

$D[n,k] = 1 + min_{x \in \{1,\ldots n\}} max(D[x-1, k-1], D[n-x, k])$.
This represents finding the floor x which gives the minimum value for the remaining drops after a single drop 1 from x. The remaining drops are represented by the expression borrowsed from 3.3.

3.5 Dynamic Programming Algorithm (5 pt.)

Design a dynamic programming algorithm that computes $D[n,k]$ in time $O(n^2k)$.

[We are expecting: Pseudocode AND a brief English description of how it works, as well as an informal justification of the running time. You do not need to justify that it is correct. **]**

Solution

```
def findMinDrops(n, k):
   # create DP table
   D = [n+1 by k+1]

   # initialize 0th and 1st floors
   for i in range k:
       D[0][i] = 0
       D[1][i] = 1

   # initialize case for 1 phone with all floors
   for i in range n:
       D[i, 1] = i

   # for all remaining floors
   # for all remaining phone counts
   # find the minimum of possible floors between 1 and the given floor
   # store that in DP table
   for i in range(2, n):
       for j in range(2, k):
           DP[i][j] = infinity
           for x in range(1, i):
               x_floor_val = 1 + max(D[x-1, k-1], D[n-x, k])
               if x_floor_val < D[i][j]:
                   D[i][j] = x_floor_val

   return D[n, k]
```

This works by:

- initializing our DP table
- initializing all of our base cases - for the 0th and 1st floors, we know how many drops it will take (0 and 1, respectively) regardless of phone count
- initializing our base case with 1 phone, in which case we will have to go through all the floors, as shown in 3.2
- then, for all remaining floors, we do what we specified in 3.3; for every floor we haven't filled in (n factor), for every possible phone count we haven't filled in (k factor), we find the minimum possible drop count created by dropping at any floor between 1 and the given floor (n factor) and store this minimum value in our DP table.

This has a runtime of $O(n^2k)$ because constructing takes $O(n^2)$ time, initializing base cases takes $O(k) + O(n)$ time (all phones for two floors, all floors for one phone). Finally, looping through all floors, for all phones, for all floors again, is $O(n^2k)$, which dominates all other terms.

4 Making Change

In addition to working as an engineer at WaddleWireless, Plucky has a part-time job at the Flipper Finance Bank, and he often has to make change (what a busy penguin!). If a customer deposits \$5 for a \$4.48 bill, he would have to provide 52¢ in change. Suppose Plucky has unlimited access to pennies, nickels, dimes, and quarters, he could pay that 52¢ with two quarters and two pennies (4 total coins), or five dimes and two pennies (7 total coins), or 52 pennies (52 total coins), or a number of other combinations. Plucky's manager loves keeping cons, so Plucky's goal is to use **as few total coins** as possible.

Consider the more general problem: Plucky has k coins which are worth distinct values $v_1, v_2, ...v_k$ (not necessarily pennies, nickels, dimes, and quarters), such that $0 < v_1 < v_2 < ... < v_k$. He has to make change amounting exactly to x. More precisely, he has to provide $[c_1, c_2, c_3, \ldots c_m]$ with each $c_j \in \{v_i, v_2, \ldots v_k\}$ so that $\sum_1^m c_i = x$. His goal is to minimize m, or in other words, to use as little coins as possible.

4.1 Greedy Approach

Plucky came up with the following greedy algorithm for making change.

4.1.1 (2 pt.)

If this algorithm returns a valid way to make change, does it return one that is optimal (i.e. that uses the minimum total number of coins)?

[We are expecting: Either a short English explanation for why this algorithm always succeeds, or a counterexample to show that it doesn't.**]**

Solution

If the algorithm finds a valid way to make change, it will be optimal. The algorithm can only ever use the largest possible coin, which means that if there is a valid way to make change, the amount of change is a multiple of the largest possible coin. There would never be a time where it would be more optimal (use less coins) than just using the largest coin IF change can be made with the largest coin, so this is optimal.

4.1.2 (1 pt.)

Does this algorithm always return a valid way of making change if one exists?

[We are expecting: Either a short English explanation for why this algorithm always succeeds, or a counterexample to show that it doesn't.**]**

Solution

No, it does not. If we have to make 6 cents out of coins of size 1 and 4 cents, it will try to use 4 cents twice and create an invalid solution rather than doing 4, 1, 1.

4.2 Divide And Conquer (3 pt.)

Plucky's friend Lucky the Lackadaisical Lemur has a divide-and-conquer algorithm and wants to show it to you! The pseudocode is below.

Lucky's algorithm correctly solves the problem! Before you start doing the problems on the next page, it would be a good idea to walk through the algorithm and understand what this algorithm is doing and why it works.

Argue that for $Coins = \{1, 2\}$, Lucky's algorithm has exponential running time. (That is, running time of the form $2^{\Omega(n)}$). You may use any statement that we have seen in class.

Hint: Consider the example of the Fibonacci numbers that we saw in class.

[We are expecting:

- A recurrence relation that the running time of your friend's algorithm satisfies when $Coins = \{1, 2\}$.
- An explanation for why the closed form for this expression is $2^{\Omega(n)}$. You do not need to write a formal proof.

]

Solution

Recurrence relation: $T(n) = T(n-1) + T(n-2) + O(1)$
. This is because for every level of the problem, we either use a coin of value one, in

which case we are left with a problem of size $n-1$, or a coin of value 2, in which case we have a problem of size $n-2$.

This is very similar to a Fibonacci sequence, which is $F(n) = F(n-1) + F(n-2)$. We know that the runtime of the Fibonacci sequence is $\Theta(\phi^x)$, which, since $\phi > 1$, can be written as 2^{kx}, or $2^{\Omega(n)}$.

4.3 Top-Down DP (5 pt.)

Now, turn Lucky's algorithm into a top-down dynamic programming algorithm. Your algorithm should also take time $O(x|Coins|)$.

[We are expecting:

- Pseudocode **AND** a short English description of the idea of your algorithm.
- An informal justification of the running time.

]

Solution

```
def topDown(coins, x):
    # create DP table with un-calculated values
    D = [-1] * (x+1)

    # initialize base case - 0 coins for 0 cents
    D[0] = 0

    def helper(target):
        # if we've already solved this subproblem, return its value
        if D[target] != -1:
            return D[target]

        min_coins = infinity
        for coin in coins:
            if target - coin >= 0:
                sub = helper(target -  coin)
                min_coins = min(min_coins, sub + 1)

        D[target] = min_coins
        return min_coins

    answer = helper(x)
```

```
            if answer == infinity:
                return None
            return answer
```

Explanation: This algorithm works by using a DP array to store all subproblems. We start by initializing a DP array. Then, starting from the desired value x, for all valid coins $\leq x$, we subtract them from our target value and calculate the minimum number of coins needed to create change for all of these subproblems. We add one to the output of each subproblem to account for the coin that we removed, and find the minimum number of coins among all tried coins to return our answer. If any subproblem returns infinity, change was not creatable using the given coins. This runs in $O(x|Coins|)$, since every problem needs to check all $|Coins|$ coins, and there are asymptotically the same number of subproblems n as there are values $1 \leq n \leq x$.

4.4 Bottom-Up DP (5 pt.)

Turn Lucky's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(x|Coins|)$.

[We are expecting:

- Pseudocode **AND** a short English description of the idea of your algorithm.
- An informal justification of the running time.

]

Solution

```
        def bottomUp(coins, x):
            # initialize DP array with 0s of size x+1
            D = [0] * (x+1)

            # initialize base case, 0 coins for 0 cents in change
            D[0] = 0

            for target in range(1:x):
                D[target] = inf
                for coin in coins:
                    if coin <= target:
                        D[target] = min(D[target], D[target - coin] + 1)

            if D[x] == infinity
                return None
            return D[x]
```

Explanation: This bottom up approach works by first initializing an empty array, with a base case of 0 (since 0 coins makes 0 change). Then, for every value between 1 and our target inclusive, we check every coin. This creates a runtime $O(|Coins|x)$, since we loop through all $|Coins|$ coins for all x values. If our DP table's value for target - that coin, plus the coin itself, is the minimum possible for the current $target$ value, we store it, building up our DP table until we get to our answer.