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.

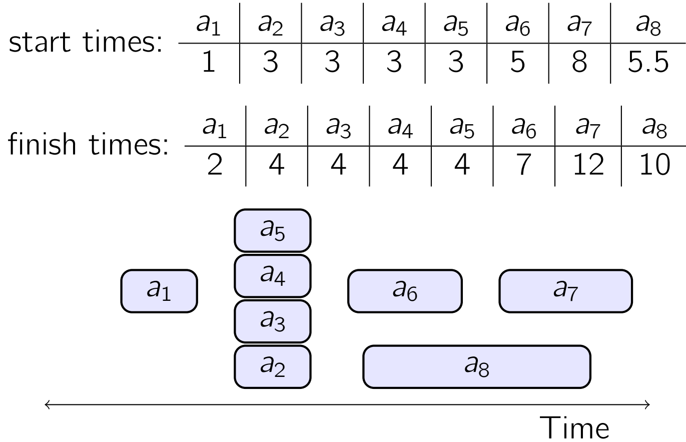
Below is an example schedule.

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).

```
def greedyClassSelection(Classes):
  result = {}
  # initialize overlap counts array OV
  OV = initOV(Classes)
  while size(OV) > 0:
      a1 = argmin(OV) #activity with the least overlaps
      result += a1
      Classes -= a1
      Classes -= {classes that conflict with a1}
      #reinitialize OV using smaller schedule
      OV = initOV(Classes)
  return result
```

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.**]**

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.**]**

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 $t + 1$.**]**

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.**]**

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.**]**

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.**]**

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.**]**

3.5 Dynamic Programming Algorithm (5 pt.)

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

[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. **]**

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 $v_1, v_2, \ldots v_k$ (not necessarily pennies, nickels, dimes, and quarters), such that $0 < v_1 < v_2 < \ldots < v_k$. 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.

```
def makeChange(Coins, x):
  if x == 0:
    return [] # empty list
  if x < 0:
    return -1
  #Coins[-1] is the coin with the largest value
  recursiveSol = makeChange(Coins, x-Coins[-1])
  if recursiveSol == -1:
    return -1
  else:
    return [Coins[-1]] + recursiveSol
```

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.**]**

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.**]**

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.

```
def makeChange(Coins, x):
    if x == 0:
        return 0
    if x < min(Coins):
        return None
    candidates = []
    for coin in Coins:
        cand = makeChange(Coins, x-coin)
        if cand is not None:
            candidates.append(cand + 1)
    if len(candidates) == 0:
        return None
    return min(candidates)
```

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.

]

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.

]

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.

]