CS 161 (Stanford, Winter 2023) Homework 7

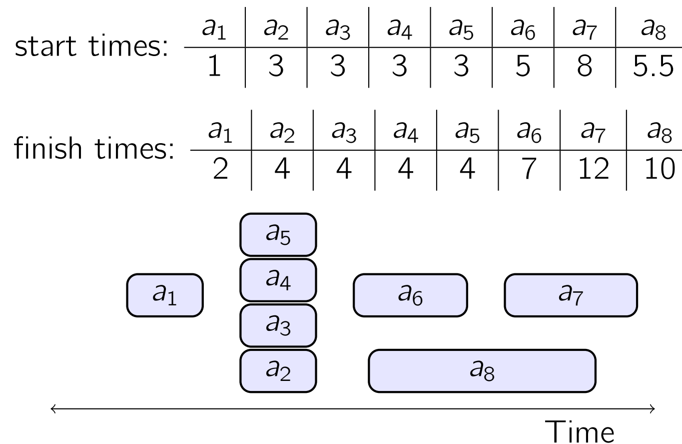
Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards. You should cite all sources you used outside of the course material.
What we expect: Make sure 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 Activity Selection (3 pt.)

Activity Selection is a classic algorithms problem. It works as follows: you are given a schedule with n activities, each of which has a start and finish time. You must select a subset of the greatest possible size, subject to the constraint that none of the activities 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 activity selection. The idea is that at each step, we greedily add a valid activity with the fewest conflicts with other valid activities. (An activity is *valid* if it doesn't conflict with an already selected activity).

```
def greedyActivitySelection(Activities):
  result = {}
  # initialize overlap counts array OV
  OV = initOV(Activities)
  while size(OV) > 0:
      a1 = argmin(OV) #activity with the least overlaps
      result += a1
      Activities -= a1
      Activities -= {activities that conflict with a1}
      #reinitialize OV using smaller schedule
      OV = initOV(Activities)
  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 Knapsacks with Nondiscrete Items

In this exercise we'll practice designing and analyzing greedy algorithms. We'll look at a continuous variant of the knapsack problem that we saw in class. You have a knapsack with a capacity of Q ounces and there are n items; the difference between this exercise and the version that we saw in class is that you can take a fractional amount of each item. For example, perhaps one item is 3.6 ounces of brightly colored sand; you can choose to take 2.5235 ounces of sand for your knapsack if that's how much you want.

Each item 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 item i available to you, and for any real number $x \in [0, q_i]$, the total value that you derive from x ounces of item i is $x \cdot v_i$.

Your goal is to choose an amount $x_i \geq 0$ to take for each item i in order to maximize the value $\sum_i x_i v_i$ that you receive while satisfying:

(1) you don't overfill the knapsack (that is, $\sum_i x_i \leq Q$), and

(2) you don'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 knapsack.

2.0 (0 pt.)

Suppose that you already have partially filled your knapsack, and there is some amount of each item left. What item should you 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 (3 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 **AND** an English explanation of what it is doing.
- 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 $t - 1$, prove that it holds for t.**]**

2.3 Ethics (2 pt.)

Suppose you work as an admissions officer at a prestigious university and you thought perhaps using an algorithm to help rank applicants would help streamline the admissions process.

However, you know that greedily admitting applicants with the highest SAT/ACT score would not necessarily result in the best class of students. Hence, you decide to take other criteria into account, such as community involvement, leadership and distinction in extracurricular activities, and personal qualities and character.

Using the concept of incommensurability, explain why an algorithm might have a difficult time deciding how to rank two applicants.

- Two things are incommensurable when we lack a common measure of value. Incommensurability makes it difficult to establish ranking relationships, such as "more than" or "less than, "better than" or "worse than."

[We are expecting: Two to four sentences explaining how two applicants can have values that are incommensurable.**]**

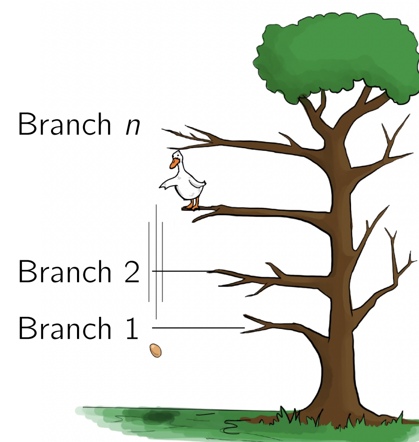
Problems. The following questions are problems. You may talk with your fellow CS 161-ers about the problems. However:

- 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 More Dynamic Programming!

Devon the Duck is making her nest in a Redwood tree. She sometimes drops her eggs, so she will place her nest in the highest branch at which she can drop an egg without it breaking. If an egg will break when dropped from branch i, it will also break when dropped from any branch j as long as $j \geq i$.

Chester the Chicken has kindly agreed to loan Devon some of his excess eggs so that Devon can run an experiment to determine which branch to place her nest on. Chicken eggs break when dropped from a branch if and only if Duck eggs do. Once it breaks, an egg can no longer be used to run experiments. Devon must be entirely certain of which branch she will place her nest on by the time she runs out of chicken eggs.



Count the minimum number of drops that Devon needs to make in the worst case, given that Chester has given her k chicken eggs.

[A Plucky the Penguin moment: if any of the branches on the tree would break Devon's eggs, then she nests on branch 0, which indicates that she nests on the ground. If there is one branch and it won't break her eggs, then Devon nests on branch 1. You won't need to worry about this too much in the following problems, just know that one branch on the tree implies two possible places to nest.]

For $n \geq 0$ and $k \geq 1$, let $D[n, k]$ be the *optimal worst-case number of drops* that Devon needs to determine the correct branch out of n branches using k eggs. 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 $D[1, j]$?

[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 egg from branch $x \in \{1, \ldots, n\}$. 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 which will compute $D[n, k]$ 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. **]**

4 Min Element Sum

Consider the following problem, MinElementSum.

> MinElementSum(n, S): Let S be a set of positive integers, and let n be a non-negative integer. Find the minimal number of elements of S needed to write n as a sum of elements of S (possibly with repetitions). If there is no way to write n as a sum of elements of S, return `None`.

For example, if $S = \{1, 4, 7\}$ and $n = 10$, then we can write $n = 1 + 1 + 1 + 7$ and that uses four elements of S. The solution to the problem would be "4." On the other hand if $S = \{4, 7\}$ and $n = 10$, then the solution to the problem would be "`None`," because there is no way to make 10 out of 4 and 7.

Your friend has devised a divide-and-conquer algorithm to solve MinElementSum. Their pseudocode is below.

```
def minElementSum(n, S):
    if n == 0:
        return 0
    if n < min(S):
        return None
    candidates = []
    for s in S:
        cand = minElementSum(n-s, S)
        if cand is not None:
            candidates.append(cand + 1)
    if len(candidates) == 0:
        return None
    return min(candidates)
```

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

4.1 (1 pt.)

Argue that for $S = \{1, 2\}$, your friend'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 $S = \{1, 2\}$.

- A convincing argument that the closed form for this expression is $2^{\Omega(n)}$. You do not need to write a formal proof.

]

4.2 (3 pt.)

Turn your friend's algorithm into a top-down dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

Hint: Add an array to the pseudocode above to prevent it from solving the same sub-problem repeatedly.

[We are expecting:

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

]

4.3 (3 pt.)

Turn your friend's algorithm into a bottom-up dynamic programming algorithm. Your algorithm should take time $O(n|S|)$.

Hint: Fill in the array you used in part (b) iteratively, from the bottom up.

[We are expecting:

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

]

5 Making Change

Lucky the lackadaisical lemur works at McDonald's, and often he has to make change. If a customer paid \$5 for an order that costs \$4.48, he would have to provide 52¢ in change. Assuming Lucky 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. Lucky's manager hates having extra change lying around, so *Lucky's goal is to use as many total coins as possible.*

Consider the more general problem: Lucky has k coins which are worth distinct values $v_1, v_2, \ldots v_k$, 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 maximize m, or in other words, to use as many total coins as possible.

5.1 Greedy Approach

Consider the following greedy algorithm for making change that Lucky came up with.

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

5.1.1 (2 pt.)

If it returns a valid way to make change, does this algorithm return one that is optimal (i.e. that uses the maximum 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.**]**

5.1.2 (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.**]**

5.2 Dynamic Programming Approach (5 pt.)

Design a DP algorithm which which computes the maximum number of coins Lucky can use to make change. Your algorithm should run in time $O(kx)$, where x is the amount you must make change for and k is the number of types of coins available to use.

[We are expecting: Pseudocode and a short English description of your algorithm. No runtime justification is necessary.**]**