Homework 3: Greedy, Divide and Conquer, DP 1

Version 2: We corrected a mistake in the description of the greedy change algorithm in problem 1 (if you are making change for k cents, and have a k cent coin, the greedy algorithm will take it).

Due Date: This assignment is due at 11:59 PM Friday February 5 (Seattle time, i.e. GMT-8).
except for the programming problem, which will be due one week later with HW4.

You will submit the written problems as a PDF to gradescope. Please put each numbered problem on its own page of the pdf (this will make selecting pages easier when you submit).

Remember to assign pages in your submission!

Collaboration: Please read the full collaboration policy. If you work with others (and you should!), you must still write up your solution independently and name all of your collaborators somewhere on your assignment.

Style and Assumptions: You should check the Style Guide for general principles on how to format your answers. You may use any algorithm described in this list without further explanation. If you wish to use an algorithm you learned in 373 (or 142/143) that isn't on the list, please ask on Ed so we can add it to the list for everyone.

1. Rich Uncle Pennybags [15 points]

1.1. Greed is Good

You've just arrived for a vacation in a country with a very algorithmically-minded currency designer. For some (integer) constant $b \geq 2$, they have coins worth b^k cents for every $k \geq 0$ (i.e. they have a 1 cent coin, a b cent coint, a b^2 cent coin, ...). The natural greedy algorithm for change-making ("add the highest value coin less than or equal to the total still needed") **is** optimal for this coinage system.

(a) Prove that for every integer $n \geq 1$, the greedy algorithm will produce the minimum number of coins to make n cents of change. [7 points]
Hint: Robbie's proof first proves the statement "The minimum set of coins will never have b copies of the same coin" (in 1-2 sentences) then argues about the algorithm. You may use this outline or do your own proof.

(b) Which of the greedy algorithm proof outlines (structural, greedy stays ahead, or exchange argument) do you think best describes your proof. Explain why in 1-2 sentences. [2 points]

1.2. Greed is Bad

(a) We saw in class that the greedy algorithm doesn't work for $1, 10, 15$ cent coins. Your proof in the last part (if it's correct) will **not** apply in this setting (because you shouldn't be able to prove a false thing). Find where your proof breaks down: identify a key assertion in your proof that is not true, and explain (in 1-2 sentences) why it doesn't work for this set of coins.
You should be finding a key point where the proof stops working. For example, you should not just say "the greedy algorithm doesn't work, because we saw a counter-example in class. Therefore the concluding sentence where I say 'the greedy algorithm is optimal' is not correct" Instead you should find the earlier point where the argumentation breaks down. [3 points]

(b) Standard U.S. currency uses: $1, 5, 10, 25, 50, 100$ cent coins. It turns out a greedy algorithm works for U.S. coins, but it doesn't take much to break that. Find positive integer values c and d such that if you added a c-cent coin to U.S. currency, making change for d cents with the greedy algorithm will require more than the minimum number of coins.
Write 1-2 sentences to justify that the greedy algorithm is not optimal for d with c added. [3 points]

2. Pizza Party [20 points]

When grading can happen in person, courses often have pizza parties. Before ordering, each TA designs a `Pizza Object` that describes their ideal set of toppings for a pizza.

- If more than half of the TAs agree **exactly** on what they want, that `Pizza` will be ordered.
- Otherwise, the default option of cheese pizza will be ordered instead.

More formally, you have an array of n `Pizza Objects`. Since `Pizza Objects` are quite complicated[1], you can call `.equals()` on them, but it takes a **long** time. Even worse, there's no `.compareTo()` implemented, nor a `.hashcode()`. So you can't sort these Objects, nor put them in a hash table.

In the case that more than half the TAs submit `equal Pizza Objects`, you should return that `Pizza`. Otherwise, you should return `Pizza.CheesePizza`.

In this problem, you'll describe a divide-and-conquer algorithm that requires $\mathcal{O}(t \cdot n \log n)$ time.

For simplicity, you may assume that the number of elements in the array is a power of 2.

(a) Write pseudocode (or English) for your algorithm. [12 points]

(b) Explain why if more than half of the TAs agree, you will return their pizza order.
You may do a formal inductive proof or give an informal explanation (but if your explanation is informal, be sure to explain both your base and recursive cases!).
You do not have to prove what your code does when cheese pizza is ordered. [4 points]

(c) Write a recurrence to describe the running time of your algorithm. When analyzing running time, assume that any call to `.equals()` will take t time. You should treat t as a variable in your running-time analysis of this problem (i.e. don't consider it a constant and have it "disappear" in the $\mathcal{O}$-notation). Briefly (1-2 sentences) justify your recurrence. You should convince yourself that the running time will be $\mathcal{O}(t \cdot n \log n)$, but you don't have to include that explanation. [4 points]

3. Force Dynamics [15 points]

This question will be **due with HW4** because we didn't get fully through this example during lecture 11. We still strongly suggest aiming to complete it this week, but want to make sure you have at least a full week of work time after seeing the lecture content.

In this problem, you'll implement Java code that corresponds to the high-level ideas we developed in Lecture 11.

The heart of the problem is the same as the one in lecture, but a few details have changed. In particular the coordinate system was changed to match how arrays are usually drawn on paper (this should make it easier for you to design your own test cases if you need to)

Baby Yoda needs to get from location $(r-1, c-1)$ to location $(0,0)$. He can only move up (decreasing the first coordinate) or to the left (decreasing the second coordinate). He cannot pass through a spot with rocks (unless he first uses the Force to knock over the rocks). Finally, he can collect an egg to eat by passing through that location.

Write a method, that given

- `bool[][] rocks`. `rock[i][j]` is true if and only if there is a rock in location (i, j).
- `bool[][] eggs`. `eggs[i][j]` is true if there is one egg in location (i, j) (and it's false if there are no eggs in location (i, j)).
- `int force`. `force` is a non-negative integer, with the number of times Baby Yoda can use the Force to knock over rocks.

[1] There are so many toppings! And crust styles. And sauces.

returns the maximum number eggs Baby Yoda can collect on his way to reach $(0, 0)$.
If Baby Yoda cannot reach $(0, 0)$, return -1.

You may make the following simplifying assumptions:

- `rocks[i][j]` and `eggs[i][j]` will not both be `1` for any location.
- `force` will be 0, 1, or 2.
- There is no need to validate input (e.g. you do not need to explicitly check that the dimensions of `rocks` and `eggs`) match, nor that `force` is not 3. We will never test invalid input.

Make sure you're following our updated indexing conventions – the row comes first, then the column. This is consistent with how arrays are usually drawn (the origin is in the upper left), but not consistent with the drawings from lecture. You may wish to look at the provided test case to help you visualize.

Your final response must be **iterative** not recursive[2], and it should run in time $\Theta(frc)$ where f is `force` and the map is $r \times c$.

You do not need to optimize memory (though you may if you wish).

[2]You can still use helper methods, but the main structure of your code needs to be loops, not using recursion.