Exam 2—D&C, DP, amortized analysis

In preparing your solutions to the exam, you are allowed to use any sources including your textbook, other students and professors, previous homeworks and solutions, or any sources on the Internet. You may ask me for feedback on potential solutions, but I will not give you any hints. Of course, I am also happy to answer general questions, go over homework problems, or answer clarifying questions about exam problems.

The exam will take place in class on Friday, November 10 (MC Reynolds 108, 1:10-2:00pm). You are not allowed to bring any notes, textbooks, calculators, or any other resources with you to the exam. Bring only something to write with; I will provide a fresh copy of the exam, paper for writing your solutions, and scratch paper.

As usual, to "design and analyze" an algorithm means to (a) describe the algorithm, (b) prove/justify its correctness, and (c) analyze its asymptotic running time. Full credit will only be given for the most efficient possible algorithms. Algorithms must be clearly explained (using pseudocode if appropriate) in sufficient detail that another student could take your description and turn it into working code. You may **freely cite any theorems proved in class** (without proof), or **use algorithms covered in class as subroutines**.

Question 1. Given an array $A[0 \ldots n-1]$ of integers, a *wobbly pair* is a pair of integers in the array that are "out of order": that is, where $i < j$ but $A[i] > A[j]$. For example, the array

$$A = [2, -1, 17, 10, 3, 8]$$

has 6 wobbly pairs, namely, $(2, -1)$, $(17, 10)$, $(17, 3)$, $(17, 8)$, $(10, 3)$, and $(10, 8)$. Put another way, if you imagine each number "looking" down the array to its right, there is a wobbly pair each time a number can "see" another number which is smaller than it. The number of wobbly pairs is in some sense a measure of how far away A is from being sorted; in fact, the number of wobbly pairs is exactly the minimum number of *adjacent swaps* needed to sort the array, and a sorted array has zero wobbly pairs. In the example, we could first swap 17 and 10, then 17 and 3, then 17 and 8, taking three swaps to move 17 to the end; then two more swaps would be needed to move 10 past 3 and 8; then one last swap would put 2 and -1 in the correct order, for a total of six adjacent swaps.

(a) Describe, and analyze the running time of, a simple brute-force algorithm to compute the number of wobbly pairs in a given array.

(b) Now design and analyze a more efficient algorithm to compute the number of wobbly pairs in a given array.

Question 2. You are given a set of n widgets and a positive integer G. Each widget also has a *froob* f_i (a positive real number) and a *grump* g_i (a positive integer). As you can tell from their names, froob is good and grump is bad. Your goal is to pick a subset of the widgets such that their total froob is as big as possible (yay!), but subject to the constraint that their total grump must be $\leq G$ (you can only deal with so much grump).

(a) Design and analyze an $O(nG)$-time algorithm to find an optimal subset, given as input the number of widgets n, the maximum grump G, and two size-n arrays containing the froob and grump values for the widgets. Be sure your algorithm finds not just the maximum possible froob but an actual subset of widgets which has that total froob.

(b) Explain (one sentence should be sufficient) why the problem would be much harder if the grump values were allowed to be positive *real numbers* rather than just positive integers.

Question 3. Suppose you are maintaining an array $X[0 \ldots n-1]$ of n counters, each counter taking a value from the set $\{0, 1, 2\}$. The following algorithm increments one of the counters; if the counter would overflow, the algorithm instead resets it to 0 and recursively increments its two neighbors. We think of the array as being circular, so counters 0 and $n-1$ are also neighbors; this is why lines 5 and 6 of the code below use $\mod$ to make the indices wrap around.

Question 3 is also courtesy of Jeff Erickson, `http://jeffe.cs.illinois.edu/teaching/algorithms/hwex/f13/midterm2.pdf`.

Algorithm 1: INCREMENT(i)

```
1: if $X[i] < 2$ then
2:     $X[i] \leftarrow X[i] + 1$
3: else
4:     $X[i] \leftarrow 0$
5:     INCREMENT($(i-1) \bmod n$)
6:     INCREMENT($(i+1) \bmod n$)
7: end if
```

For example, if $X = [0, 0, 0]$ then INCREMENT(1) results in $X = [0, 1, 0]$; if $X = [0, 2, 0]$ then INCREMENT(1) results in $X = [1, 0, 1]$; if $X = [2, 1, 0]$ then INCREMENT(0) results in $X = [0, 2, 1]$.

(a) Suppose $n = 5$ and $X = [2, 2, 2, 2, 2]$. What does X contain after we call INCREMENT(3)? (You do not need to show any work or intermediate steps, just the final contents of X.)

(b) Suppose all counters are initially 0. Prove that INCREMENT runs in $\Theta(1)$ amortized time. Assume as the cost model that changing any $X[i]$ costs 1 unit. Caution: you may not assume that INCREMENT is called repeatedly on the same location! Your proof should work for any sequence of INCREMENT calls, no matter which indices they increment. (Hint: use the accounting method!)