COSC 240, Spring 2020: Problem Set #3

Assigned: Thursday, 2/13
Due: Thur, 2/27, at the beginning of class (hand in hard copy).
Lectures Covered: 10 to 13.
Academic Integrity: You can work with other people in the class but you must write up your own answers in your own words. You can also use the textbook and talk to the professor. You may not use any other resources (e.g., material found online) or talk to people outside the class about these problems. See the syllabus for details on the academic integrity policy for problem sets.

Amortized Analysis

Assume you run a testing center where students arrive throughout the day to take the GRE. (For simplicity, assume students never leave). Your testing center has a classroom of size 2^i, for each $i \geq 0$. You only have one proctor, however, so all students at the test center at any given time must be in the same classroom. You have also noticed that students perform poorly if a room seems empty, so you insist on the rule that the room containing the students at any given time must have more full than empty seats.

To accommodate these constraints you use the following algorithm to handle each new arrival of a student:

- Start the day using the smallest available classroom (which is size $2^0 = 1$).
- Once you run out of space in a classroom of size 2^i (e.g., the room is full and then a new student arrives), move all of the existing students and the new student to the classroom of size $2 \cdot 2^i = 2^{i+1}$.

You are concerned about the cost of this room scheduling policy. In particular, you calculate the cost of each student arrival to be the number of students that must be moved (including the new student) to accommodate the arrival. More formally, consider a sequence of n arrivals. Let c_i be the cost of the i^{th} student arrival. You can define $c_i = 1 + m_i$ where:

$$m_i = \begin{cases} i-1 & \text{if } i-1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

The three questions that follow ask about this above scenario. The purpose of these questions is to explore how the three types of amortized analysis studied in class might help you more accurately understand the arrival costs over time in this scenario.

1. Prove that arrivals have a constant amortized cost by using the *aggregate analysis* technique discussed in class (and Chapter 17.1 of the textbook).
2. Prove that arrivals have a $\log n$ amortized cost, assuming n total students arrive, by using the *accounting method* technique discussed in class (and Chapter 17.2 of the textbook).
3. **Optional Extra Credit.** Provide an improved version of your answer to the previous problem that now applies the accounting method to establish a constant amortized cost.

   (Hint: students who arrive after a classroom expansion might want to help the students who arrived before the expansion regain the credit they will need to pay for their movement in the next expansion.)

4. We now turn our attention to a different problem. Consider a simple unary counter that counts from 1 to k before wrapping back around to 1 (for some $k \geq 1$). One way to implement this counter is with an array A of size k. Initially $A[1] = 1$ and $A[j] = 0$, for $2 \leq j \leq k$. A variable p keeps track of the smallest position in the array that currently stores a 0. If there are no 0's in the array, then $p = k + 1$. We initialize $p \leftarrow 2$.

   To INCREMENT the counter, there are two cases. If $p = k + 1$, loop through A and set each position to 0, then set $A[1] \leftarrow 1$ and $p \leftarrow 2$. Otherwise, if $p \leq k$, then set $A[p] \leftarrow 1$ and $p \leftarrow p + 1$.

   Let c_i be the cost of the i^{th} INCREMENT. If we focus only on changing array entries when calculating cost, we can use the following definition for c_i:

$$c_i = \begin{cases} 1 & \text{if } i \text{ is not a multiple of } k, \\ \text{k+1} & \text{else.} \end{cases}$$

   In the worst case, therefore, $c_i = k + 1$. Prove that INCREMENT has constant amortized cost using the potential method.

Dynamic Programming

1. Consider the following problem. You run ticket sales for a baseball stadium and are trying to sell the valuable seats in the row right behind the home team dugout. There are n total seats in the row.

   For each $i \in \{1, 2, ..., n\}$, let p_i be the price for a *contiguous block* of i seats (i.e., a block containing seats j, $j + 1$, ... , $j + i - 1$, for some $j \in \{1, 2, ..., n - i + 1\}$). These prices were set by a complicated pricing algorithm so they do not necessarily increase with group size (it might be possible, for example, that $p_7 < p_4$, as groups of 4 are more common than groups of 7, and so on). With this in mind, you should not assume anything about these p_i values other than the fact that they are all integers greater than 0.

   Your goal, given a set of p_i values, is to figure out how to break up the dugout row into contiguous blocks so as to maximize the money you make selling the blocks for their corresponding block prices. (For example, if $n = 6$ and you break the row into one block of size 3, and three blocks of size 1, then you would earn $p_3 + p_1 + p_1 + p_1$ by selling the row in blocks of those sizes.)

   Because there are an exponential number of different ways to break up the row into blocks, standard brute-force algorithms are too slow. The **three-part** problem that follows asks you to develop a more efficient dynamic programming solution.

   (a) For each $i \in \{1, 2, ..., n\}$, let $q[i]$ be the maximum amount of money you can make breaking up the first i seats of the row into contiguous blocks of size i or less. It is clear to see that $q[0] = 0$, $q[1] = p_1$, and $q[n]$ is the final answer you are trying to calculate.
   Define $q[i]$ with a recurrence by filling in the blank line in the following:

$$q[i] = \begin{cases} 0 & \text{if } i = 0 \\ p_1 & \text{if } i = 1 \\ \text{----------} & \text{if } i > 1 \end{cases}$$

   (Hint: the answer I have in mind includes a max statement containing i different values.)

   (b) Using your answer from part (a) of this problem, design a dynamic programming algorithm that calculates $q[n]$.

(c) Show how to update the algorithm from part (b) so that it in addition to returning $q[n]$, it also prints the sizes of the blocks whose prices add up to $q[n]$. You can either write a new algorithm from scratch, or just describe the new lines required for this printing and specify where they should be added to your part (b) solution.