Algorithms
Computer Science 140 & Mathematics 168
Spring 2012
Homework 4b
Due Tuesday, February 14

The first exam will be on Tuesday, February 21. Part 1 of the exam will be in-class and will comprise short questions that survey the topics and ideas that we've covered so far: Divide-and-conquer, dynamic programming, greed, and amortization. You may prepare an 8.5×11 sheet with notes on both sides to bring to the in-class part of the exam. You will then take Part 2 of the exam home with you. This part of the exam will comprise two homework-style problems. You can take that part of the exam in a 3-hour block of your choosing and those problems will be due at the beginning of class on Thursday, February 23. For the take-home part of the exam, you may use your 8.5×11 sheet, your own lecture notes, and your homework solutions and our sample solutions, but no other resources.

1. [**20 Points**] **Four Russians at Millisoft!** Recall that the edit distance function is used to determine the "distance" between two strings. Specifically, the edit distance from string $S1$ to string $S2$ is defined to be the minimum number of insertions, deletions, and substitutions needed to get from $S1$ to $S2$. As we argued in class, the edit distance from $S1$ to $S2$ is the same as the edit distance from $S2$ to $S1$. Here is the recursive function that we developed for edit distance in class. (There is a very slight difference between this implementation and the one in class, but the logic is identical. In the implementation in class, option 1 was considered in two different parts of the function, depending on whether `S1[i] == S2[j]` or not. Here, I've put option 1 in one place.)

```
ED(S1[1:i], S2[1:j])
   if i == 0:  return j
   elif j == 0:  return i
   else:
      if S1[i] == S2[j]:  option1 = ED(S1[1:i-1], S2[1:j-1]
      else: option1 = 1+ ED(S1[1:i-1], S2[1:j-1])
      option2 = 1 + ED(S1[1:i-1], S2[1:j])
      option3 = 1 + ED(S1[1:i], S2[1:j-1])
      return min(option1, option2, option3)
```

(a) Use the extremal method to show that as we fill in the dynamic programming (DP) table row-by-row, left-to-right, a given cell in the DP table can differ from its preceding cell by at most 1. That is, it cannot be more than one larger or more than one smaller than its predecessor. (After you've shown this, you can just claim that by symmetry the analogous property is true for the columns.)

(b) Millisoft would like to use a Four Russians dynamic programming implementation of ED for its next-generation spell-checker. The character set that is used in Millisoft's product contains $3^5 = 243$ distinct symbols. For simplicity, assume that the table is $n \times n$ (that is $S1$ and $S2$ have the same length). In a few sentences, describe how the Four Russians DP will work. That is, in a few sentences describe what the tile types will look like and then describe how those tiles will be used to compute the edit distance between our two strings.

(c) In a few sentences, explain the derivation of the running time of your algorithm as a function of n and the tile width/height k.

(d) Give a value of k, the tile width/height, that results in your DP running in time $\Theta(\frac{n^2}{\log n})$. The value of k will have some specific constants in there that are important and can't be disregarded.

2. **[25 Points] Stack + Stack = Queue.** The Shmorbodian Division of Millisoft Research has a really great implementation of a stack. Like any good stack implementation, it supports PUSH and POP in $O(1)$ time. Your boss wants you to implement a queue, but not from scratch! Your boss conjectures that a queue can be implemented using two stacks.

   Describe how a queue can be implemented using two stacks. Then, prove that your stack-based queue has the property that any sequence of n operations (selected from ENQUEUE and DEQUEUE) takes a total of $O(n)$ time resulting in amortized $O(1)$ time for each of these operations! Your proof may use any one of the three amortized analysis techniques that we discussed in class.

3. **[30 Points] Minqueues!**

   A Minqueue is an abstract data type (ADT) that supports the following operations:

   ENQUEUE(x): Inserts the number x into the Minqueue.

   DEQUEUE(): Removes the element that has been in the Minqueue for the longest time.

FIND-MIN(): Returns the smallest value in the Minqueue but *does NOT* remove it from the Minqueue.

Your officemate at Millisoft, Dr. Anna Litik, has proposed a clever data structure for this problem which she calls "Real Queue and Helper Queue." Here's how it works: When an element is enqueued, it is placed into a regular queue (implemented as a doubly linked list). However, it is *also* placed at the tail of a special "helper queue" (also implemented as doubly linked list). The helper queue will always contain a subset of those elements in the real queue in sorted order from small elements at the front to large elements in the back. In particular, when an element x is inserted at the tail end of the helper queue, it checks to see if it is smaller than the element right in front of it in the helper queue. If so, it removes the element in front of it in the helper queue and again compares itself to its predecessor. It repeatedly removes its predecessors until it gets to a point that its predecessor is smaller than it! To dequeue, we simply remove that element from the head of the real queue. We also check to see if it is at the head of the helper queue, in which case we remove it from that queue as well. Finally, to determine which element is the minimum, we simply look at the front of the helper queue! Here's your task: Prove that the total running time of *any* sequence of n operations in time $O(n)$ time (or, as we sometimes say, $O(1)$ amortized time per operation). To build your amortization muscles, please give **three different** proofs of this result, one using the **aggregation method**, one using the **accounting method**, and one using the **potential method**.

4. **[25 Points] Optional Bonus Problem: Glass Balls!** Imagine that you work in a building that is n stories tall and you own two completely identical glass balls. Your objective is to determine the lowest floor k such that dropping a ball from floor k will cause it to break (but dropping a ball from any lower floor will not cause it to break). This is called the "lowest breaking floor." You may assume that the ball will break at some floor. Your objective is to determine the lowest breaking floor in the least number of ball drops. Of course, you could do it this way: Drop a ball from the first floor. If it breaks, the answer is 1. If it doesn't break, go up to the second floor and try again. If the ball breaks now, the answer is 2. If the ball doesn't break, try the next floor. This strategy works and actually only requires one ball. However, it requires n drops in the worst case. Your objective is to improve the number of ball drops substantially using your two identical balls. Describe an algorithm that determines the least breaking floor in the least number of ball drops **and**

prove that your algorithm uses the least number of ball drops! (*Hint:* You may wish to begin by assuming that n has a convenient form. You shouldn't constrain n to be a constant, but you may wish to begin by constraining it to be in a convenient form like a power of 2, a perfect square, etc. Your proof of optimality is likely to use strong induction.)