CSE 5211: Analysis of Algorithms Fall 2025
Assignment 02: Fun with Amortization and Priority Queues
Total Points: 16

Please see Homework 01 handout for information on group composition, and the preamble to all assignments.

1. **Binary Counter by Cheating (4 points):** In Lecture 02, we saw how to implement an *increment* operation of an n-digit binary counter in $O(1)$ amortized time. We store the digits of an n-bit binary number in an array A of length n. We wish to increment our number from 0 to $2^n - 1$, updating A accordingly. For example, if $n = 3$, we start with $A = [0\ ;0\ ;0]$, then we toggle the last entry to obtain $A = [0\ ;0\ ;1]$, eventually reaching $A = [1\ ;1\ ;1]$ after several increments. Suppose that you are allowed to "cheat" and can have the digit 2 in your counter. The counter's decimal value is still $\sum_{i=0}^{n-1} 2^i a_i$ (assuming we go from a_0 to a_{n-1} in increasing significance of digits). So we now have several different ways to represent a number, for example, the decimal 4 can be represented in the array as $[1; 0; 0]$ or $[0; 2; 0]$ or $[0; 1; 2]$. Please show that introducing the 'bit' 2 allows us to perform the *increment* operation in $O(1)$ worst-case time.

2. **Enumerating Permutations (4 points):** Consider a length-n array $A = [1\ ;2\ ;3\ ;\ldots\ ;n]$. We would like to step through all $n!$ permutations of A, updating the array as we go to represent each subsequent permutation. Permutations should be generated in lexicographic order; for example, with $n = 3$ we start with $A = [1\ ;2\ ;3]$, then move to $A = [1\ ;3\ ;2]$, $A = [2\ ;1\ ;3]$, $A = [2\ ;3\ ;1]$, $A = [3\ ;1\ ;2]$, and finally $A = [3\ ;2\ ;1]$. Please describe how to implement an operation *next-perm* with $O(1)$ amortized running time that modifies A to produce the next permutation in lexicographic order.

3. **Lazy Deletion (4 points):** Consider the three approaches we discussed for mergeable heaps: size-augmented heaps, null-path-augmented heaps, and leftist heaps. In each of these, *delete*, *decrease-key*, and *increase-key* are problematic due to the need to maintain augmented information. Why is this, and how can we fix this issue by being somewhat "lazy" so that all priority queue operations run in $O(\log n)$ amortized time?

4. **A "Binomial Heap" Comprised of Sorted Arrays (4 points):** At the beginning of our discussion on binary heaps, we briefly discussed the use of a single sorted array as a simple but inefficient means of implementing a priority queue. In this problem, we consider a generalization of this idea that uses several sorted arrays (or linked lists, if you think this is preferable) of different lengths. Each priority queue element will be stored in one of these arrays. Let us require that for each array in our collection (say it has length L), the next-largest array must have length at least $2L$. Since this property ensures that we will have at most $O(\log n)$ different sorted arrays in our data structure, *find-min* should take $O(\log n)$ time. To ensure that this property is maintained, we augment each array with its length, and we store these arrays in a linked list in order of their lengths. To insert an element e into the structure, we insert a new length-1 array containing only e. If this happens to violate the property above (i.e., if there already exists a length-1 array), we repeatedly merge the array

containing the new element with the next-largest array until the property finally becomes satisfied (using the same linear-time merge operation as in merge sort). To perform *remove-min*, we first locate the minimum element at the end of one of our arrays and then remove it by shortening the array. Again, this might violate the property above if our array shrinks to less than twice the size of the next-smallest array, so in this case we again perform successive merges until the property is restored. Both *insert* and *remove-min* have $O(n)$ worst-case running times. However, for a challenge, can you prove that both *insert* and *remove-min* run in only $O(\log n)$ amortized time? Is it possible to implement the remaining priority queue operations *decrease-key*, *increase-key*, and *delete* in $O(\log n)$ amortized time as well?

Submission: Please submit your solution as a single **pdf** file named `h02.pdf` on Canvas. Your solution must be written in a 12pt font size, and double-spaced. Please limit your answers to 2 pages per problem.