Take-Home Midterm — *Applied Algorithms, T. II/2012–23*

(due: fri feb 24 @ 11:59pm)

Ground Rules:

- There are 4 problems. Do *any* 3 of them; if you do all 4, the worst 3 will be credited. Solve them by yourself—no collaboration.
- Typeset your answers and hand in a PDF file electronically to Canvas.
- You can look things up on the Internet and elsewhere; refrain from blindly copying solutions. If you do look something up, please cite your sources.
- For multi-part problems, if you can't solve a subproblem but need it subsequently, you can pretend you've solved it and invoke it in the following parts.

Problem 1. [100 points] *Dynamic Binary Search.* For the most part, binary search is reserved for static sequences: for a length-n sorted array, binary search takes $O(\log n)$ time to locate an element in the worst-case. But to keep the array sorted, we need $O(n)$ time to insert a new element. We wish to extend binary search to support fast insertion, and will design and analyze a data structure that comes close to the performance of a binary search tree by maintaining several sorted arrays.

Our goal is to support `insert(x)` and `find(x)` operations. To do so, we will maintain the following invariant: Let n be the current number of elements in the collection. With $k = \lceil \log_2(n+1) \rceil$, we can write n in binary as $n = (n_{k-1}n_{k-2} \dots n_0)_2$. Based on this, we will be keeping k sorted arrays $A_{k-1}, A_{k-2}, \dots, A_0$, where each A_i (1) has length exactly 2^i, (2) is either full (if $n_i = 1$) or empty (if $n_i = 0$), and (3) is fully sorted.

While each individual array is sorted, no relationship between arrays is enforced. Furthermore, observe that the total elements stored across the k arrays is $\sum_{i=0}^{k-1} n_i 2^i = n$.

(a) Describe how to support `find(x)`. What is the worst-case running time? (*Hint:* If we don't know which array `x` appears, we can try every array. There are at most k arrays to try.)

(b) Describe how to support `insert(x)`. What is the worst-case running time? How about the amortized running time? (*Hints:* Some `insert`s are costly but not all of them are. Take inspiration from binary counters—when we increment a counter by 1, sometimes carry has to propagate quite far, but this can't happen too often. Also, remember that if we have two sorted arrays X and Y, we can merge them into a single sorted array in $O(|X|+|Y|)$ worst-case time using a standard merge routine from merge sort.)

Problem 2. [100 points] *Probabilistically* In an alternate universe, SS lives on a number line: his home is at 0 and the line extends indefinitely in both directions all the way to $+\infty$ and $-\infty$. One bright morning, SS wants to take a walk randomly for $2n$ steps, $n > 0$. A step changes his position on the number line by 1 depending on the direction of his walk. To walk randomly, SS tosses a fair coin at each step: a heads +1 to his position, and a tails, -1 to his position.

(a) After $2n$ steps, what is the probability that he is at home where he started?

(b) After $2n$ steps, how many times in expectation did he come back home? As an example, suppose the tosses were `HTHHTTTT`; he would have come home twice. (*Hints:* The thing has a closed form. Feel free to use Wolfram Alpha and the like. In lieu of a closed form formula, a plot showing the trend for n up to 1000 is fine as well)

Problem 3. [100 points] *Make Distinct.* The problem of deduplication (i.e., make the elements distinct) is a common routine in database engines (e.g., `SELECT DISTINCT ...`) and in data processing more broadly. Given an array (vector) of elements, the goal is produce a new array containing all the input elements (in any order) but without duplicates. Consider the following two approaches to deduplication:

(i) UNIQUE-HASHSET: Add all the elements into a hash-based set and generate the resulting array by going through the set.

(ii) UNIQUE-SORTED: Sort the elements, noting that in the sorted array, identical elements will be adjacent—and generate the resulting array as we go through the sorted array.

Design an experiment to study when which approach should be preferred. It is important to pay attention to at least (i) the diversity of data types (e.g., `i32`, `f64`, `String`, mixed records), (ii) the size of the input array, and (iii) the fraction of repetition (i.e., how much shorter is the output compared to the input?). We care about time performance (speed) and space needs (memory footprint). For this problem, assume everything is run single-threaded.

Problem 4. [100 points] *LP Fun.* Consider the following linear program (LP):

$$\begin{array}{ll} \text{Maximize} & 2x_1 + x_2 \\ \text{Subject to:} & -2x_1 - x_2 \leq -1 \\ & x_1 - x_2 \leq 3 \\ & 4x_1 + x_2 \leq 17 \\ & x_2 \leq 5 \\ & x_2 - x_1 \leq 4 \\ & x_1, x_2 \geq 0 \end{array}$$

(a) Use `scipy` to solve this LP.

(b) Find the dual LP of the above program.

(c) Use `scipy` to solve the dual program.

(d) How do their objective values compare?