CS161 Homework 2 **Due:** 22 April 2016, 12 noon
Submit on Gradescope **Handed out:** 15 April 2016

Instructions: Please answer the following questions to the best of your ability. If you are asked to show your work, please include relevant calculations for deriving your answer. If you are asked to explain your answer, give a short ($\sim$ 1 sentence) intuitive description of your answer. If you are asked to prove a result, please write a complete proof at the level of detail and rigor expected in prior CS Theory classes (i.e. 103). When writing proofs, please strive for clarity and brevity (in that order). Cite any sources you reference.

1 (15 points) LexicoSort

In class, we learned how to use CountingSort to efficiently sort sequences of values drawn from a set of constant size. In this problem, we will explore how this method lends itself to lexicographic sorting.

Let $S = `s_1 s_2 ... s_a`$' and $T = `t_1 t_2 ... t_b`$' be strings of length a and b, respectively (we call s_i the ith letter of S). We say that S is *lexicographically less than* T, denoting $S <_{\text{lex}} T$, if either

- $a < b$ and $s_i = t_i$ for all $i = 1, 2, ..., a$, or
- There exists an index $i \leq \min\{a, b\}$ such that $s_j = t_j$ for all $j = 1, 2, ..., i-1$ and $s_i < t_i$.

Lexicographic sorting algorithm aims to sort a given set of n strings into lexicographically ascending order (in case of ties due to identical strings, then in non-descending order).

For each of the following, describe your algorithm clearly, and analyze its running time and prove correctness.

(a) Give an $O(n)$ lexicographic sorting algorithm for n strings where each string consists of exactly one letter from the set a-z (that is, each string is of length 1).

Solution: (3 points)

Algorithm and Running time: If we map the letters a to z to the integer values 1 to 26, this becomes an instance of the CountingSort algorithm presented in class, which has runtime $O(n + 26) = O(n)$.

Correctness: Because each string is of one letter, the mapping preserves the lexicographic ordering of input strings. Since we proved that the CountingSort algorithm correctly sorts the input array (of integers), correctness of our algorithm follows.

(b) Give an $O(mn)$ lexicographic sorting algorithm for n strings where each string contains letters from the set a-z and is of length at most m. (Hint: pad words of length $< m$.)

Solution: (6 points)

Algorithm: First, we pad strings of length $< m$ to strings of length exactly m by appending spaces at the end; we use the same mapping from Part (a) while mapping each space to 0 (so each letter is mapped to an integer in $[0, 26]$ now). We then perform the CountingSort algorithm for m iterations as follows. During the first iteration we sort the n strings based on their last characters. Then, for $i > 1$, during the ith iteration we sort the n strings based on the ith-to-last character (i.e. the ith counting right to left).

Correctness: First, padding strings with 0's preserves the lexicographic ordering between any strings (to see why, if $S <_{\text{lex}} T$ and at least one of S and T is padded with 0's, then either S is a prefix of T (the first case defined above) or there is some index i such that prefixes of S and T of length $i-1$ are identical while $s_i < t_i$; in both cases, padding with extra 0's do not change their ordering after we map alphabets to integers).

Now we show inductively that the algorithm described above returns strings sorted in lexicographic order. For the base case, let the strings have length 1. Then this is simply a case of CountingSort, which we have proven in class (or in Part (a)). Suppose we have strings of length $m > 1$. If we consider the suffixes of length $m-1$ (all but the most significant character for each string), we can assume inductively that our algorithm correctly sorts them in $m-1$ iterations. Now, in lexicographic order, words are sorted according to the most significant character, and then words with the same such character are sorted by their suffixes. But this is exactly what happens if we first order by the suffixes, then sort by the leftmost character, since CountingSort preserves the relative order for words with the same leftmost character (this is known as "stability" as we learned in class; see Lecture notes #6). So our algorithm correctly sorts words of length m in m iterations.

Running time: The total runtime is $O(mn + m(n + 27)) = O(mn)$ (the first mn term for padding with 0's and the second term for running the CoutingSort algorithm m times).

(c) Give an $O(mn + mc)$ lexicographic sorting algorithm for n strings of maximum length m with letters drawn from a character set of size c.

Solution: (6 points)

We can use the same algorithm as in Part (b), except that at each iteration, the runtime is $O(n + c)$ instead of $O(n + 27) = O(n)$, since the character set has size c (where c cannot be omitted anymore because it is not a fixed constant). Thus, the final runtime is $O(mn + m(n + c)) = O(mn + mc)$ as desired (again, the first mn term for padding and the second term for the algorithm). Correctness of the algorithm does not change.

2 (15 points) Po Flips Pancakes

Poe the penguin has found a rare ingredient: Kaili yams. He decides to use this special ingredient to make a rare treat: pancakes! After much work, Poe the penguin has made a huge stack of n pancakes on a plate. The pancakes are labelled 1 to n from top to bottom, and pancake i has size P_i ($P_i > 0$ and larger P_i means larger pancake). He decides to feed Barr the Bear with his new creation. Unfortunately, Barr the Bear is an artistic eater and wants the pancake tower to look like a pyramid (i.e. sorted pancakes with largest at the bottom). To appease the bear, Poe decides to use his special pancake tongs to rearrange the pancakes. With the tongs, he can flip the order of i^{th} pancake (from the top) to j^{th} pancake (from the top) for any $1 \leq i \leq j \leq n$. As an example, if the initial pancake sequence of sizes is $[1, 1, 5, 3, 3]$, using the tongs on pancakes 2 to 5 will result in $[1, 3, 3, 5, 1]$ (by flipping $[1, 5, 3, 3]$ into $[3, 3, 5, 1]$). This is a tiring operation however, and will cost Poe $j - i$ units of energy. Poe wants to find a sequence of pairs (i, j) such that flipping the pancakes by following this sequence of pairs will cause the pancakes to become sorted, and so that the chosen flips use little energy.

In this problem we will design algorithms to help Poe. These algorithms will take an array of pancake sizes $\{P_1, \ldots, P_n\}$ as input and will sort the pancakes, assuming that a "flip" takes constant time. (Other operations such as reading the array etc also take constant time, as usual.) The running time of the algorithm is then computed as usual but assigning $O(1)$ time per flip. In addition to running time, we will analyze the energy of the algorithm. The energy is defined as the sum of the energies of all flips performed by the algorithm.

(a) Suppose all pancakes are of size 1 or 2. Design an $O(n \log n)$ time algorithm that sorts the pancakes using $O(n \log n)$ energy. Analyze the running time and energy of your algorithm and prove correctness.

(Hint: Mergesort.)

Solution: (5 points)

Algorithm:

Split the pancakes into two equal halves, and recursively sort them. Notice that each half will be a sequence of 1's followed by a sequence of 2's. To make the current array of pancakes sorted, we need a

single flip operation which consists of a suffix of the left half (of 2's, if any) and a prefix of the right half (of 1's, if any).

Energy and running time analysis:

After two recursive calls, we need (at most) one flip operation described earlier, which uses $O(n)$ energy if there are n pancakes. Let $E(n)$ denote the amount of energy used for an array of n pancakes. Then we have $E(n) = 2E(n/2) + O(n)$, which yields $E(n) = O(n \log n)$ as desired.

This algorithm simply uses the MergeSort algorithm we learned in class, but we need to find a sub-array (during each merge step) of 2's followed by 1's. This can be implemented in linear time because we simply iterate the left half (right half) to find a longest suffix (prefix) of 2's (1's). Therefore if $T(n)$ is the running time of this algorithm on n pancakes, then the running time of this algorithm is $O(n \log n)$.

Correctness:

Proof of correctness follows from that of the MergeSort algorithm; this algorithm will always output a sorted array of pancakes as desired.

(b) Suppose the pancakes are of arbitrary positive sizes. Design an algorithm that runs in $O(n \log^2 n)$ expected time that sorts the pancakes using (expected) $O(n \log^2 n)$ energy. Analyze the running time and energy of your algorithm and prove correctness.

(Hint: Use the previous part to perform quicksort; even if you did not solve the previous part, you may assume that such algorithm exists, and use it as a black-box without justification.)

Solution: (10 points)

Algorithm:

Randomly pick a pivot p (among $O(n)$ pancake sizes). For all pancakes with size $\leq p$, mark them as 1. For all pancakes with size $> p$, mark them as 2. Then we move the pancakes so that all pancakes that are marked as 1 appear before pancakes that are marked as 2. To do so, we use the algorithm in part A. Let us call this marking and sorting procedure as the Post-pivoting procedure (for reference later). Then we recursively sort the pancakes with label 1 and pancakes with label 2 (as we would do in the QuickSort).

Energy and running time analysis:

Note that our algorithm has the same structure as the (randomized) QuickSort algorithm we learned in class, except that we introduced an additional procedure after picking a pivot p. The Post-pivoting procedure runs in time $O(n \log n)$ and uses energy $O(n \log n)$ as we analyzed in Part (a). The rest is analogous (stating this much is sufficient). Similarly, since the randomized QuickSort's runtime is $O(n \log n)$ in expectation, its recursion tree's depth is $O(\log n)$ in expectation (you can state this to get full credit). Therefore both the running time and energy usage is bounded by $O(n \log^2 n)$.

(Note that your algorithm should choose the pivot uniformly at random, rather than picking the first, last, or i-th element for this question, because you should not assume that the result of MergeSort on pancakes results in a uniform distribution of the pancakes).

Correctness:

Note that although we marked pancakes as 1's and 2's, after the algorithm from Part (a) terminates, we end up with an array of pancakes such that the first half only contains pancakes of size $\leq p$ and the second half only of size $> p$ (no need to prove this statement as it is given as a hint). From this, the correctness of our algorithm follows from that of the QuickSort algorithm.

3 (15 points) Approximate Sorting

We say a sequence of distinct numbers $a_1, a_2, \ldots a_n$ is b-approximately sorted if the following property holds: for each $i = 1, 2, \ldots n$, if a_i is the j-th smallest number in the sequence, then $|i - j| \leq b$. In other words, each

number is at most b positions away from its actual sorted index. A b-approximate sorting algorithm is an algorithm that takes a sequence of distinct numbers as input and produces a sequence that is b-approximately sorted.

(a) Show that given any n distinct numbers, the number of permutations of those numbers that are $\sqrt{n}$-approximately sorted is $O(n^{cn})$, for some $c < 1$.

Solution: (3 points)

Let $b_1, b_2, \ldots, b_n$ be n distinct numbers sorted in increasing order. If $a_1, a_2, \ldots, a_n$ is a $\sqrt{n}$-approximately sorted permutation of $b_1, \ldots b_n$, then a_i equals b_j for some $i - \sqrt{n} \leq j \leq i + \sqrt{n}$, so it can take on at most $2\sqrt{n}+1$ different possible values. There can therefore be at most $(2\sqrt{n}+1)^n$ approximately sorted permutations of the n numbers. Now, for all $n > 9^{10}$:

$$(2\sqrt{n}+1)^n \leq (3\sqrt{n})^n = (9n)^{n/2} < (n^{0.1} \cdot n)^{0.5n} = n^{0.55n},$$

and hence the number of approximately sorted permutations is $O(n^{0.55n})$. In fact, we picked 9^{10} a bit arbitrarily. For any constant $\epsilon > 0$ we can pick a constant n_0 depending on ϵ so that for all $n > n_0$, the number of approximately sorted permutations is $O(n^{(\frac{1}{2}+\epsilon)n})$.

(b) Use Part (a) to find a lower bound on the number of leaf nodes in the decision tree for any comparison-based $\sqrt{n}$-approximate sorting algorithm, and prove that any such $\sqrt{n}$-approximate sorting algorithm must have worst case complexity $\Omega(n \log n)$.

Solution: (6 points) Assume the approximate sorting algorithm takes $a_1, a_2, \ldots a_n$ as input, and produces $a_{i_1}, a_{i_2}, \ldots a_{i_n}$ as output. For each $i = 1, 2, \ldots, n$, if a_i is the j-th smallest number in the input sequence, then define $\sigma(i) = j$, so that σ is a permutation of $1, \ldots n$. Observe that $a_{i_1}, a_{i_2}, \ldots a_{i_n}$ is approximately sorted iff $\sigma(i_1), \sigma(i_2), \ldots, \sigma(i_n)$ is an approximately sorted permutation of $1, \ldots n$. By Part (a), we see that there are at most $O(n^{cn})$ approximately sorted permutations of n distinct numbers, and therefore σ can equal at most $O(n^{cn})$ distinct permutations (of $1, \ldots n$) for $a_{i_1}, \ldots a_{i_n}$ to be a valid output. In other words, only $O(n^{cn})$ distinct orderings of the n input numbers can map to the leaf node that produces $a_{i_1}, a_{i_2}, \ldots a_{i_n}$ as output. But there are $n!$ ways for the input to be ordered, and therefore there must be at least $\Omega(\frac{n!}{n^{cn}})$ leaf nodes. Using Stirling's approximation $\ln n! \approx n \ln n - n + O(\ln n)$, we have $\frac{n!}{n^{cn}} \approx e^{n \ln n - n + O(\ln n) - cn \ln n} = \Omega(e^{(1-c-\epsilon')n \ln n})$ (where $\epsilon' > 0$ can be any small positive constant and $c = \frac{1}{2} + \epsilon$ we found in Part (a)). In particular, because each comparison can yield at most two child nodes in decision tree, the height of the decision tree of any comparison-based sorting algorithm is $\Omega((1 - c - \epsilon')n \log n) = \Omega(n \log n)$.

(c) Let $k > 1$ be an arbitrary constant. Can we construct an $\frac{n}{k}$-approximate sorting algorithm that does better than $O(n \log n)$? (Hint: consider the selection algorithm from lecture).

Solution: (6 points)

We can use the selection algorithm to find the $\frac{n}{k}$-th smallest number in $O(n)$ time. Then the group of numbers smaller than or equal to it correspond to the $\frac{n}{k}$ smallest numbers, and this group can be found in linear time. We can the repeat the process to find the next $\frac{n}{k}$ smallest numbers, and so on so forth, until we have divided the list into k groups of $\frac{n}{k}$ numbers. We can then output these groups in increasing order, sorting within each group arbitrarily. As long as the groups are correct, no number will be off by more than $\frac{n}{k}$ places, so our list will be $\frac{n}{k}$-approximately sorted. Our algorithm takes $O(kn) = O(n)$ time, since k is constant.

4 (10 points) Testing membership

The goal of this problem is to determine if an element x is a member of a set S.

(a) Describe a deterministic algorithm to determine whether $x \in S$. What is the runtime of this algorithm?

Solution: (1 point)

Go through S and compare each of its elements to x. The runtime is $O(|S|)$.

(b) We will walk you through a randomized algorithm for determining whether $x \in S$. In this algorithm, we perform k independent "tests" on x, each test taking $O(1)$ time. The test are relatively reliable, so that:

- Given an element x such that $x \in S$, x always passes the test.
- Given an element x such that $x \notin S$, x fails the test with probability p.

The tests are all independent, so that whether x passes or fails one test does not influence its probability of passing or failing the other tests.

(i) What is the runtime of the randomized algorithm?

Solution: (1 point)

$O(k)$.

(ii) Given that x is not in S, what is the probability that x passes all k tests?

Solution: (2 points)

x passes a test with probability $1-p$. Since the tests are independent, the probability that x passes all k tests is $(1-p)^k$.

(iii) Let $\epsilon > 0$ be a constant. Suppose that after the k tests we label an element as a member of S if and only if it passes all k tests; otherwise we label it as being not in S. How large should k be so that for any $x \notin S$ the probability that it was incorrectly labeled as being in S, is at most ϵ. (This latter probability is called the "false positive rate".)

Solution: (3 points)

From the previous part we know that x passing all k tests has probability $(1-p)^k$ when $x \notin S$. This is the false positive rate we wish to bound above by ϵ.

That is, $(1-p)^k < \epsilon$ is what we want, and we get $k \log(1-p) < \log \epsilon$, or $k > \frac{\log \epsilon}{\log(1-p)}$.

(iv) Suppose we are certain that $x \notin S$, and we want to run as many tests as it will take to prove this, even if we have to sit there all day running tests. What is the expected number of tests we would need to run in order to prove that x is not a member of S? What is the variance?

Solution: (3 points)

If x fails the test, we stop immediately. This procedure follows a geometric distribution with success probability p.

The expected number of tests is $1/p$ and the variance is $(1-p)/p^2$.