Homework 3

COMP221 Spring 2025 - Suhas Arehalli

Complete the problems below. Note that point values are roughly correlated with effort, but inversely correlated with expected difficulty. Check the course website & syllabus for further instructions.

If any problem is unclear, or you think you found a typo, please let me know ASAP so I can clarify!

Problems

1. **An Even Quicker Return to Sorting** *(20pts)*

   Let's develop a new kind of sorting algorithm, based on the following idea:

   Suppose you only need to sort *strings*. A string s of length k can be written as $c_1 c_2 \dots c_k$, where each c is a letter of the alphabet Σ.[1]. Note that:

   - Letters are *totally ordered* (i.e., for any pair of characters, we know which appears first alphabetically - think Comparable in Java).
   - Strings are typically sorted under *lexicographic order* ("alphabetical" order), which you get by first comparing the first letter, and if they match, you break the tie by comparing the second letter, and so on.

   With this, we have a fairly neat sorting strategy for an Array A of strings: For each letter in the alphabet $c \in \Sigma$, gather all of the strings in an Array A that have c as their first letter together. You then know where these strings belong in a sorted version of A relative to strings with different initial first letters (all words that start with a come before words that start with b, etc.), but you don't know where they belong relative to each other (i.e., you haven't sorted the group of words that start with a amongst themselves!). Of course, you can sort all elements with the same first letter by making a recursive call that sorts each subarray that begins with the same first letter by the second letter!

   (a) (0 points) Get comfortable with the idea presented above by applying it to a list of strings. Work out the finer details before moving on.

   (b) Turn this idea into a piece of pseudocode for a function called PrefixSort(Array A) that uses a helper function PrefixSort(Array A, Integer i, Integer low, Integer $high$) that sorts $A[low \dots high]$ based on the ith character in each string. Perform this sort *in-place*, only manipulating A by *Swap*-ing the positions of elements. You may assume that

[1] We call the set of elements in an alphabet Σ as is conventionally done in, say Theory of Computation, but that means we need to be careful not to confuse this with summation notation!

- A has length n
- All strings have length $\leq k$ for some $k \in \mathbb{N}$
- Strings s with length $k' < k$ are padded with the null character $\_ \in \Sigma$ such that for $k' < m \leq k$, $s[m] = \_$. You may refer to k freely in your pseudocode.
- $\Sigma = \{\_, a, b, \ldots, z\}$
- That the letters are ordered such that $\_ < a < b < \cdots < z$

All of this padding/$\_$ business exists so that we can sort strings of different lengths, and to ensure that shorter strings that are prefixes of longer strings appear first. i.e., *duck* precedes *ducks*, and this is enforced mathematically by padding *d-u-c-k* as *d-u-c-k-*$\_$, and then noting that $\_ < s$ (the null character comes before the letter s, any any other letter for that matter!). *(8pts)*

(c) Write a recurrence relation for the time complexity $T(k, n)$ for this function, assuming an average-case where each input array always has a roughly equal distribution of first letters. Note that the time complexity is dependent on two factors (like you saw with graphs) — the max-length of the strings k, and the length of the array n. *(6pts)*

(d) Consider (informally) the time complexity in the worst case, as opposed to the average-case in the previous part, using a recurrence tree. How much work is done at each level? What is the height of the tree? Propose a worst-case time complexity in Big-O notation. *(6pts)*

****Hint****: How does k play into this? On a similar note, how does i change in each recursive call?

2. **Searching for Sorted Lists** *(20pts)*

Let's think about sorting a bit differently — as a variant of the search problem! Let P_A be a set that contains all permutations p of the n elements within an Array A. That is, we have n elements which we can shuffle into any order; P_A is a set that contains every possible ordering of those n elements, with each unique ordering called a *permutation* of those n elements.

For example, for $A = [3, 1, 2]$, $P_A = \{[1, 2, 3], [1.3.2].[2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$.

Here's our scenario for this search-for-sorting problem: At the start, we only know nothing about the relative ordering of $A[1], \ldots, A[n]$. But, we can pick two values $A[i]$ and $A[j]$, $1 \leq i, j \leq n$ with $i \neq j$ and call the method Compare on them, and Compare will then return whether $A[i] < A[j]$.

For example, if $A = [3, 1, 2]$ as above, Compare($A[1]$, $A[3]$) returns False because $3 \not< 2$.

Each time we call Compare, we will narrow the space of possible permutations of A that can be in proper sorted order. Our question will be simple: How many comparisons must you do to find the sorted permutation?

Our goal is to eventually prove what was promised in class, that comparison-based sorting (i.e., and sort that orders elements by relying on calls to Compare) is $\Omega(n \log n)$. We'll do this by showing that even with an (impossibly) good comparison-based sorting algorithm, we can construct a worst-case input that takes order $n \log n$ time.

(a) What is $|P_A|$? Or, equivalently, how many permutations are there of n elements? *(3pts)*

(b) Suppose that each element in A is *unique*. Prove that for each permutation of A $p(A) = [A[p_1], \ldots, A[p_n]] \in P_A$, with $p_1, \ldots, p_n$ representing the indices of A in some shuffled order, there exists some A such that $p(A)$ is sorted. That is, show that for some shuffled set of indices $1 \leq p_1, \ldots, p_n \leq n$, there exists some A of length n such that $[A[p_1], \ldots, A[p_n]]$ is sorted. *(7pts)*

For example, if $n = 3$, the permutation $p(A) = [A[3], A[1], A[2]]$ has a corresponding array $A = [2, 3, 1]$ that makes the permutation order sorted. Show that this holds *in general*, for any n and any permutation $p(A) \in P_A$.

****HINT****: Prove this is true *by construction*: Show how you can build A such that that initial array will always result in the the permutation being sorted!

(c) Note that under our scheme, COMPARE is the only way for us to understand the contents of A. By the prior part, we know that *any* permutation in P_A could correspond to the sorted solution. Suppose we run COMPARE$(A[i], A[j])$ once for some indices i, j and it returns true. Can we rule out any permutations $p(A) \in P_A$? Characterize precisely the permutations that cannot be sorted (i.e., describe the property of $p(A)$s in P_A that tells us that $p(A)$ can't be sorted once we have the result from COMPARE). Suppose that COMPARE returned false — what permutations can't be sorted now? *(4pts)*

(d) Now let's get to our actual search-for-sorting argument. We are doing a *worst-case* analysis for *an impossibly good* comparison-based sorting algorithm. This means that we are in the following game-like scenario: We pick an i and j to COMPARE, and our nemesis will pick the *worst* possible A for this circumstance (i.e., one such that COMPARE will return the value that will minimize the number of permutations we can eliminate from the running using our logic from the prior part). Given this set-up, what is the greatest number of permutations we could possibly hope to remove from the running with one call to COMPARE? *(3pts)*

****Hint****: Think Binary Search.

(e) Assuming that you can always pick i, j to eliminate the number of permutations you gave in the last part, how many calls to COMPARE do we need, at minimum, to find the right permutation? Explain, and then express this in terms of a big-Ω w.r.t. n. *(2pts)*

****Warning****: Remember that n is the length of the array, not the size of P_A (look at your answer to part 1 of this question!).

****Hint****: You may need to use an identity from the textbook to reduce the order of the time complexity you found.

(f) If all of the above is true, and all that you showed in the previous problem are true, there should be something that appears contradictory on first glance (double check your answers if not!). Explain why there isn't actually a problem here. *(1pt)*