Homework 3

COMP221 Spring 2026 - 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. **Searching for Sorted Lists** *(20pts)*

   Let's think about sorting as a variant of the search problem. Here, we are searching for a *permutation*, or re-arrangement, of the elements that puts the array into sorted order. One thing searching through permutations lets us do is be agnostic toward the actual values of within the array — we just want to know their relative order to sort them!

   Lets let P_n be the set of permutations of any array of length n $A[1 \dots n]$, where each permutation is notated as an array of *indices*. That is, for $n = 3$, we can write $P = [3, 2, 1]$ gives us the order $[A[3], A[2], A[1]]$. Generalizing a bit, an arbitrary permutation $P[1 \dots n]$ provides us the order $[A[P[1]], A[P[2]], \dots A[P[n]]]$. For convenience, we will call this array $P(A)$

   Here's our scenario for this search-for-sorting problem: At the start, we only know nothing about the relative ordering of $A[1], \dots, A[n]$. 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 $A = [3, 1, 2]$. Compare(A, 1, 3) returns False because $3 \not< 2$.

   Each time we call Compare, we will gain information that 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? In other words, we're looking for a lower bound on the number of comparisons necessary to sort an arbitrary array A.

   While it seems like we're inventing a new, strange sorting algorithm, this actually is a a high-level description of every sorting algorithm we've seen so far! While we don't explicitly state that we're ruling out permutations in a QuickSort or a MergeSort, we do repeatedly call an equivalent of Compare on pairs of elements to get information on the relative ordering of elements in our input array A and make decisions (swaps!) that bring us closer to being sorted. What our analysis here will argue is that we need at least $\Omega(n \log n)$ calls to Compare to guarantee we can even be certain what the right order of elements is. Put another way, if we don't have $\Omega(n \log n)$ calls to compare, I can construct an input that your algorithm sorts incorrectly!

(a) What is $|P_n|$? 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 $A[1 \dots n]$, $P[1 \dots n]$, there exists some A such that $P(A)$ is sorted. That is, show that for some permutation $P = [p_!, \dots, p_n]$, there exists some $A[1 \dots n]$ such that $[A[p_1], \dots, A[p_n]]$ 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 $p(A) \in P_n$.

****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[1]. By the prior part, we know that *any* permutation in P_n could correspond to the sorted solution. Suppose we run COMPARE(A, i, j) once for some indices i, j and it returns true. Can we rule out any permutations $P \in P_n$? Characterize precisely the permutations that cannot produce a sorted $P(A)$. Suppose that COMPARE returned false — what permutations can't produce sorted $P(A)$ now? *(4pts)*

(d) Suppose you could *partition* P_n into two parts, P_n^1 and P_n^2, without knowing which permutation sorts A. We then discard the part that doesn't contain the permutation that sorts A. What strategy for choosing a partition has the best worst-case performance? *(1pts)*

****Hint****: Imagine that a devious adversary always chooses the permutation that sorts A to be in the larger part of the partition. How should you design the partitions?

(e) With the previous parts in mind, show that for any strategy of selecting i, j to call COMPARE, There exists an A such that you can rule out at most $\frac{1}{2}|P_n|$ permutations. *(2pts)*

(f) Using the prior parts, how many calls to COMPARE do we need, at minimum, to find the right permutation? Explain, and then conclude that this gets you to an $\Omega(\log(n!)) = \Omega(n \log n)$ lower bound on runtime.

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

[1] elements of A should be treated like mystery boxes — it's costs an operation to their relative order!

2. **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 $c_1c_2 \dots c_k$, where each c is a letter of the alphabet Σ.[2]. 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 can design a pretty clever sorting strategy for an Array $A[1 \dots n]$ of strings:

1. 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 (this should take $\Theta(n)$ time).
2. You then know where these groups of strings belong relative to groups of strings with different first letters (all words that start with a come before words that start with b, etc.). Put them in their respective group orders.
3. 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!). Solve this by making a recursive call that sorts all the elements that begin with the same first letter by their second letter.
4. Repeat until we are sorted.

You'll be asked to write and analyze a sorting algorithm (PREFIXSORT) that implements this scheme.

For convenience, we can start with a pretty simple alphabet and some reasonable assumptions about strings, though always keep the potential for generalization in mind:

- All strings have length k for some $k \in \mathbb{N}$
- $\Sigma = \{0, 1\}$
- The letters are ordered such that $0 < 1$ (as expected).

Note that, with these assumptions, we are dealing with binary strings. Moreover, they are pretty much treated like k-bit unsigned integers (for those who have taken computer systems).

(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 — there are some ideas that you'll need to fil in, but should follow naturally.

(b) Turn this idea into a piece of pseudocode for a function called PREFIXSORT(Array $A[1 \dots n]$) that uses a recursive helper function PREFIXSORT(Array $A[1 \dots n]$, 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. *(8pts)*

[2] 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!

(c) Write a recurrence relation for the average-case time complexity $T(k, n)$ for this function, assuming an average-case where each input array always has an equal distribution of 0s and 1s in each position. Note that the time complexity is dependent on two factors (like you saw with graphs in Data Structures) — 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?

(e) 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)*