Homework 2

COMP221 Spring 2024 - Suhas Arehalli

Complete the problems below. This assignment is out of 100 points, and the point values of each question. Note that point values are roughly inversely correlated with expected difficulty.

- You may complete the implementation section using your choice of Python, C, or Java.
- Write up your solutions to all parts of this homework using some typesetting software (LaTeX/Overleaf is recommended, but any typeset PDF is acceptable).
- You'll submit this assignment through Github Classroom (link on Moodle), with your written solutions in a pdf named "[LASTNAME1]_[LASTNAME2]_Homework1.pdf".
- **The due date for this assignment will be posted on the course website.**
- **You may submit assignments in pairs.** Put both peoples' names at the top of your submission.
  - **Both people are expected to have contributed to all parts of the assignment** – work together, and make sure you and your partner are both convinced of your solutions.
  - If you discussed problems with other students, list their names at the top of your assignment (there will be no penalty for this — it's encouraged!).
  - However, don't look at other group's written solutions/code, or share your written solutions/code with other groups. **In addition, do not search around the internet for solutions to these problems.** Seek help from your partner and course staff, the textbook, and course materials. Consult the syllabus for more details on academic integrity policies.

Note that we may not have covered all of the topics involved in this assignment at the point of release. This is to give you time to complete the parts you can as early as you can. I recommend you make note of parts we haven't covered yet so you can come back to them.

1 Implementation (39pts)

1. **A Quick Return to Sorting**
   (a) Update your sort-timing code from HW1. To facilitate simpler testing, we want to be able to run your sorting algorithms regardless of the language you're using. The starter code in your Github repository contains sample code for loading in command-line arguments in the context of this assignment, implemented in C, Python, and Java. Familiarize yourself with how to handle these inputs, and then update your code to print out to the command-line

the elements of the provided array in sorted order, each in a new line. The provided bash file (hw2.sh) should be modified to reflect the way you've implemented the code such that running

```
./hw2.sh –ints 3 -7 6 2 10 –alg bubblesort
```

will print

```
-7
2
3
6
10
```

You may separate your command-line sorting code from your CSV-constructing code, but be sure to modify the provided bash file so that the above works. **The grading for the two implementation problems will depend on this being done correctly!**

(b) Using your updated sort-timing code, implement QuickSortLomuto, a QuickSort implementation using the Lomuto partition method (i.e., a left-to-right pass through the array). Use the pseudocode from the course notes as a guide. Make sure your implementation is written in such a way that it can be timed with your code from HW1. It may be helpful to put the recursion into helper functions, as discussed in lecture. *(20pts)*

(c) Implement QuicksortHoare, a QuickSort implementation using the Hoare partition method (Using two indicies that move from edge-to-middle). Use the pseudocode from Activity 9 as a guide. As before, make sure your implementation is written in such a way that it can be timed with your updated sort-timing code. *(19pts)*

(d) Following instructions from HW1, construct two more CSV files full of timing data for these two new sorting algorithms, this time named **"[LASTNAME1]_[LASTNAME2]_QuickSortLomuto_times.csv"** and **"[LASTNAME1]_[LASTNAME2]_QuickSortHoare_times.csv"**.

2 Algorithmic Analysis (60pts)

Complete Questions 1 and 2. Choose **one** of problems 3 or 4 to solve.

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

   You never stop talking about sorting in an Algorithms class! Here, we'll work on developing a sorting algorithm with 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.[1]. Note that:

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

[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!

With this, we have a fairly neat sorting strategy for an Array A of strings: For each letter in the alphabet $l \in \Sigma$, gather all of the strings in an Array A that have l 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 by the second letter!

(a) 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

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

All of this padding/l_{null} 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-*l_{null}, and then noting that $l_{null} < s$ (the null character comes before the letter s, any any other letter for that matter!). *(8pts)*

(b) Prove that this sorting algorithm is correct. Refer to the definition of correctness for sorting we discussed in class. *(6pts)*

****Hint****: Start by proving that the helper function works. What does correctness mean here? State correctness formally, prove it, and show that the call you make in the main PrefixSort then results in the intended sorted order.

(c) Write a recurrence relation for the **average**-case time complexity $T(k, n)$ for this function. Assume that the average input has letters in Σ roughly evenly distributed (i.e., grouping the strings in A by their ith letter creates roughly equal partitions, for any i in bounds). Note that the time complexity is dependent on two factors (like you saw with graphs in COMP128!) — the max-length of the strings, and the length of the array. *(4pts)*

(d) Analyze the recursive structure of this function (like the recursion trees we saw in class!). What is the height of this tree? How much work is done at each level? *(2pts)*

****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)*

Sorting is still around! But let's think about this differently, as a variant of binary search! Let P_A be a Set that contains all permutations 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 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 COMPARE them. As a result of this decision, we figure out whether $A[i] < A[j]$ or not.

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

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

(b) Suppose that for each $1 \leq i \leq n$, $A[i] \in \{1, \ldots n\}$. That is, the array A is filled with the integers $1, \ldots, n$ in some shuffled order. Prove that for each permutation $p = [A[p_1], \ldots, A[p_n]] \in P_A$, there exists some initial ordering of those integers in A such that p is sorted. *(5pts)*

For example, if $n = 3$, the permutation $[A[3], A[1], A[2]]$ 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 $p \in P_A$.

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

(c) Briefly explain how you can extend the result you proved in the previous question to apply to any Array A, now simply assuming that the array contains no duplicates (you can do it without, but allowing duplicates adds needless complexity here). *(2pts)*

(d) Suppose you plan to run COMPARE once. Given that you don't know anything about the elements of A before doing any comparisons, at this point *any* permutation in P_A could correspond to the sorted solution. Suppose we run COMPARE($A[i], A[j]$) and it returns true. Can we rule out any permutations $p \in P_A$? Characterize precisely the permutations that cannot be sorted (Describe the properties of ps in P_A that tell us that p can't be sorted). Suppose that COMPARE returned false — what permutations can't be sorted now? *(4pts)*

(e) Now let's get to our actual search-for-sorting time complexity argument. We are doing a *worst-case* analysis for *any* comparison-based sorting algorithm. This means that we want to pick the *best* i, j to COMPARE, and we will assume that we have 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). Given this set-up, what is the greatest number of permutations we could possibly hope to remove from the running? *(3pts)*

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

(f) 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 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 use without proof that $\log_2 n! \in \Omega(n \log n)$. [2]

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

[2] To see this, observe that $\log_2 n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) \in \Omega(n \log n)$.

3. **Change-makers** *(20pts)* Suppose you are a cashier making change for a customer, and you have coins with values in the set $C = \{1, 5, 10, 25\}$. You receive k, an integer number of cents, and your goal is to return a list L with **minimum** length such that (1) $\forall c \in L, c \in C$ and $\sum_{c \in L} c = k$. i.e., I need you to find the fewest possible coins such such that the values of those coins add up to k.

   (a) Write pseudocode for a greedy algorithm that does this. *(6pts)*

   (b) Prove that this algorithm is correct, given the problem specification above (sums to k, only uses coins in C, and that no smaller L that solves the problem can exist). *(6pts)*

   (c) Determine the worst-case, big-Θ time complexity of *your* algorithm in part 1. Do this in terms of k. *(3pts)*

   (d) Construct a graph to represent this problem, clearly indicating what the vertices and edges represent. Cast this problem as a graph problem. What kind of canonical graph algorithm should one use? Given your specification of the vertices and edges of the graph, what is the time complexity of solving the problem this way? *(3pts)*

   ****Hint****: Make sure you have a finite number of vertices. We don't want an infinite graph!

   (e) Describe the kind of information about the problem/problem space your solution utilizes that the graph version doesn't. i.e., What is it about the problem that makes it so your bespoke solution is faster than the graph translation? *(2pts)*

   ****Hint****: Try tracing the execution of your algorithm in terms of the graph interpretation. What part of your graph algorithm will you never have to do?

4. **And that's an order!** *(20pts)* You are a space anthropologist (or, if you're Ted Chiang, maybe a linguist!). You have found yourself stranded on an alien planet, but have managed to find a still-intact alien ship. In the time you're spent on this planet, you've gathered bits and pieces about how to use the planet's residents' advanced technology, so you know that all you need to do to travel back home on this ship is to enter a valid launch code. However, you know that you need to enter the symbols in the launch code in *alien-alphabetical* order. Your last task is to decode the rules of alien-alphabetical order from the few scraps of alien-alphabetical text you've gathered.

   Put formally, you have an alphabet of alien symbols Σ and a set of alien-alphabetical strings of letters from that alphabet C (i.e., for $s \in C$, $s = l_1 l_2 \ldots l_{k_s}$ where k_s is the length of the string s). If s is *alien-alphabetical*, that means that for $s = l_1 l_2 \ldots l_{k_s}$, $l_1 \leq_a l_2 \leq_a \cdots \leq_a l_{k_s}$, where $l_i \leq_a l_j$ means that l_i precedes l_j in the order of the alien alphabet. Your goal is to figure out the order of the alien alphabet (i.e., for any two $l_1, l_2 \in \Sigma$, $l_1 \neq l_2$, you want to figure out if $l_1 <_a l_2$ or $l_2 <_a l_1$!)

   For example, if $\Sigma = \{a, b, c, d\}$ and $C = \{ccc, cbb, a, ca, dab, dc\}$, you can deduce that $d \leq_a c \leq_a a \leq_a b$. Sometimes, we don't have enough information to know exactly the right order: there may be multiple orders consistent with our data! For example, if $\Sigma = \{a, b, c, d\}$ and $C = \{ab, dc\}$, we know that $a \leq_a b$ and $d \leq_a c$, but $a \leq_a b \leq_a d \leq_a c$, and $d \leq_a c \leq_a a \leq_a b$, and $a \leq d \leq b \leq c$, etc. are all alien-alphabetical orders that could generate the strings in C!

   (a) Reinterpret this problem as a graph problem: Define the graph's vertices and edges in terms of the problem specification above, and then describe, in graph terminology, what the solution to the problem will look like if it exists. *(6pts)*

(b) Write an algorithm that takes in C and Σ and uses your graph approach from the previous problem to find a valid orderings of the letters in the alphabet. *(4pts)*

(c) Prove that your algorithm returns an ordering that is consistent with C (i.e., that no string in C can disprove the ordering you produce). *(4pts)*

(d) What is the runtime of this algorithm, in terms of $|C|$, $|\Sigma|$, and the maximum length of a string in C, $k = \max_{s \in C} k_s$ *(2pts)*

(e) Suppose you were given a C such that **no** ordering could be consistent with it (i.e., $C = \{ab, ba\}$. What does this correspond to in graph terms? *(4pts)*

3 Another Little Puzzle (1pt)

Suppose there are $2k + 1$ people on an island, k have blue eyes, and k have brown eyes, and one has green eyes. They do not know the color of their own eyes but do know the colors of every other resident of the island's eyes. The only person who can communicate with the others is the one with green eyes. At the end of each day, if a person can be **certain** of the color of their eyes, they must leave the island.

The green eyed islander tells all the other participants that there is at least 1 person with blue eyes. Determine the number of days from this point that it takes for every blue-eyed person on the island to leave, and prove that you are correct.

****Hint****: Proceed by induction.