Homework 3: More Graphs, Divide and Conquer

Due Date: The written parts of this assignment will be due at 11:59 PM on Friday February 2nd. As always, that means that the TAs will grade **every** written problem you submit to gradescope before that deadline.
You will also be able to resubmit two written problems with every assignment. The due date for this week's resubmissions is Monday February 5th.

Collaboration: You are allowed (and encouraged!) to discuss these problems at a high level with others. But, please read the full collaboration policy to ensure you are keeping your discussions at the right level, and remember to cite any collaborators.

Problems to Submit: We will count your 1 best mechanical question and your 3 best long-form questions.
We strongly recommend you skim all the problems in a section before attempting them. While we try to make problems in a given section of approximately equal difficulty, you may find some to be easier than others.

Directions Unless otherwise noted, you are allowed to use algorithms described in class (or prerequisite courses) as though you had library implementations of them. For example, you can say "run the BFS-based 2-coloring algorithm from class on the graph G" or "run the bipartite checking algorithm from lecture 5 on G." We also have a list of data structures and algorithms you can use from 373 here.

Since this is a course about efficient algorithms, algorithms that are slower than necessary are unlikely to get "E" scores, but (unless otherwise noted) we do not care about constant factors. In general, an algorithm that is correct but (mildly) slower than optimal will get an equal or higher score to an algorithm that is fundamentally incorrect, but fast.

Mechanical Problems

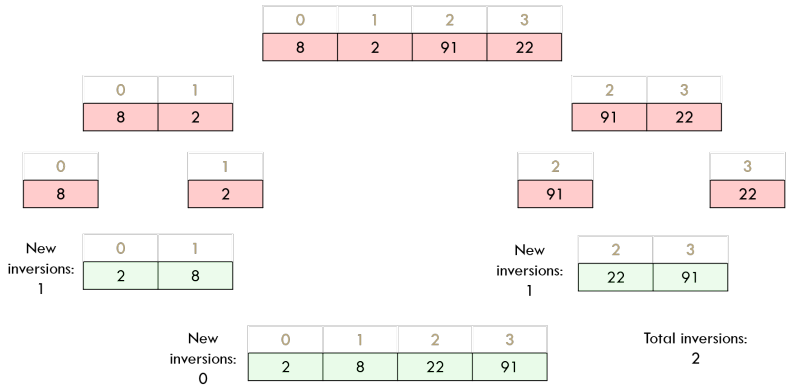
1. Inversions By Hand

Consider the following array

0	1	2	3	4	5	6	7
17	5	1	32	8	10	9	50

Show the execution of the final inversion counting algorithm from class would work on this array. You will show for each call the number of new inversions (that is the number of inversions added in the conquer step, not counting those from the recursive calls). Also include the total number of inversions at the end of the execution. You do not need to show the merge step-by-step.

An example of what we're looking for:



Long-Form Problems

2. Finding Valleys

Let $A[1, \ldots, n]$ be an array of n integers. Call an array a **valley** if there exists an index i such that

- for every j: if $j < i$, then $A[j] > A[j+1]$
- for every j: if $j > i$, then $A[j] < A[j+1]$

Intuitively, the array values decrease to the index i and then increase. Note that an array that is fully sorted **does** count as a valley.

(a) Given an array $A[1, \ldots, n]$ that is a valley, describe an algorithm that finds the valley index i. For example,

- on input $[3, 2, 1, 0]$ you should return 4 (the fourth element is the valley)
- on input $[3, 1, 4]$ you should return 2.

For a score of S or better, your algorithm must run in $\mathcal{O}(\log n)$ time.

(b) Briefly explain why your algorithm works. In your recursion you will "ignore" part of the array, you should at least explain why the index i cannot be there.

(c) Give a recurrence that describes the running time of your code, and give the overall big-$\mathcal{O}$ runningtime of the algorithm.

3. Playing Hooky

In an alternate reality, the 417 staff are deciding where to travel to escape from their weekly duties. Robbie really wants to go to Chicago but he's willing to change his mind, provided that more than $2/3$ of the TAs can agree on a different place. Before starting the discussion, every TA writes up a `Location Proposal` detailing their ideal travel spot.

- If more than $2/3$ of the TAs agree **exactly** on what `place` to visit, then Robbie is out of luck.
- Otherwise, the staff will travel to Chicago.

More formally, you have an array of n `Locations` (called `proposals`). You can call `.equals()` on two `Locations`, but it takes a **long** time. Even worse, there's no `.compareTo()` implemented, nor a `.hashcode()`. So you can't sort these Objects, nor put them in a hash table.

In the case that more than $2/3$ of the TAs submit `equal Location Proposals`, you should return that `Location`. Otherwise, you should return `Location.RobbieFavorite`.

In this problem, you'll describe a divide-and-conquer algorithm that requires $\mathcal{O}(t \cdot n \log n)$ time.

For simplicity, you may assume that the number of elements in the array is a power of 2.

(a) Write pseudocode (or English) for your algorithm. Your algorithm **must** use divide and conquer. There are other (possibly more efficient) algorithms that aren't divide and conquer, but they are not permitted for this question (we want you to practice the new technique).

(b) Prove the following implication: If more than $2/3$ of the TAs agree on a `location`, then in at least one of the two subarrays more than $2/3$ of the TAs agree on a `location`. For simplicity, you may assume the number of elements is a multiple of 6.

(c) Write a recurrence to describe the running time of your algorithm. When analyzing running time, assume that any call to `.equals()` will take t time. You should treat t as a variable in your running-time analysis of this problem (i.e. don't consider it a constant and have it "disappear" in the $\mathcal{O}$-notation). Briefly (1-2 sentences) justify your recurrence. You should convince yourself that the running time will be $\mathcal{O}(t \cdot n \log n)$, but you don't have to include that explanation.

4. Cloning Crystals

In the mystical realm of Technovia, the wise Alchemist has discovered a way to self-clone crystals, starting with a single negatively-charged crystal. The self-cloning can process in one of two ways:

- All currently existing crystals clone into a specified number of copies of themselves (the original being destroyed).
- All currently existing crystals clone into a specified number of copies, with negatively-charged crystals being replaced by positively-charged crystals, and positively-charged crystals being replaced by negatively-charged crystals (the original being destroyed).

The Alchemist has a list of integers stored in an array `stream`. He will feed a (contiguous) subarray of the commands in `stream` to the crystal-cloning machine, representing the number and types of copies of crystals to make. Seeing a positive number k causes all crystals to make k copies of themselves (the first bullet above). Seeing a negative number k causes all crystals to make k copies of their opposites (positive becomes negative, and vice versa).

The land of Technovia is low in negatively-charged crystals, so the Alchemist wants to design an algorithm that will choose a contiguous subarray which will maximize the number of negatively-charged crystals cloned. To do so, he has requested your help.

Below is a sample of the data the Alchemist has provided, and although you notice that it contains both positive and negative numbers, he has requested that you maintain their sign as it is imperative to the top secret process.

Example:

Data stream: [-1, 2, -3, -4, 5]

Expected Output: $120 = (2 \cdot -3 \cdot -4 \cdot 5)$

(a) The Alchemist requests that you submit an initial description/proposal of an algorithm you are planning to implement (in pseudocode). Additionally, **you must use Divide and Conquer** and not any other strategy when processing the data.

(b) The Alchemist is unfortunately very unfamiliar with the way code functions, and has asked you to explain the intuition behind your algorithm at a high level. Specifically, what is each recursive call returning?

(c) The Alchemist also needs you to explain the runtime of your proposed algorithm as he needs to ensure his crystal-cloning machine can handle the speed. Be sure to include both a recurrence and the final running time.

5. Count Those Inversions [coding]

Implement the inversion counting algorithm discussed in class in Java. As with our last coding problem, we will provide a class file and method stub, which you should complete.

Your algorithm must be as efficient in big-$\mathcal{O}$ terms as the last one discussed in class (i.e., $O(n \log n)$ time). In particular, brute force (the $\Theta(n^2)$ algorithm) will receive a grade of U, even though it will likely pass all the test cases on gradescope.

As in class, you may assume that all elements in the array are distinct.

You may wish to start with the pseudocode from lecture.

Remember these tips for autograding:

- Our autograders get very confused by `print` statements we didn't ask for. Be sure to comment those out before uploading.
- You may submit to the autograder as many times as you wish, we will take the score of your last submission.

- A submission that pasts all tests (and meets the efficiency requirement) will get a score of E. A submission that gets at least 8/10 from the autograder, meets the efficiency requirement, but does not pass all tests will get a score of S.
- We will only check the efficiency requirement at the end of the quarter (the autograder will **not** check for efficiency). If you have concerns about whether you are implementing the correct algorithm, please ask in office hours.

Resubmission

You may resubmit two problems from an earlier homework (either HW1 or HW2 or one from each). When you do, remember to fill out the form on the assignments page so we know which problem you submitted.