Homework 3: More Graphs, Divide and Conquer

Version 2: Oct. 28. We made updates to the prompt of problem 2: adding a simplifying assumption and clarifying the requirements for part (b).

Due Date: The written parts of this assignment will be due at 11:59 PM on Friday October 28. 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 October 31st.
The coding problems may be submitted at any time through the last day of classes (Friday Dec. 9), but we strongly recommend you finish any you intend to do by the due date for the written assignment so you have time to do next week's problems next week.
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

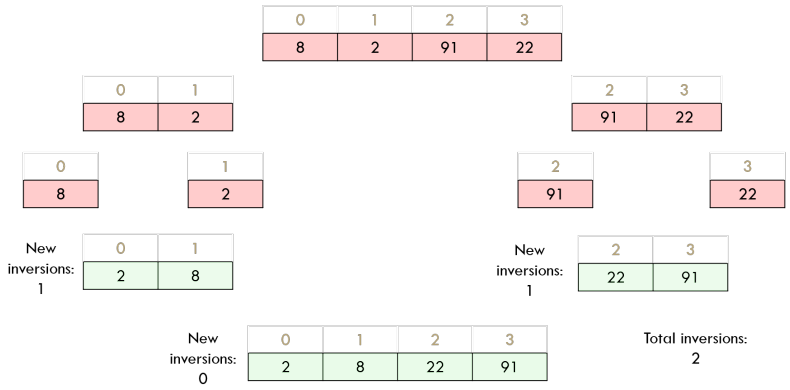
1. Inversions By Hand

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. Network Resilience [long-form, written]

This problem is a last "graph modeling" problem left over from last week. You should not need to divide and conquer on this problem.

You are tasked with managing a network of n servers, each of which are of questionable quality due to cost-cutting measures. Some servers have a direct link between each other, but not every pair of servers necessarily has a direct link between them. However, two servers that aren't directly linked can still communicate with each other by repeatedly passing along information to servers that they are directly linked until the info makes its way to the other server. Communication between servers is two-way, so server A can communicate with server B if and only if server B can communicate with server A.

We'd like to choose one of the servers to be the primary server that manages the rest of the servers, but we're concerned about what happens if that primary server goes down. Fortunately, these servers have a feature where in the event of a crash, they can transmit the recovery data to one and *only one* other server that they're directly linked, after which they turn into an expensive (but actually not that expensive) paperweight. From there, that other server can pass on the recovery data to all of the servers that its directly linked, each of which can in turn continue to pass on the data to any servers that they were directly linked to. You may assume that in the initial network, there are always sufficient intermediate links to enable communication between *any* two servers in the network, even if they don't have a direct link between them.

We say that a safe recovery can be made if **all** of the remaining $n - 1$ servers can receive the recovery data when the primary server crashes.

(a) If we think of the network with the primary server and without the primary server as being represented by two different graphs, what property does the graph corresponding to the original network (with the primary server) have? For the graph corresponding to the network without the primary server, what kind of property do we need this graph to have in order to make a safe recovery?

(b) Come up with a modification of BFS that will help us to choose a primary server that enables a safe recovery. If there are multiple valid choices of primary server that would enable a safe recovery, your algorithm only needs to provide one of them.

(c) Argue that we can make a safe recovery if we choose the primary server to be the output of the algorithm described in part (b).

Hint: it may be helpful to characterize safe recovery in terms of the ability to have two servers that were previously connected to the primary server (before it broke) communicate with each other even after the primary server goes offline.

3. Divide and Sauerkraut

The 417 staff realized that their true calling is restauranteering and are deciding what kind of restaurant to start. Robbie really wants to start a sauerkraut restaurant, but he's willing to change his mind, provided that more than $2/3$ of the TAs can agree on a better idea. Before starting the discussion, every TA writes up a `Restaurant Proposal` detailing their ideal restaurant.

- If more than $2/3$ of the TAs agree **exactly** on what `Restaurant` to start, then that is the option they will pursue.
- Otherwise, the default option of sauerkraut restaurant will be pursued.

More formally, you have an array of n `Restaurants` (called `proposals`). Since `Restaurants` are quite complicated[1], you can call `.equals()` on them, 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 Restaurant Proposals`, you should return that `Restaurant`. Otherwise, you should return `Restaurant.SauerkrautRestaurant`.

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 efficient algorithms that don'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 `restaurant`, then in at least one of the two subarrays more than $2/3$ of the TAs agree on a `restaurant`. 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. The Evil League of Evil

Dr. Horrible has found a way to develop rapidly self-cloning bots to take over every server in the world. He starts with one evil bot. The self-cloning can proceed in one of two ways:

- All currently existing bots clone into a specified number of copies of themselves (the original being destroyed).
- All currently existing bots clone into a specified number of copies, with evil bots being replaced by good clones, and good clones being replaced by evil clones (the original being destroyed).

Dr. Horrible has a list of integers stored in an array `stream`. You will feed a (contiguous) subarray of the commands in `stream` to the bots, representing the number and types of copies of themselves to make. Seeing a positive number k causes all of them to make k copies of themselves (the first bullet above). Seeing a negative number k causes all of them to make k copies of their opposites (good become evil, evil good).

The Evil League of Evil has required Dr. Horrible to use a different `stream` array they give him each month. Dr. Horrible wants an algorithm that will choose a contiguous subarray to feed to one (evil) bot which will maximize the number of (evil) clones created upon feeding them that subarray.

In order to complete this mission, he has requested your help.

[1] There are so many themes! And menu items. And tchotchkes.

He has provided a sample of the data the Evil League 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.

Dr. Horrible's Example:

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

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

(a) Dr. Horrible requests that you submit an initial description/proposal of an algorithm you are planning to implement (in pseudocode). Additionally, at the top of the top secret file, it's written that as part of the Evil League's policies, **you must use Divide and Conquer** and not any other strategy when processing their data.

(b) Dr. Horrible is unfortunately very unfamiliar with the way code functions, and has asked you to explain the runtime of your proposed algorithm as he needs to ensure his primary Device of Evil can handle the speed.

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.