1. Revision

Select a question from any of your previous problem sets and do a full revision. It should be a full write-up question. Correctness/optimality is not the focus of this task. For example, if you designed an \( O(n^2) \) algorithm and you now know that an \( O(n \log n) \) solution is possible, you should still rewrite your original \( O(n^2) \) algorithm. But of course, if you had factual mistakes, such as mathematical ones in the time analysis, you should correct them. On the other hand, clarity and conciseness are. Here are some required revision tasks that you should perform:

1. Identify at least one lemma and restructure your proof around it. You may and probably should add additional ones. If your chosen example already has many lemmas pulled out, then you should pick a different one. If all of your proofs have plenty of lemmas already, then congratulations, you may skip this one.

2. Cut 25% to 50% out from what you wrote. It may sound drastic, but trust that there is room to cut. You may want to first go over the list below, as reductions are a natural result for many.

3. Attach a copy of your original writing.

4. Start with a short paragraph summarizing the main revisions that you undertook.

As you revise, please pay attention to address the following:

- Unless the algorithm is really simple, the description should not be a repeat of what’s in your pseudo code. You now have enough experience with algorithm design to know that every algorithm hinges on a crucial step/idea/invariant. This is the place to focus on that key point. You want to give the reader a big picture of how it generally works before delving into the details.

- Avoid too many low-level details even in the pseudo code section. Trust that the readers are competent programmers who can implement the lowe-level structures if they understand what needs to happen. There is hardly ever need for full pointer notations or complex index arithmetics - if you find yourself writing complex index arithmetics, you should instead explain the purposes of the jumping around in the array resulting from the index manipulations.

- Time analysis must include data structure justifications. Graph-based algorithms that call for graph construction must clearly state/explain the combinatorics of the constructed graph, i.e. number of nodes and edges.

- Greedy algorithms must have optimality proof.

- Try to go beyond superficial revisions that are essentially wordsmithing. If changing words is all that is needed, then your original writing didn’t really have clarity or conciseness issues. Most of the time, structural revisions are needed that involve pulling out lemmas and reordering or removing paragraphs. Do not be afraid to attempt it.
In addition, here are some more mechanical revisions:

- For each sentence, find the part of the sentence that links to the previous sentence. Is it at the beginning or the end? It should be at the beginning. If no part of the sentence has anything to do with the previous one, then you have a bigger problem.
- For each paragraph, summarize the main point. Is that main point spelt out anywhere? Make sure each sentence in the paragraph supports the main point you want to make. Check to see if the first and the last sentences are on the same point.
- Give your writing to an intelligent outside reader, and ask her where she gets lost.
- Look for words or phrases that add little and get rid of them.

2. Solve the following recurrence relations; i.e. express each one as \( T(n) = O(f(n)) \) for the tightest possible function \( f(n) \), and give a short justification (some are longer than others, but full inductions are not required). Unless otherwise stated, assume \( T(1) = 1 \).

1. \( T(n) = 4T(n/2) + n^2 \)
2. \( T(n) = 3T(n/4) + \sqrt{n} \)
3. \( T(n) = 7T(n/3) + n^3 \)
4. \( T(n) = 2T(n/2) + n\log{n} \)
5. \( T(n) = 2T(\sqrt{n}) + 1 \), where \( T(2) = 1 \)

2. Full write-up. Haverford discovered in late August that some fake OneCards had been mixed in with the new ones they were planning to give the incoming freshmen. Security knew that strictly less than half of them were fake and that the fake cards had incorrect access codes, while all the real cards had the same, correct, access code. They had a machine that could check if two OneCards had the same access code (but the machine couldn't say what the code was). Design an \( O(n\log{n}) \) divide-and-conquer algorithm that would have helped Security to quickly find a single real OneCard to give a freshman arriving for an ID. Extra credit: find an \( O(n) \) solution. It is ok if your extra credit solution is not divide and conquer.

3. Full write-up. A typical representation of a grayscale image is an \( n \times n \) 2D array, where each \([i,j]\) index stores a pixel value between 0 and 255. In image processing, it is often the case that we would like to find those pixels that are brighter than all the surrounding pixels. This has applications in feature detection and many other filtering techniques. Design a divide-and-conquer algorithm that finds any one such pixel in \( O(n\log{n}) \) time. You may limit the neighbors to the 4 edge neighbors. Extra credit: find an \( O(n) \) solution.

Please hand in your assignment in class.