CSE101: Basic data structures/algorithms test
Never forget brute force

1 Instructions

1. You will get two questions, each on a different paper.
2. You will not get any feedback during the test; you cannot ask the TA/tutors whether your solution is correct. This is a test.
3. The TAs/tutors will not answer any questions about the questions, to ensure fairness (otherwise, some TAs might give hints that other TAs will not).
4. For each question, explain the algorithm using clearly written pseudocode. Some guidelines for pseudocode:
   (a) Number each line of pseudocode. Clearly demarcate loops with indentation. For clarity, also demarcate loops either using parentheses (like C/C++) or numbering with Roman numerals, letters, etc., to differentiate from the main pseudocode.
   (b) You do not need to write pseudocode for any algorithm/data structure taught in class. For example, you can write: "Run the merge algorithms on sorted arrays A and B", or "Insert key into the AVL tree T". You do not need to explain these further.
   (c) Try to avoid lines of pseudocode that itself involves complicated operations. Do not write: "find the median in list of numbers.". How? Linear search? Binary search? Putting in a fancy data structure? Explain how this would be done in pseudocode.
   (d) It's ok to get indexing wrong and have off-by-one errors.
   (e) For any question asking you to construct a data structure: clearly write pseudocode for all the desired operations.
5. For each question, you must give a running time analysis, in terms of big-Oh. For maximum clarity, it is best to write out the big-Oh running time of each Step of your algorithm (which is conveniently numbered!). If you have used some algorithm taught in class, you can simply state the running time. For example: "Mergesort runs in $O(n \log n)$, as explained in class."
6. Be careful about the input parameters. Sometimes the input size is stated in terms of n and something else (for example, there may be k arrays of size n, so the input size is nk). It is best to state what n, k, etc. are, if they are not specified in the problem.

7. If you do not write clearly, then there will be some subjectivity in grading. The TAs and tutors will make a best effort in understanding your solution, but you may lose points if you are not clear.
8. Each question is worth 5 points, which is split up as follows. If you give a correct brute-force algorithm, you get three points. If you write the running time correctly, you will get two more points. If your algorithm beats brute-force, you will get 0.5 point extra credit.
9. You are free to first write the brute-force solution, and then write the improvement.
10. Some questions may say: "for full credit, using $O(\log n)$ storage" or "for full credit, do not use a hash table" or "for full credit, give an $O(n^2)$ algorithm". In this case, for full credit, your algorithm needs to respect these conditions. If your algorithm does not respect these conditions (but it is still correct), you will lose 2 points.
11. Using $O(\log n)$ (or $O(1)$) storage means that you can only store a constant number of pointers and potentially perform recursion, with at most $O(\log n)$ recursive calls (since that is the maximum stack memory). This means you cannot build a hash table or allocate an empty array to copy elements into.
12. As long as your algorithm is correct, the running time analysis is correct, and the memory or time conditions are respected, you will get full credit. If your algorithm and running time analysis are correct, but the memory/time conditions are not respected, you get 4 points (for that question). For almost all problems, a simple brute force algorithm with nested for-loops will suffice. It is very, very easy to do well in this test.
13. If there are some ambiguities, feel free to make reasonable decisions (but explain them!). For example, the question might not specify how to output (either print, or output as an object). Make reasonable choices, and as long as you get the algorithm correct, you will get credit. You are free to define objects that might be convenient, such as pairs or triples of integers. (You don't need to write out code for them.)