Name: ________________________

Computer Science II — Fall 2021

Final Exam

This exam should have six pages. Closed book and notes.
No calculators or computers allowed.

Problem 1: [15 points]

Below is some modestly mysterious map code. Describe *briefly* in English what the `mystery` method does. What would be a good, descriptive name for this method?

```
public Map mystery(Map m) {
    Map x = new HashMap();
    for(Object a : m.keySet()) {
        Object b = m.get(a);
        x.put(b, a);
    }
    return x;
}
```

Problem 2: [20 points]

On the final assignment, you had to find some way to store the `Expression` objects corresponding to the formulas in the spreadsheet, in addition to a map containing the values of the cells. Some groups chose to use a 2D array to hold the Expressions and others used a Map that mapped cell names (e.g. “B3”) to their corresponding Expressions. Below, briefly discuss the tradeoffs between the two approaches with respect to performance, memory requirements, and any other implementation aspects you think are relevant. In your discussion, try to identify at least one positive aspect of each approach.

Problem 3: [25 points]

Part a) Below, draw the Binary Search Tree (BST) that would result from inserting the following sequence of values into an initially-empty BST: 50, 30, 80, 40, 45, 35, 90, 60, 85. (That is, first insert a 50 into an empty tree, then insert the 30 into the tree, etc.)

Part b) Draw the BST that would result from removing the 30 and 45 from the tree above.

Part c) Is it possible to create a *perfect* BST that contains the values from Part b? If so, draw the tree. If not, explain why not.

Problem 4: [20 Points]

Some sorting algorithms stop as soon as the list is sorted, which is an advantage when sorting almost-sorted arrays (those in which only a few values are out of place). Selection Sort isn't one of those, but we could add code to make it stop early: Consider a version of Selection Sort where, on each iteration, we made an *extra* pass over the *unsorted* portion of the array to see if it was already ordered. If so the algorithm stops, otherwise the sorting step would proceed as normal.

Part a) If we made the proposed modification, what is the *best case* Big-O complexity of our new sorting algorithm? For full credit, explain your analysis briefly, and the scenario that would produce this best-case behavior.

Part b) If we made the proposed modification, what is the *worst case* Big-O complexity of our new sorting algorithm? For full credit, explain your analysis briefly, and the scenario that would produce this worst-case behavior.

Problem 5: [40 Points]

You've been hired by Bradco, a local company that's creating an online dictionary. As a starting point, they've decided to grab text off of the web and build a collection of the *unique* words appearing in the text (much like we did in our word-counting lab). Bradco's software engineers have proposed a variety of ways of finding and *printing* the unique words from the full collection, and want your help in deciding which is the best approach. (Assume that there are N words grabbed from the web, containing duplicates, stored in a very large array of strings.) Give a Big-O estimate of the run time for each of the approaches below. For full credit, explain your answers briefly.

Part a) For each position *i* in the array of words, start a *new* traversal from the *beginning* of the array, looking for duplicates. If you find a duplicate (a word matching the word at position *i*) before reaching position *i*, increment *i* without printing and continue. (No need to print – we must have printed the word at position *i* already.) Otherwise, print the word at position *i* and then increment *i* and continue.

Part b) Sort the words into alphabetical order using a fast sorting algorithm. Since duplicates will then be adjacent, we can make a pass through the sorted words, ignoring over any words that are identical to the preceding word and printing the others.

Part c) Build a Binary Search Tree containing the words by inserting them one after another. During each insertion, check to see if the current word is already present. If not, print the current word before adding it to the tree.

Part d) Use a HashSet to contain the unique words: For each word from the array, check to see if it's in the set. If not, print the word and add it to the set, otherwise skip to the next word in the array.