CS 11 Spring 2010 — Mid-term #2 **Solutions**

1. (25 points) [QUESTION:] Consider the problem of verifying that a given n-element array contains one of every value from 1 through n. For example, if $n = 5$, then the following array should pass this verification because each value from 1 to 5 inclusive appears exactly once in the 5-element array:

```
entry number:  0  1  2  3  4
       value:  5  2  3  4  1
```

Now consider the following two methods that implement different algorithms for performing this verification:

```
public static boolean isValidVersionA (int[] x) {

    boolean[] observed = new boolean[x.length];

    for (int i = 0; i < x.length; i++) {

        if ((x[i] < 1) || (x[i] > x.length) || (observed[x[i] - 1])) {
            return false;
        }

        observed[x[i] - 1] = true;

    }

    return true;

}

public static boolean isValidVersionB (int[] x) {

    for (int i = 0; i < x.length; i++) {
        for (int j = i + 1; j < x.length; j++) {
            if ((x[i] < 1) || (x[i] > x.length) || (x[i] == x[j])) {
                return false;
            }
        }
    }

    return true;

}
```

The question: As n becomes arbitrarily large, which verifier is going to run faster (e.g., perform fewer operations), `isValidVersionA` or `isValidVersionB`? Can you state a rough estimate (e.g., a *Big-O* expression) for the number of operations each verifier would perform as a function of n?

[Answer:] `isValidVersionA` will require fewer operations than `isValidVersionB`. Specifically, version A visits each element of `x` once, controlled by a aingle loop, thus implying $O(n)$ operations, where $n = x.length$. In contrast, in version B, for each value, compares it to every subsequent value in the array, thus comparing each value to, on average $\frac{n}{2}$ other elements. Therefore, $O(n^2)$ operations are performed by version B.

[Discussion:] Your answers were quite varied. Many ran the risk of providing answers with no justification. If you answered correctly, then all was well, but if any part of your answers were incorrect, then you lost more points for lack of explanation. Assigning version A into the category of $O(n^2)$ and version B into $O(n^3)$ was common, yielding the correct overall conclusion, but with a failure to count the operations correctly. In fact, a even for those who presented completely correct answers, many commonly asserted that *there were n* ***passes*** *through the array* for version A. It was evident, from context, that the intended statement was that *there were n* ***operations performed in the single pass*** *through the array.* Similarly, many people erroneously referred to *if-loops*—a term that has never escaped my lips during lecture. However, I think this type of terminological sloppiness allowed some of you to mislead yourselves in calculating total operations.
Many others pulled big-O running times from thin air, claiming everything from $O(nlgn)$ for one of the algorithms implemented, to a stunning $O(3n!)$. These reveal some fundamental misunderstanding about the operations performed and how to count them.

2. (25 points) [QUESTION:] Write a method that takes as parameters two pointers to arrays of `int` and compares them for *multiset equality*: that is, the two arrays are considered equal if they contain the same number of each value. The order in which the values appear is irrelevant; what matters is that for each value in one array, there must be a unique, equal value in the other array. For example, the following two arrays are equal multisets:

```
3 9 8 2 9 2 4 7
9 9 7 2 2 3 8 4
```

[ANSWER:]

```
public static boolean multisetEqual (int[] x, int[] y) {

  if (x.length != y.length) {
    return false;
  }

  boolean[] matched = new boolean[x.length];

  for (int i = 0; i < x.length; i++) {
    boolean found = false;
    for (int j = 0; (j < y.length) && (!found); j++) {
      if ((x[i] == y[j]) && (!matched[j])) {
        matched[j] = true;
        found = true;
      }
    }
    if (!found) {
      return false;
    }
  }

  return true;

}
```

[DISCUSSION:] There are many ways to solve this problem, but the crux of it is matching equal values from different positions in the two arrays and then somehow "remembering" that matching so that neither value can be used for matching to some other position in the other array. There were a number of common errors, so I'll enumerate them here:

(a) **No marking:** Although many attempted to compare the values of one array to all of the values of the other array, there was no attempt to *mark* used entries of these arrays to avoid incorrect multiple matchings. Each entry should match to some value in the other array no more than once.

(b) **Marking with `null`:** If you want to *mark* a particular entry as having been used for a match so that it is not considered again, you may not write something like `y[i] = null`. Since `y[i]` is a space that holds an `int`, and `null` is a pointer value, and **not** a valid `int` value.

(c) **Marking with `0`:** Instead, you may want to use something like `y[i] = 0` or `y[i] = -1` to mark an entry as *used*. While such an assignment is a correct use of data types, it is not a valid solution to the problem. Specifically, `0`, `1`, and any other `int` value you may conjure are legal values that may "naturally" appear in these arrays. Using one of those values as a marker—that is, as bookkeeping information, or *metadata*—creates an ambiguity that will bring about incorrect results for some inputs.

(d) **Premature returns:** Often, a `return` statement, sometimes `true` and sometimes `false`, was placed within loops that would need to continue for the algorithm to complete its work.

(e) **Sort-and-compare:** Many came to the strategy of sorting the arrays and then performing a more straightforward, position-by-position comparison. This strategy is legitimate, but some lost points if their sorting method altered the original arrays—something a caller of this method would not expect.

Sadly, for some, the solution attempted was so badly mangled that the intent of the algorithm was unclear. Those received little credit, although some would be earned if a strategy emerged.

3. (25 points) [QUESTION:] Consider sorting an array of values by using the following strategy: Search the array for the lowest value, and then swap that value with the value at index 0; then search the array for the second lowest value, and swap *that* with the value at index 1; search for the third lowest, and swap it into position 2; and so on. This approach to sorting is known as *selection sort*.

The question:

(a) Write a method that performs *selection sort* on an array of `int` obtained as a parameter.

(b) Was your method *in-place* or *out-of-place*?

(c) What is the *big-O* number of operations of your method?

[ANSWER:]

(a)
```
public static void sort (int[] x) {

  for (int i = 0; i < x.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < x.length; j++) {
      if (x[j] < x[minIndex]) {
        minIndex = j;
      }
    }
    int temp = x[i];
    x[i] = x[minIndex];
    x[minIndex] = temp;
  }

}
```

(b) *In-place:* The values are moved within the array, using only a single extra `int` space to facilitate swapping. An *out-of-place* algorithm would create another array equal to (or, at least, proportional to) the length of the array being sorted.

(c) If $n = x.length$, then each of the n entries of the array is eventually used to swap the smallest remaining value into it. For each such position, all of the positions beyond that one are searched for the smallest remaining value—on average, $\frac{n}{2}$ such position must be searched. Therefore, $O(n^2)$ operations are performed.

[DISCUSSION:] There were so many ways to err with this question, so I will stick to the big ones. The most common error was to begin the inner-loop search for the next smallest value by setting a counter to 0 (or sometimes 1). Doing so will bring the entire array under consideration, **including the smaller values already selected** to be brought to the front of the array. Thus, the same, overall minimum value will be found over and over again, and moved into the next slot of the array (as traversed by the outer loop).
Another common error, again inside the inner-loop search for the next smallest value, was to compare not each item with the minimum value found so far, but rather to

compare pairs of adjacent values. This bubblesort-like approach simply doesn't work in finding the minimum among the remaining, unselected values.
Most correctly labeled their algorithm as being *in-place*, they then proceeded to stick their foot in it, providing reasons that have nothing to do with the *in-place/out-of-place* designation, costing themselves points.
Many marked their algorithms as requiring $O(n^2)$ operations. Unfortunately, although a properly written selection sort does indeed require that number of operations, some of you wrote algorithms that would have performed $O(n)$ or $O(n^3)$ operations, yet still labeled your own as require $O(n^2)$. The question, of course, asks for the number of operations perfomed by **your** method.

4. (25 points) [QUESTION:] Consider the game *simple-sudoku*: a dumbed-down version of the real *sudoku*. This game is played on a 9 x 9 grid of integers whose values are between 1 and 9. A solved simple-sudoku grid contains one of each value (1 to 9) in each row and in each column. The game begins with only a few values, scattered around the grid, and the player must fill in the remaining values to construct a solution.

The question: Write a method named `testGrid` that accepts a pointer to a simple sudoku grid, represented as a two-dimensional array of `int`, and then tests that grid to determine if it is a correctly solved simple-sudoku grid. That is, your method must return `true` if the 2-D array is of the correct size, each row contains the values 1 to 9, and each column contains the values 1 to 9; it must return `false` otherwise.

Hint: It may be useful to write one or more supporting methods that `testGrid` calls to perform repeated tasks.

[ANSWER:]

```
public static boolean testGrid (int[][] board) {

  if (board.length != 9) {
    return false;
  }

  for (int i = 0; i < 9; i++) {
    if (board[i].length != 9) {
      return false;
    }
    if (!testLine(board[i]) || !testLine(extract2nd(board, i))) {
      return false;
    }
  }

  return true;

}

public static boolean testLine (int[] line) {

  boolean[] found = new boolean[9];
  for (int i = 0; i < line.length; i++) {
    if (found[line[i] - 1]) {
      return false;
    }
    found[line[i] - 1] = true;
  }
  return true;

}
```

```
public static int[] extract2nd (int[][] board, int x) {

  int[] line = new int[board.length];
  for (int i = 0; i < board.length; i++) {
    line[i] = board[i][x];
  }
  return line;

}
```

[Discussion:] First, I stated during the exam that checking `board[0].length` was sufficient for testing the length of the second dimension, so nobody lost points for doing it. However, the better solution, shown above, is to test the lengths of **every** second dimension array.
A wide variety of other mistakes occured, often including syntactic misuse of the two-dimensional array so severe that I could not discern the intent. Many people wrote methods that would test the contents of the rows but not the columns (or *vice versa*). Some took the approach of summing the values in a row or column, observing that $\sum_{i=1}^{9} = 45$ and thus testing against that value. Unfortunately, unless the test was otherwise cleverly devised, this test is insufficient.[1] And, of course, the most common problem was the expiration of time, leaving people scribbling madly to get **something** down on this last question.

[1]Do you see why?