CS 11 Spring 2008 — Mid-term exam 2
Answer Key

1. [Question:] (20 points) Consider the following program:

```
class Foo {

    public static void main (String[] args) {

        double[] d = new double[Integer.parseInt(args[0])];
        for (int i = 0; i < d.length; i++) {
            d[i] = -i;
        }

        String s = new String("zippitydodah");
        if (s == (args[1])) {
            baz(d, true);
        } else {
            baz(d, false);
        }

        System.out.println("Length: " + d.length);
        for (int i = 0; i < d.length; i++) {
            System.out.println("d[" + i + "] = " + d[i]);
        }

    } // main

    public static void baz (double[] d, boolean x) {

        if (x) {
            d = new double[d.length / 2];
        }

        for (int i = 0; i < d.length; i++) {
            d[i] = i * 2;
        }

    } // baz

} // class Foo
```

The question: What is the output of this program if it is invoked like this?

```
java Foo 4 zippitydodah
```

[Answer:] Taken from an actual run of the code:

```
Length: 4
d[0] = 0.0
d[1] = 2.0
d[2] = 4.0
d[3] = 6.0
```

[Discussion:] This code requires that you understand how arrays and pointers are passed to methods, as well as how `String`s are compared to one another. The key observations are:

(a) The comparison `(s == (args[1]))` will always yield `false` as a result. `s` point to a **new** `String` object, and thus `args[1]` cannot point to the same object. Since the equality comparitor (`==`) only compares pointers, and these two pointers must point to different objects, then these two pointers cannot be equal.

(b) Since the above comparison is `false`. the call to `baz` does not create a new, half-sized array within that method. Critically, though, even if a new array is created, that array is never passed back to the caller. That is, there is no way for the pointer `d` in `main` to point to the array created is `baz`.

(c) Given that `baz` does not create a new, half-sized array, any changes that it makes to the values through its pointer `d` **will** be visible to the caller, `main`. That is, the array into which `baz` assigns values is the same array to which `main` passed a pointer—they share that array.

2. [QUESTION:] (20 points) 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:] For a large n, `isValidVersionA` is going to run faster. Because `isValidVersionA` visits each value in the array `once`, comparing it to an array of `boolean` that indicates which values has been seen (and which haven't), it requires $O(n)$ operations. In contrast, `isValidVersionB` compares every element to every other element, looking for duplicates. Thus, every pairing of elements requires approximately $\frac{n(n-1)}{2}$ comparison, which implies $O(n^2)$ operations.

[DISCUSSION:] Many people provided reasonable approximations for the running times of both algorithms. I ignored whether the constants were correct, and I ignored whether you wrote the Big-O expression using the correct form. I was most concerned that you noticed that algorithm A required roughly n operations, and algorithm B required roughly n^2.
Many people simply provided incorrect estimates for one or both algorithms. (Note, an algorithm with a running time of $O(n^n)$ is going to take a **long** time.) Many seemed to use $O(n\ log_2\ n)$ simply because we had discussed it, and not with any particular reasoning or calculation behind it.

3. [QUESTION:] (30 points) 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:] We use the `isValidVersionA` method from the previous question.

```
private static boolean testGrid (int[][] grid) {

    // Check the dimensions.
    if (grid.length != 9) {
        return false;
    }
    for (int row = 0; row < grid.length; row++) {
        if (grid[row].length != 9) {
            return false;
        }
    }

    // Check each row.
    for (int row = 0; row < 9; row++) {
        int[] rowValues = grid[row];
        if (!isValidVersionA(rowValues)) {
            return false;
        }
    }

    // Check each column.  Here we  make a new array to hold column
    // values, since they are spread across the 2nd dimenional arrays.
    for (int column = 0; column < 9; column++) {
        int[] columnValues = getColumnValues(grid, column);
        if (!isValidVersionA(columnValues)) {
            return false;
        }
    }

    // Passed all of the checks!
    return true;

}

private static int[] getColumnValues (int[][] grid, int column) {

    int[] columnValues = new int[9];
    for (int row = 0; row < 9; row++) {
        columnValues[row] = grid[row][column];
    }

    return columnValues;

}
```

[Discussion:] There are many ways to skin this cat (and even more ways not to). The most common error was not to check the lengths of the arrays and verify the size of the grid. Syntactic errors abounded, but I was as permissive as I could manage—if I could understand what you **meant**, I might have marked it, but I accepted it.
On the more semantic side, many people did not get the sequence of checks right, returning either `false` or `true` too soon. Many forgot to reset critical arrays used for checking at the right moments. Others checked rows but not columns.

4. [QUESTION:] (30 points) Recall the future stock profit problem from lab-5. You are given an array of that contains the prices at which one particular stock will trade for the next n days. You may only buy and sell the stock *once*, and you may not sell before you buy (no shorting the stock, if you know what that is). Thus, your goal is to maximize your profit from that purchase and sale of the stock by choosing your days wisely.

   Futhermore, consider that you have at your disposal a set of methods that have already been written, and that you use in code that you write to solve this problem. They are:

   - `public static int findMin (int[] values, int begin, int end)`
     Find the smallest value from the array `values` starting with the entry number `begin` and ending with the entry number `end` (inclusive). Return the entry number at which this minimal value is found.
   - `public static int findMax (int[] values, int begin, int end)`
     Find the largest value from the array `values` starting with the netry number `begin` and ending with the entry number `end` (inclusive). Return the entry number at which this maximal value is found.
   - `public static int[] selectBest (int[] values, int[] entries1, int[] entries2, int[] entries3)`
     The pointer `values` points to an array of integers. Each of `entries1`, `entries2`, and `entries3` points to a 2-element array of entry numbers within `values`. We define the following differences:
     - `difference1 = values[entries1[1]] - values[entries1[0]]`
     - `difference2 = values[entries2[1]] - values[entries2[0]]`
     - `difference3 = values[entries3[1]] - values[entries3[0]]`

     This method returns a pointer to `entries1` if `difference1` is the largest of the three differences; it returns `entries2` if `differences2` is the largest of the three; and it returns `entries3` if `differences3` is the largest.

   The question: Write a **recursive method** that employs a **divide and conquer approach** to solving this problem. The method must accept, as a parameter, the array of stock prices, and it must return a 2-element array of `int` that specifies both the day on which to buy and the day on which to sell the stock in order to maximize profit. *The method may also require other parameters if you so choose.*

[Answer:] Using the methods provided in the question makes the solution reasonably short:

```
public static int[] calculateMaxProfit (int[] prices,
                                        int lowDay,
                                        int highDay) {

    // The base case.
    if (lowDay == highDay) {
        int[] result = { lowDay, highDay };
        return result;
    }

    // Recursively find the best solutions for the left and right halves.
    int midDay = (lowDay + highDay) / 2;
    int[] leftResult = calculateMaxProfit(prices, lowDay, midDay);
    int[] rightResult = calculateMaxProfit(prices, midDay + 1, highDay);

    // Find the best solution that crosses the halves.
    int[] acrossResult = { findMin(prices, lowDay, midDay),
                           findMax(prices, midDay + 1, highDay) };

    // Return the best of the three.
    return selectBest(prices, leftResult, rightResult, acrossResult);

}
```

[Discussion:] There were two major variations on this solution. The first skipped the use of `findMin` and `findMax`, instead having the recursive method also find the minimum and maximum values and returning a 4-valued response including these values. That solution works just fine if done correctly.

The second major variation was similar to the one above, but actually performed the work of creating new, half-sized arrays, copying the appropriate values into them, and recurring on those. The trick with this approach is getting the sizing of the arrays right, and even worse, compensating for the change in index numbers for the right half. Many such solutions failed to account for these details, and lost a bit of credit as a consequence.

Most other attempts failed to employ recursion as all. A common error was to find the minimum and maximum values within each half and consider that the solution for the maximum profit in those halves, failing to actually recur on the halves and do a proper search for their solution. Other answers employed loops, interfering with the recursion's progression.