Introduction to Computer Science I
Fall 2012
Mid-term exam solutions

1. Question: (20 points) Consider the following recursive method:

```
public static long fib (int n) {

    System.out.println("Begin: " + n);
    long result;
    if (n < 2) {
        result = 1;
    } else {
        result = fib(n - 2) + fib(n - 1);
    }

    System.out.println("End: " + n + " is " + result);
    return result;

}
```

If some method (e.g., `main()`) were to call this one, passing an argument of 4, **what output would this method generate?**

Answer:

```
Begin: 4
Begin: 2
Begin: 0
End: 0 is 1
Begin: 1
End: 1 is 1
End: 2 is 2
Begin: 3
Begin: 1
End: 1 is 1
Begin: 2
Begin: 0
End: 0 is 1
Begin: 1
End: 1 is 1
```

```
End: 2 is 2
End: 3 is 3
End: 4 is 5
```

Discussion: The number of people who answered this question correctly, or even recognized the need for output to be generated with each recursive call, was disappointingly few.[1] Many answers consisted solely of the first and last lines of output; of those, a significant fraction also showed, in various forms, that the calculation would be performed recursively. Unfortunately, few of those who showed the recursion noticed that the output would be generated with each such call.

A smaller number of people ignored or failed to notice the recursion at all. Others did not understand that *output* refers to the text emitted by the calls to `System.out.println()`, and not to the value returned by the method. A few simply did not read the question carefully, and failed to show or address either the printed output or the returned value.

Given the answers to this question, the order in which recursive calls proceed requires additional attention during class.

[1]On whom the disappointment rests is intentionally unspecified.

2. Question: (20 points) Remember the lab in which you wrote a program that would print a variety of pretty patterns as ASCII art? Let's add to that collection of patterns. Specifically, we want to add `printUpsideDownTriangle()` as a method. Like the others it has one parameter, *size*, that indicates how many rows the triangle has. A triangle of size 5 should look like this:

```
*****
 ****
  ***
   **
    *
```

Write this method to print this pattern of any given size.

Answer:

```
public static void printUpsideDownTriangle (int size) {
  for (int row = size; row > 0; row = row - 1) {
    int spaces = size - row;
    for (int space = 0; space < spaces; space = space + 1) {
      System.out.print(' ');
    }
    int stars = row;
    for (int star = 0; star < stars; star = star + 1) {
      System.out.print('*');
    }
    System.out.println();
  }
}
```

Discussion: A number of errors in constructing the loops were common. Often, one of the two inner loops were either forgotten or were themselves nested (making the innermost one a third-level of depth). At a greater level of detail, people often miscalculated the number of spaces and stars to print, at times printing a constant number of one of the other (usually stars, for some reason), and not having those numbers change with the row number.
Many people did not read the question carefully, writing code to prompt the user for the triangle's size and taking that value from the keyboard. As often as not, these steps were performed even for those who properly accepted that same value as a parameter to be passed by the calling method.

3. Question: (30 points) Arrays are useful, and we would like some methods that perform common array tasks. Write the bodies of the following methods that operate on arrays of `int`:

   (a) `public static int[] copy (int[] original)` A method that makes a duplicate of a given array and returns a pointer to the duplicate.

   (b) `public static void reverse (int[] array)` A method that reverses the order of the elements in an array *in-place* (that is, within the given array itself).

   (c) `public static boolean compare (int[] x, int[] y)` A method that returns `true` if the two arrays store identical contents, `false` otherwise.

   Answer:

```
public static int[] copy (int[] original) {
  int[] duplicate = new int[original.length];
  for (int i = 0; i < original.length; i = i + 1) {
    duplicate[i] = original[i];
  }
  return duplicate;
}

public static void reverse (int[] array) {
  for (int i = 0; i < array.length / 2; i = i + 1) {
    int other = array.length - i - 1;
    int temp = array[i];
    array[i] = array[other];
    array[other] = temp;
  }
}

public static boolean compare (int[] x, int[] y) {
  if (x.length != y.length) {
    return false;
  }
  for (int i = 0; i < x.length; i = i + 1) {
    if (x[i] != y[i]) {
      return false;
    }
  }
  return true;
}
```

Discussion: The types of mistakes here were various. Although the `copy()` method was often written correctly (or nearly so), the other two contained many problems.

For `reverse()`, the most common mistake was to traverse the entire array. This approach, when combined with a swapping of values from the lower- and upper-indexed locations in the array, would yield a double-switching of values that restored the array to its original order! Just as commonly, people would traverse the array in a reasonable way, but fail to swap the values in any manner that could be excused as *correct*. The final common error was to create a new array and assemble the reversed values within it, but then then to fail to recognize that doing so (without copying that result into the passed `array`) would not be an *in-place* solution.

The `compare()` method was similarly riddled. Most students missed the need explicitly to compare the lengths of the arrays as an initial test; arrays of different lengths cannot hold identical contents. Some attempted the simplified solution in which `x` and `y` were simply and directly compared (that is, `x == y`), which does not check the contents of the array and should have been suspiciously simple. Others compared two items, but upon finding a single such pair of values identical, they would prematurely `return true`. A final common error (that, to be honest, made me grumpy at moments) was the tendency to `System.out.println()` the boolean return rather than returning it.

4. QUESTION: (30 points) Consider *Pascal's Triangle*, printed in a slightly lopsided form:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

The first line, and the elements at the edges, are always 1. Every other element in this triangle is the *sum of the element above it with the element above and to the left of it.* That is, element 3 on row 5 is the sum of element 2 and 3 from row 4.
Write a method that prints Pascal's triangle. It should accept a *size* parameter that indicates how many rows to print.

ANSWER:

```
public static void pascal (int size) {
  int[] current = {1};
  for (int row = 0; row < size; row = row + 1) {
    print(current);
    int[] previous = copy(current); // From Question 3.
    current = new int[previous.length + 1];
    current[0] = 1;
    for (int i = 1; i < current.length - 1; i = i + 1) {
      current[i] = previous[i] + previous[i - 1];
    }
    current[current.length - 1] = 1;
  }
}

public static void print (int[] array) {
  for (int i = 0; i < array.length; i = i + 1) {
    System.out.print(array[i] + " ");
  }
  System.out.println();
}
```

DISCUSSION: First, a reminder that this question was intended to be particularly challenging. I did not expect many correct solutions, and simply wanted to give each of you an opportunity to tackle a problem whose solution would require quick and clear (but somewhat novel) application of solutions and con-

cepts that we had covered. Consequently, the average score on this question was expectedly low. Second, it is possible to solve this problem with some recursive method support (specifically, a method that calculates the value at position (row, col)), but getting it right is tricky.
Although few managed an answer that suggested a potentially working solution, many did manage a nested loop structure to represent the triangle shape of the pattern that needed to be printed. Often that nested loop structure (particularly the inner loop) was faulty, failing to address the incrementally increasing length of each row. Some attempted an array-based structure like the one given above, but failed to create two arrays, using one as the current row whose values are derived from the other that stores the previous row. Many more solutions simply attempted to somehow calculate the values from the row and column positions alone, signifying a lack of time to develop a better solution more than anything else.