Fall 2005 – CS 11 – Final exam Answers

1. Question: (5 points) In the following snippet of code, identify all places where type casting would occur automatically or would need to occur through a forced cast. For each cast, indicate whether it is automatic or forced, and indicate the types involved. Most of all, be sure to indicate exactly which variable or expression is being cast.

```
int x = 5;
char y = x;
char z = y + 13;
double d = x;
double e = x * 5.0;
int q = y + z;
```

   Answer: Line by line, with forced casts inserted as needed...

   - `int x = 5;` No cast.
   - `char y = (char)x;` Forced cast, *int* → *char*.
   - `char z = (char)(y + 13);` Automatic cast of `y`, *char* → *int*, followed by a forced cast on the result of the addition operation, *int* → *char*.
   - `double d = x;` Automatic cast, *int* → *double*.
   - `double e = x * 5.0;` Automatic cast of `x`, *int* → *double*.
   - `int q = y + z;` Automatic cast of the result of the addition operation, *char* → *int*.

   Commentary: Many people missed some of the automatic casts. Commonly, answers were not specific about either which part of the expression was cast or what types were involved in the cast.

2. QUESTION: (5 points) What is the output of the program below when invoked with the command line:

```
java Foo 4 newyear
```

```
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("newyear");
        if (s == (args[1])) {
            baz(d, 1);
        } else {
            baz(d, -1);
        }

        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, int x) {

        if (x == 1) {
            d = new double[d.length / 2];
        }

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

    } // baz

} // class Foo
```

ANSWER:

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

COMMENTARY: The first common error concerns the line, in `main()`, that compares `s` to `args[1]`. Specifically, `s` points to a `String` object that is, on the previous line, explicitly created anew. It cannot be the case that `args[1]` points to that same object. The comparison is one that *compares pointer values*, and since the two pointers cannot point to the same object, the two pointers cannot be equal.
The second error concerns the line, in `baz()`, in which `d` is assigned to point to a new array of half its original length. Neither the creation of this new array nor any subsequent changes to

the new array will be visible to `main()`, and thus will not be printed as part of the output of the program. That's because `d`, which is the only pointer to that new array, is a local variable that disappears when `baz()` ends. Note that this problem arises only for those who believed that `x` should be `1`.

The final source of error addresses those who (correctly) believe that `x`, within the call to `baz()`, should be `-1`. Specifically, in that case, `d` points to the original array that was created within and passed from `main()`. Thus, changes to the array—specifically, the loop within `baz()` that reassigned the value of each element of the array—*are* visible to `main()` and are printed as part of the program's output.

3. QUESTIONS: (10 points) Consider the problem of verifying that a given n-element array contains one of every value from 1 through n. Now consider the following two method 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;

}
```

As n becomes arbitrarily large, which verifier is going to run faster (e.g. perform fewer operations)? Can you state a rough formula (e.g. a *Big-O* expression) for the time each verifier would take as a function of n?

ANSWER: Version A performs fewer operations. Specifically, version A is $O(n)$, while version B is $O(n^2)$.

COMMENTARY: Nearly everyone got this question, and few people lost points on it. Nice work! The key observation is that version A makes a single pass over the n-element input array, recording in a second array the occurrence of each value. Version B, however, while visiting each element in the array, visits every *other* element in the array as well. This, for each of the n elements, it visits all other $n - 1$ elements, yielding an $O(n^2)$ running time.

4. QUESTION: (10 points) Write a **non-recursive** method that computes the n^{th} Fibonacci number. Recall that the Fibonacci sequence is defined as:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

ANSWER: The following iterative method calculates the n^{th} Fibonacci number:

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

    if (n < 0) {
        System.out.println("ERROR: Invalid request for fib(" + n + ")");
        System.exit(1);
    }

    if (n <= 1) {
        return n;
    }

    int x = 0;
    int y = 1;
    for (int i = 2; i <= n; i++) {
        int temp = x + y;
        x = y;
        y = temp;
    }

    return y;

}
```

COMMENTARY: Most provided good answers for this problem as well. Large errors usually revolved around not understanding the question and writing a method that performed a fundamentally different task. Small errors were in the details of computing the exact Fibonacci number requested.

5. QUESTION: (15 points) 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[] a, int[] b) {

    if (a.length != b.length) {
        return false;
    }

    int j;
    boolean[] matched = new boolean[b.length];
    for (int i = 0; i < a.length; i++) {
        for (j = 0; j < b.length; j++) {

            if ((a[i] == b[j]) && (!matched[j])) {
                matched[j] = true;
                break;
            }

        }

        if (j == b.length) {
            return false;
        }

    }

    return true;

} // multisetEqual
```

COMMENTARY: There are a number of different approaches that could legitimately be used for this problem. The one presented above tries to match each element in `a` with an element in `b`, using an array of `boolean` to keep track of elements from `b` have already been matched to avoid double-matchings.

One popular but not fully legitimate approach was to create a "counts" array in which you counted the number of occurrences of each value the arrays. You have no information about the range of values that appear in these arrays. Thus, to find the minimum and maximum values and then create an additional array that spans that range ignores that the size of the counts array could be well beyond the capacity of a normal machine. An additional and common error in solutions that used this method was to assume (incorrectly) that all values in the original arrays would be positive, thus not compensating for negative values when accessing the counts array. Nonetheless, use of this approach only cost a modest number of points, as it is an otherwise correct strategy.

6. QUESTION: (15 points) *Matrix addition* involves taking two 2-D arrays of the same height and width (let's call them A and B) and then, for each position in the matrices, adding the value from A in that position to the value from B in that position and placing it is a third matrix, C, in that position. Thus, matrix addition takes this form (for two 3×3 matrices):

$$\begin{bmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{bmatrix} = \begin{bmatrix} A_1 + B_1 & A_2 + B_2 & A_3 + B_3 \\ A_4 + B_4 & A_5 + B_5 & A_6 + B_6 \\ A_7 + B_7 & A_8 + B_8 & A_9 + B_9 \end{bmatrix}$$

Write a method that accepts 3 pointers to 2-D arrays of `double` (that is, three matrices). This method should determine **whether one of the matrices is the result of matrix addition of the other two**.

ANSWER:

```
public static double[][] whichIsSum (double[][] x,
                                     double[][] y,
                                     double[][] z) {

    if (isSum(x, y, z)) {
        return x;
    } else if (isSum(y, x, z)) {
        return y;
    } else if (isSum(z, x, y)) {
        return z;
    } else {
        return n;
    }

} // whichIsSum

public static boolean isSum (double[][] s, double[][] a, double[][] b) {

    for (int i = 0; i < s.length; i++) {
        for (int j = 0; j < s[0].length; j++) {

            if (s[i][j] != a[i][j] + b[i][j]) {
                return false;
            }

        }
    }

    return true;

} // isSum
```

COMMENTARY: This question is easy to answer if you break your solution into two methods, as shown above. As a single method, a solution would be unwieldy. Testing any one matrix as a sum of two other matrices is just a matter if testing each position in the one matrix is the sum of the same positions in the other two. With a method that can perform such a test, another method can call on that method with each possible arrangement of matrices. Since addition is commutative, the number of combinations is small.
The most common error here was to assume that your method(s) should test only *one* arrangement of matrices—that is, that the only test required was to determine whether the first was the sum of the second and third. Doing so ignored an important part of this problem.

7. QUESTION (20 points) Consider a program that is invoked thusly:

```
java Triangles 37
```

The `main()` method of the *Triangles* program needs to convert the command line argument (here, 37) into an `int` and then pass that `int` to an already written `printPattern()` method. Write the `main()` method described above. Furthermore, assume that there **is no such method as** `Integer.parseInt()`. Thus, you must **also write** a method named `myParseInt()` that converts a string representation of an integer value (as a sequence of numeral characters) into an actual `int`. Your method should throw a pointer to a `NumberFormatException` object if it cannot perform the conversion, and `main()` should catch that exception, printing a helpful error message when it occurs.

ANSWER:

```
public static void main (String[] args) {

    if (args.length != 1) {
        System.out.println("USAGE: java Triangles <pattern size>");
        System.exit(1);
    }

    int size = myParseInt(args[0]);
    printPattern(size);

}

public static int myParseInt (String s) {

    boolean isNegative = false;
    int i = 0;
    if (s.charAt(0) == '-') {
        isNegative = true;
        i = 1;
    }

    int v = 0;
    while (i < s.length()) {

        if ((s.charAt(i) < '0') || (s.charAt(i) > '9')) {
            throw new NumberFormatException();
        }

        v = (v * 10) + ((int)(s.charAt(i) - '0'));
        i++;

    }

    if (isNegative) {
        v = -v;
    }

    return v;

}
```

COMMENTARY: The most common problem was the use of length *if-then-else* chains to convert

each character into an integer value. The second most common (and more significant) error was to forget that some such conversion had to be performed: that is, some forgot that the value of the character `'5'` is not the integer `5`.

One common approach was to calculate the exact contribution of each character to the final value right away. That is, for the string "`514`", the first step was to multiply 5 by 10^2. Unfortunately, this approach requires exponentiation—an operation that is not available in Java (at least not without a method designed to perform it). As the above approach shows, it's also not necessary. (Do you see how it works?)

The final common error was the handling of strings that could not be converted into integers. Some forgot to test for invalid (non-numeric) characters at all; others handled such errors by printing an error message and exiting. The only correct handling of such strings was to create and throw a `NumberFormatException` object.

Note that the handling of negative numbers was not critical, but it is better for your method to have been able to handle it.

8. QUESTION: (20 points) You have three types of stamps at your disposal for placing postage on letters and mailing them: *1-cent* stamps, *11-cent* stamps, and *37-cent* stamps. You mail lots of very oddly shaped and weighted packages, so you often need unusual amounts of postage. You just dispise licking the awful adhesive backing of the stamps, so you find it worthwhile to lick as few as possible.
Complete the body of the following **recursive** method that searches for the fewest stamps possible to create the exact desired postage. The method must return a 3-element array where the 0^{th}, 1^{st}, and 2^{nd} elements of the array indicate the number of 1−, 11−, and 37−cent stamps needed.

```
public static int[] makePostage (double postageNeeded) {
```

ANSWER: The whole class (although your answer only needed the `makePostage` method and any non-trivial supporting methods.

```
class MakePostage {

    public static void main (String[] args) {

        if (args.length != 1) {
            System.err.println("USAGE: java MakePostage <postage amount in cents>");
            System.exit(1);
        }

        int amount = Integer.parseInt(args[0]);
        int[] result = makePostage(amount);

        System.out.println(" 1 cent stamps: " + result[0]);
        System.out.println("11 cent stamps: " + result[1]);
        System.out.println("37 cent stamps: " + result[2]);

    }

    public static int[] makePostage (int amount) {

        if (amount < 0) {
            return null;
        } else if (amount == 0) {
            return new int[3];
        }

        int[][] postageMatrix = new int[3][];
        postageMatrix[0] = makePostage(amount - 1);
        postageMatrix[1] = makePostage(amount - 11);
        postageMatrix[2] = makePostage(amount - 37);

        int[] stampCounts = new int[3];
        for (int i = 0; i < 3; i++) {
            stampCounts[i] = sum(postageMatrix[i]);
        }

        int smallestIndex = minIndex(stampCounts);
        if (smallestIndex == -1) {
            return null;
        }
        postageMatrix[smallestIndex][smallestIndex]++;
```

```
        return postageMatrix[smallestIndex];

    }

    public static int sum (int[] x) {

        if (x == null) {
            return -1;
        }

        int sum = 0;
        for (int i = 0; i < x.length; i++) {
            sum = sum + x[i];
        }

        return sum;

    }

    public static int minIndex (int[] x) {

        int minIndex;
        for (minIndex = 0; x[minIndex] == -1; minIndex++) {}

        for (int i = minIndex + 1; i < x.length; i++) {

            if ((x[i] < x[minIndex]) && (x[i] != -1)) {
                minIndex = i;
            }

        }

        return minIndex;

    }

}
```

COMMENTARY: This was a hard problem, particularly in getting all of the details right. Most particularly, finding a way to end the recursion, to record the correct information about the stamps, and to exclude results from recursive calls on impossible (i.e. negative) amounts of postage was tricky. I did not expect anyone to get all of those details right, and in truth, nobody did. The above solution is a bit tricky—be sure you see how it really works.

It was more important that you show a basic strategy for searching for the right combination of postage (one major source of error) and for returning to the originally caller an array of `int` that reflected how many of each stamp to use (another major source of error). The types of errors here varied widely.

Note that in my solution, I used an integer number of cents instead of a floating-point number of dollars. The problem asked that you use a `double`, and I graded it with that requirement. However, in real life, use of `double` values was problematic because of rounding errors (e.g. $0.03 - 0.01 = 0.19999999998$ because of precision issues that we never discussed.) Thus, my solution actually works, but it does not exactly conform to the question as asked.