CS61B Week 7: Midterm Review

1. Write a function `counts` that, when given an array of positive ints in the range of `[0, Integer.MAX_VALUE]`, returns an array where the value at each index in the array is the number of times that index appeared in the input. For example, if given `[1, 4, 5, 4, 3, 1, 4, 4, 4]`, `counts` will return `[0, 2, 0, 1, 5, 1]`.

```
public static int[] counts(int[] input) {
    int maxValue = 0; // determines how large your result array should be
    for (int x : input) {
        if (x > maxValue) {
            maxValue = x;
        }
    }
    int[] result = new int[maxValue + 1]; // Java inititalizes integer arrays to 0
    for (int x : input) {
        result[x]++;
    }
    return result;
}
```

2. How can you check to see if an int is less than 0 using only == or != and the bit operators?
   There are multiple solutions to this. Given an int a, we can do `a >>> 31 == 1`, which simply checks if the sign bit of `a` is 1. This is an acceptable answer.
   We can actually do this without using any integer literals though: `a >>> (a ^ (~a)) != a ^ a`. This works since `a ^ (~a) = -1` and `a ^ a = 0`, and shifting by -1 is the same as shifting by -1 mod 32, which is 31. Alternate solution: `a >>> (a ^ (~a)) != a >> (a ^ (~a))`

3. Jack has a bug: his `Account` objects keep losing (or gaining) money seemingly at random as his program executes. What is the probable cause of the error in this code that he bug-submitted:

```
public class Account {
    /** A new account with the given ID and initial balance BALANCE. */
    Account(String id, int balance) {
        _id = id;
        _balance = balance;
    }
    void deposit(int amount) {
        if (amount < 0)
            throw new IllegalArgumentException("negative deposit");
        _balance += amount;
    }
    void withdraw(int amount) {
        if (amount < 0 || amount > _balance)
            throw new IllegalArgumentException("invalid amount");
        _balance -= amount;
    }
    int balance() {
        return _balance;
    }
```

```
    /** ID of this account. */
    final String _id;
    /** Current balance. */
    static int _balance;
}
```

Each account is supposed to have its own `_balance` value. But Jack has declared it static, meaning that it is a class variable, with only one value shared by all instances.

4. Given the following classes, what does `Templ.s(new Grandchild(2))` print? It may be a runtime error.

```
abstract class Templ {
    abstract int f();
    int g() {
        return 2 * f();
    }
    static void s(Templ x) {
        System.out.println(x.g());
    }
}
class Child extends Templ {
    Child(int z) {
        _z = z;
    }
    int f() {
        return _z;
    }
    int _z;
}
class GrandChild extends Child {
    GrandChild(int z) {
        super(z);
        _z = 5*z;
    }
    int g() {
        return 3 * f();
    }
    int _z;
}
```

The method `s` is called with an argument with dynamic type `GrandChild`. Calling `g` on this, therefore, calls `GrandChild.g`, this being the behavior of instance methods. This in turn calls `Child.f`, inherited from `Child` by `GrandChild`. That method returns the `_z` defined in `Child`, since the method is defined in `Child` so the variable `_z` is scoped to that class. Hence the result will be 3 * 2 = 6.

5. Fill in the following method to obey its comment. Introduce any auxiliary methods you want.

```
/** Distribute the elements of L to the lists in R round-robin
 * fashion. That is, if m = R.length, then the first item in L
 * is appended to R[0], the second to R[1], ..., the mth item
 * to R[m-1], the m+1st to R[0], etc. So if R starts out containing
 * the IntList sequences [1, 2], [3, 4, 5], and [], and L starts out
 * containing [6, 7, 8, 9], then distribute(L, R) causes R to end up
 * containing [1, 2, 6, 9], [3, 4, 5, 7], and [8]. May destroy the
 * original list L. Must not create any new IntList elements. */
static void distribute(IntList L, IntList[] R) {
    int i;
    i = 0;
    while (L != null) {
        IntList p = L;
        int k = i % R.length;
        L = L.tail;
        R[k] = attach(R[k], p);
        i = i + 1;
    }
}

/** Append the list element Y destructively to the end of X, returning
* the modified X. */
static IntList attach(IntList X, IntList Y) {
    Y.tail = null;
    if (X == null) {
        return Y;
    } else {
        IntList p;
        for (p = X; p.tail != null; ) {
            p = p.tail;
        }
        p.tail = Y;
        return X;
    }
}
```

6. A `FileList` is a kind of read-only `List<String>` whose items are words that come from a `Scanner`. Thus, if the Scanner, `input`, is created to read from a file containing

```
My eyes are fully open to my awful situation,
I shall go at once to Roderick and make him an oration.
```

then after

```
FileList f = new FileList(input);
```

we'd have `f.get(0).equals("My")`, `f.get(8).equals("situation,")`, and so forth. We require that the `FileList` never tries to read any more from the `Scanner` than needed to fulfill the needs of any particular call on its method. For example, it never asks its Scanner to read the fifth word from the input until the user first calls `.get(4)` to get the fifth item of the list, or makes some other call (such as `.size`) that requires actually finding out what that item is (or whether it exists at all). Once the fifth item has been read, subsequent calls to `.get(4)` retrieve the same item from memory. As a result, we should be able to apply the `FileList` to a Scanner that takes its input from the terminal, without having its operations hang until the user has typed in all the input. Fill in the methods shown for the partial definition of `FileList` below to meet this specification. Add any instance variables or private methods you need. You may use any classes in the Java library.

```
public class FileList extends AbstractList<String> {

    private ArrayList<String> _contents = new ArrayList<String>();
    private Scanner _input;
    /* We assume that _input is set by an (unmentioned) constructor. */

    @Override
    public int size() {
        while (_input.hasNext()) {
            _contents.add(_input.next());
        }
        return _contents.size();
    }


    @Override
    public String get(int k) {
        while (_contents.size() <= k) {
            if (!_input.hasNext()) {
                throw new IndexOutOfBoundsException("" + k);
            } else {
                _contents.add(_input.next());
            }
        }
        return _contents.get(k);
    }
}
```