Midterm 1, CS161, Spring '12, Section 1

1. The following is the signature for a version of `Merge` that we studied:

   ```
   Merge(int [] A, int i, int j, int k)
   ```

   It is of type `void`.

   (a) Give a complete set of preconditions on the values i, j, and k. These should be expressed in terms of boolean operators, such as "<". Don't use English.

   Solution:$0 \leq i \leq j < k < A.length$; *this just says that* i, j, k *are valid indices and that* $A[i..j]$ *and* $A[j+1..k]$ *are nonempty.*

   (b) Give any additional preconditions on A. You can express these in English, as long as it is succinct and leaves no ambiguity.

   Solution:$A[i..j]$ *and* $A[j+1..k]$ *are in nondecreasing (sorted) order.*

   (c) Give the postcondition in English. Be succinct and don't leave any ambiguity about what is expected.

   Solution: $A[i..k]$ *has been sorted; the rest of the array is left unchanged.*

2. Here is a variation on binary search called *ternary search.* It solves the same problem that binary search does. Fill in the missing pieces of code so that the method will meet its postcondition whenever its precondition is met.

```
// preconditions:  A[i..k] is in nondecreasing order; i <= k and i and
//   k are valid indices in A
// postcondition:  if val occurs in A[i..k] return an index where it occurs
//   there; otherwise return -1

public int TS(int [] A, int i, int k, val)
    if (k-i+1 < 3)  // fewer than three elements -- the recursive case would
                    //   commit the 'cardinal crimes' we talked about
    {
        for (int c = i; c <= k; c++) // one way to check one or two elements
           if (val == A[c]) return c;
        return -1;
    }
    else
    {
       int third = (k - i + 1) / 3;
       if (val < A[i+third]) return TS(A,i,i+third-1,val);
       else if (val > A[k - third]) return TS(A,k-third+1,k,val);
       else return TS(A,i+third,k-third,val);
    }
}
```

3. Study the attached implementation of `ListArrayBased`. Let's assume a precondition on `add` that the class for the object that's passed must have a `toString()` method.

Write a `toString` method for the `ListArrayBased` class. For simplicity, let's require it to return a String that, when printed, displays each object in the list, using the `toString` for that object, followed by a space.

```
public String toString()
{
    String result = new String();
    for (int i = 0; i < getSize(); i++)
        // when items[i] appears in a String expression, the toString()
        //  of the object it references is called ...
        result += items[i] + " ";
    return result;
}
```

4. If your `toString` works, what will appear on the screen when we run the attached `main`?

```
Good afternoon good evening and good night
```

5. Suppose we now add the following immediately after the `for` loop in the `try` block of this `main`:

```
L.add(5,"!");
```

What will be the output we see when we run it? (Careful! Study the code for the class.)

Solution:

```
Out of bounds!!
```

(Not "Index out of bounds on add")

6. Now suppose that we restore the `main` to be the attached one, but then change third line of the class definition to the following:

```
private static final int MAX_LIST = 3;
```

What will appear on the screen now when we run `main`?

```
Good afternoon good evening and good night
```

No exception is thrown. If you wrote "You really screwed up this time!" then you did. (However, I'll give partial credit for this answer, since it shows you know how the exception is handled when it is thrown.)

7. Suppose we restore the attached `main`, but then change the following line:

```
        ListArrayBased L = new ListArrayBased();
```

to this:

```
        ListInterface L = new ListArrayBased();
```

Will it compile? If not, why not? If so, what will the screen output of `main` be?

Solution: *Yes, a* `ListArrayBased` *object can be assigned to a variable of type* `ListInterface`*, since* `ListArrayBased` *implements* `ListInterface`*.*

8. Suppose that instead we change this line to the following:

```
        ListInterface L = new ListInterface();
```

Will it compile? If not, why not? If so, what will the screen output of `main` be?

Solution: *It won't compile.* `ListInterface` *is not a class, so it has no constructor or any other executable code.*

9. Let's restore our `main` to the original one listed above, and then suppose that immediately following the `for` loop in the `try` block, we add the following:

```
    L.add(3,L);
```

(a) Why will the compiler not complain?
Solution: *An object of any type can be added to the list. L is a reference to an object, so the compiler won't complain.*

(b) What will happen when we hit the print statement at the end of `main` now?
Solution: *Element number 3 of L is L, so when L's* `toString()` *gets to this element, it calls L's* `toString()`*. This is actually a recursive call to itself! However, this commits one of the cardinal crimes we talked about – failing to make a recursive call on a smaller instance. It will go into infinite recursion. This will trigger a "stack overflow" error message.*

10. A trucker shows up at a loading dock where there is a set of boxes to choose from. They are all going to the same town. Each box is labeled with its weight, which is an integer number of pounds, and the fee he will be paid if he takes the box there, which is an integer number of dollars. His dilemma is that if he exceeds t pounds, he will get fined at the truck weigh station.

He must pick and choose from the boxes in order to maximize the fees, subject to the constraint that he can't exceed t pounds.

Write a method with the following signature:

```
// precondition:  0 < n == Weights.length == Fees.length.  For each
//  i from 0 through n-1, Weights[i] and Fees[i] give the weight and fee
/   for item i.
// postcondition:  the maximum fees he can collect without
//  exceeding t pounds has been returned
int bestLoad(int n, int [] Weights, int [] Fees, int t)
```

Here's a strategy: The best load will either contain the last item ($n-1$) or it won't.

(a) If it doesn't contain this item, then he can solve the problem by solving the smaller instance of the problem he gets when he removes item $n-1$ from consideration.

(b) If it does contain this item, then he should put item $n-1$ in the truck, and solve the smaller instance he gets when he removes item $n-1$ from consideration, and optimally fills the remaining capacity of the truck. This is also a smaller instance of the problem.

Since he doesn't know which one will be best, he can solve both problems and pick the best of the two. Since they both involve smaller instances of the problem, the strategy is nicely suited to recursion.

```
int bestLoad(int n, int [] Weights, int [] Fees, int t)
{
    if (n == 0 || t == 0) return 0;
    else
    {
       // best you can do without item n-1 ...
       int v1 = bestLoad(n-1, Weights, Fees, t);

       // best you can do with item n-1 ...
       int v2 = 0;
       if (Weights[n-1] <= t)  // if item n-1 will fit put it in the truck;
          // calculate Fees[n-1] plus the best you can do with the remaining
          //  items and remaining capacity ...
          v2 = Fees[n-1] + bestLoad(n-1, Weights, Fees, t-Weights[n-1]);

       // return the best of the two options ...
       if (v1 > v2) return v1;
       else return v2;
    }
}
```

```
// A ListInterface.java file is not shown, but assumed to be available.
public class ListArrayBased implements ListInterface
{
    private static final int MAX_LIST = 50;
    private Object items[];
    private int numItems;

    public ListArrayBased()
    {
        items = new Object[MAX_LIST];
        numItems = 0;
    }

    public boolean isEmpty()
    {
        return (numItems == 0);
    }

    public int getSize()
    {
        return numItems;
    }

    protected void setSize(int n)
    {
        numItems = n;
    }

    public void add(int index, Object item) throws Exception, IndexOutOfBoundsException
    {
        if (index < 0 || index > getSize())
            throw new IndexOutOfBoundsException("Index out of bounds on add");
        else if (numItems >= MAX_LIST)
            throw new Exception ("Now you really screwed up!");
        else
        {
            for (int pos = getSize() - 1; pos >= index; pos--)
                items[pos+1] = items[pos];
            items[index] = item;
            setSize(getSize() + 1);
        }
    }

    public Object get(int index) throws IndexOutOfBoundsException
    {
        if (index < 0 || index >= getSize())
            throw new IndexOutOfBoundsException();
        else
            return items[index];
    }
}


---------------------

public static void main(String [] args)
    {
        String [] S = {"Good afternoon",
                        "good evening", "and good night"};

        ListArrayBased L = new ListArrayBased();
        try
        {
           for (int i = 0; i < S.length; i++)
               L.add(i, S[i]);
        }
        catch(IndexOutOfBoundsException e)
        {
            System.out.println("Out of bounds!!!");
            System.exit(0);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println(L);
    }
```