Assorted Reference Material

IntList

```
public class IntList {
   public int head;
   public IntList tail;
   public IntList (int head, IntList tail) {
       this.head = head; this.tail = tail;
   }
}
```

Excerpt from class java.lang.String

```
package java.lang;
public final class String implements Serializable, Comparable, CharSequence {

   /** The Kth character of THIS, 0 <= K < length(). */
   public char charAt (int k) { ... }

   /** The number of characters in THIS. */
   public int length () { ... }

   /** A value <0 if THIS comes before X in lexicographic order, ==0
    *  if equal, and >0 if THIS comes after X. */
   public int compareTo (String x) { ... }

   /** The String containing characters #START, START+1, ..., END-1 of
    *  THIS. */
   public String substring (int start, int end) { ... }
}
```

java.util.Iterator

```
package java.util;
public interface Iterator<SomeType> {
    /** True iff there is an item for next() to deliver. */
    public boolean hasNext ();

    /** The next item to be returned by THIS. */
    public SomeType next ();

    /** (Optional) Remove the last item returned by THIS. */
    public void remove ();
}
```

1. [3 points] Provide simple and tight asymptotic bounds for the running times of the following methods as a function of the value of N, the length of `A`. If possible, give a single Θ bound that always describes the running time, regardless of the input. Otherwise give an upper and a lower bound. (To be precise, we assume there is some set of values for `A`, one of each size, and we want you to put as good a bound as you can on the running time as a function of the size that will hold for any such set of values).

a.
```
static void foo(int[] A) {
  int N = A.length;
  for (int n = N; n > 1; n -= 1) {
    for (int i = 1; i < n; i += 1) {
      if (A[i - 1] > A[i]) {
        int tmp = A[i - 1];
        A[i - 1] = A[i];
        A[i] = tmp;
      }
    }
  }
}
```

Bound(s): $\Theta(N^2)$

b.
```
static int bar(int[] A) {
  int N = A.length;
  int S;
  S = 0;
  for (int i = 0; i < N; i += 1) {
    for (int j = i+1; j < N; j += 1) {
      if (A[j] == A[i]) {
        S += 1;
        break;
      }
    }
  }
  return S;
}
```

Bound(s): $\Omega(N), O(N^2)$

c.
```
static boolean sump(int[] A, int S) {
   return sump1(A, S, 0);
}

private static boolean sump1(int[] A, int S, int k) {
   int N = A.length;
   if (S == 0)
      return true;
   else if (k >= N)
      return false;
   else if (S >= A[k] && sump1(A, S-A[k], k+1))
      return true;
   else return sump1(A, S, k+1);
}
```

Bound(s) on `sump`: $\Omega(1), O(2^N)$

(Include time of the helper function `sump1` in the time for `sump`).

2. [4 points]

a. Assuming that $0 \leq \texttt{m} \leq 31$, fill in the blank with a Java expression involving only the variable `m` (not `s` or `k`) so that the program fragment below prints `"OK"`:

```
int k, s;
k = 1; s = 0;
for (int i = 1; i < m; i += 1) {
    s = s|k;
    k = k<<1;
}

if (s == (1<<(m-1 > 0 ?  m-1 :  0)) - 1 )
   System.out.println ("OK");
```

The expression in the blank may not call any methods.

However, since the problem was harder than intended (should have started at `i = 0`*), we also accepted* `(1<<(m-1)) - 1` *and* `(1<<m) - 1`*.*

b. One of the three comments below must be wrong (that is, unsatifiable regardless of the method body). Which one and why just that one? [NOTE: you *must* tell why to get full credit.]

```
/** Upon completion, A contains the concatenation of the original
 *  list A in front of the list B.  The original contents of
 *  B are not affected.  */
void concat1 (IntList A, IntList B) { ... }

/** Returns the result of concatenating the contents of A in front of
 *  the contents of B.  The original contents of A and B are not
 *  affected. */
IntList concat2 (IntList A, IntList B) { ... }

/** Upon completion, A contains the concatenation of the original list
 *  A in front of the list B.  The original contents of B are
 *  not affected.  Assumes that A is not empty. */
void concat3 (IntList A, IntList B) { ... }
```

The comment on concat1 must be wrong. If A is null, then there is no way to cause A to change from within concat1. concat2 is possible to implement, since one can return the result directly. concat3 also works because we can simply set the tail of the last element of A (there must be at least one) to B.

c. What does the following legal program print when the main program is invoked?

```
class Exceptions {
    public static void main (String... args) {
       try {
           f (42, new IllegalArgumentException ());
       } catch (IllegalArgumentException e) {
           System.out.println ("main");
       }
    }

    static void f (int n, IllegalArgumentException err) {
       if (n > 39) {
          try {
             f (n-1, err);
          } catch (IllegalArgumentException e) {
            System.out.printf ("f(%d) err%n", n);
          }
          System.out.printf ("f(%d)%n", n);
       } else {
          g (n, err);
       }
    }

    static void g (int n, IllegalArgumentException err) {
       if (n < 3)
           throw err;
       else f(n-1, err);
    }
}
```

This throws an exception from g when $n < 3$, which is caught by the last invocation of f to create a **try** *block, which is f(40,...). Therefore, it must print*

```
f(40) err
f(40)
f(41)
f(42)
```

d. If I compile and run the program below with "`java G`," what is printed? (If execution causes a runtime error, just write "`error`" at the appropriate point in the printout. If the program won't compile, just say so.)

```
abstract class F {
   abstract int a (int x);

   F c (F g) {
      System.out.println ("F.c");
      return new C(this, g);
   }

   F r (int n) {
      System.out.println ("F.r");
      if (n <= 1)
         return this;
      else return this.c (r(n-1));
   }

   F c2 () {
      return this.r(2);
   }

   static class C extends F {
      private F f, g;
      C (F f, F g) { this.f = f; this.g = g; }
      int a (int x) { return f.a(g.a(x)); }
   }
}

class G extends F {
   int i;
   F r (int n) { System.out.println ("G.r"); return new G (n * i); }
   G (int i) { this.i = i; }
   int a (int x) { return x + i; }
   public static void main (String... args) {
      F f = new G (21);
      System.out.println (f.c2().a(0));
   }
}
```

The `c2` *method of* `F` *calls the* `r` *method of* `G`*, which overrides that of* `F`*, so that we end up calling* `a` *on* `G`*. Thus we get*

```
G.r
42
```

3. [1 point] What kind of journey are we undertaking whan that Apryll, wyth hys shouris sote, the droughte of Marche hath percyd the rote?

A pilgrimage (the Middle English phrase is from the Prologue to The Canterbury Tales by Geoffrey Chaucer).

4. [6 points] Fill in the following method to obey its comment.

```
/** Set each R[k] to a sublist of L such that R[k] contains
 *  <=k+1 elements and the concatenation of all the  R[k] in order
 *  gives a prefix of the original list L.  Each list R[k] is made
 *  as large as possible subject to these rules, with earlier lists
 *  taking precedence.  For example, if the original L contains
 *  [ 1, 2, 3, 4, 5, 6, 7 ], and R has 6 elements, then on return R
 *  contains [ [1], [2,3], [4,5,6], [7], [], []].  If  R had only 2
 *  elements, then on return it would contain [[1], [2,3]].
 *  May destroy the original contents of the IntList objects in L,
 *  but does not create any new IntList objects. */
static void triangularize (IntList[] R, IntList L) {
    // One of many possible solutions.
    int i, k;  /* i: index into R; k: number of items in R[i] */
    i = 0; k = 0;
    while (i < R.length) {
       if (k == 0)
          R[i] = L;
       if (L == null) {
           i += 1; k = 0;
       } else if (k == i) {
           IntList next = L.tail;
           L.tail = null; L = next;
           i += 1; k = 0;
       } else {
           L = L.tail; k += 1;
       }
    }
}
```

5. [7 points] Programs that work with very long sequences of characters (such as text editors) typically do not use ordinary Strings to represent the sequences because most operations on long Strings are not efficient. Instead, these programs use representations implemented using collections of much smaller Strings, each no more than a fixed, maximum size. For example, if this maximum size were 4, then such a representation could internally represent the string `"Hi, world"` as the list of Strings `["Hi, ", "worl", "d"]` or as `["Hi", ", w", "orld" ]`, etc. The `SegmentedString` class declared below is one such representation (incomplete and not particularly efficient).

a. Fill in the indicated blanks in the implementation of `SegmentedString` to fulfill the comments. From the point of view of the client who uses this class, it represents a list of characters. The fact that internally it breaks the input into smaller strings must remain completely invisible to the client. The compiler supports autoboxing, so you can assume that values of type `char` and values of type `Character` are pretty much interchangeable.

```
/** A sequence of characters. */
public class SegmentedString extends AbstractList<Character>
   implements List<Character> {

    private final static int MAX_SEGMENT_LENGTH = <some integer>;

    /** The constituent substrings of THIS. The full sequence of characters
    * that THIS represents is the concatenation of the Strings in segments.
    * Each segment has length no greater than MAX_SEGMENT_LENGTH. Segments
    * need not be of maximum length. */
    private List<String> segments;

    /** Initializes an empty sequence. */
    public SegmentedString() { // FILL IN (about 1 line)
        segments = new ArrayList<String> ();

    }

    /** Adds STR to the end of THIS character sequence. */
    public void append(String str) { // FILL IN (about 5 lines)
        for (int k = 0; k < str.length (); k += MAX_SEGMENT_LENGTH)
            segments.add (str.substring (k, Math.min (str.length, k + MAX_SEGMENT_LENGTH)));
    }
```

```
    /** Returns the length of (number of characters in) the sequence. */
    public int size() { // FILL IN (about 5 lines)
        int S;
        S = 0;
        for (String s : segments)
            S += s.length ();
        return S;
    }

    /** The character at index K in the sequence.
     *  Throws IndexOutOfBoundsException when K is negative or >= size(). */
    public Character get(int k) { // FILL IN (about 10 lines)
        int i;
        i = 0;
        for (String s : segments) {
            if (s.length () > k - i)
                return s.charAt (k - i);
            else
                i += s.length ();
        }
        throw new IndexOutOfBoundsException ();
    }
}
```

b. Suppose that we want to provide an iterator for `SegmentedString` that allows us to iterate through the entire sequence in time $O(N)$, where N is the size of the string. We can do so by adding the following to the `SegmentedString` class. Fill in just the representation—that is, the constructor, the instance variables and their comments—for the `SegIterator` class below to allow this. Make sure the comments explain exactly what each instance variable is supposed to mean.

```
public Iterator<Character> iterator () {
    return new SegIterator ();
}

private class SeqIterator implements Iterator<Character> {
    SeqIterator () {  // FILL IN
       seg = posn = 0;
    }

    /** At each point, segments.get (seg).charAt (posn) is the next
     *  character to be delivered, if there is one.  When all
     *  characters are exhausted, seg >= segments.length.  Assumes
     *  that segments are never empty strings. */
    private int seg, posn;
}
```