UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B   P. N. Hilfinger
Fall 2011

Test #1 Solutions

Assorted Reference Material

IntList

```
public class IntList {
   public int head;
   public IntList tail;
   public IntList (int _head, IntList _tail) {
       head = _head; tail = _tail;
   }
}
```

Excerpt from class java.util.String

```
package java.util;
public final class Scanner implements Iterator<String> {
   /** A new Scanner that reads from the file SOURCENAME.  Throws
    *  FileNotFoundException if the file does not exist. */
   public Scanner(File sourceName) throws FileNotFoundException {...}
   /** A new Scanner that reads from the string INPUT. */
   public Scanner(InputStream input) { ... }
   /** A new Scanner that takes its input from the TEXT. */
   public Scanner(String text) { ... }

   /** Finds and returns the next complete token from the input
    *  source.  Raises an exception if !hasNext().  A "token" is a
    *  maximal contiguous sequence of characters from the input source that
    *  does not include a substring matching the current delimiter
    *  pattern.  By default, the delimiter pattern matches
    *  whitespace. */
   public String next() { ... }
   /** True if there is another token in the input. */
   public boolean hasNext() { ... }

   /** Unsupported operation */
   public void remove() { throw new UnsupportedOperationException(); }
}
```

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. Give brief justifications of your answers.

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

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

> *Either the inner loop is never reached, and the outer loop executes N times, or the outer loop executes $K < N$ times, and then executes the inner loop $N - K$ times and returns. Either way, we get $\Theta(N)$ operations.*

b.
```
static void convolve(int[] R, int[] A, int[] B) {
    int N = A.length;
    assert N == R.length && N == B.length;
    for (int i = 0; i < N; i += 1) {
        R[i] = 0;
        for (int j = 0; j <= i; j += 1) {
            R[i] += A[j] * B[i - j];
        }
    }
}
```

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

> *The number of operations depends totally on N. The inner loop executes $i + 1$ times for $i = 0..N - 1$, making the cost proportional to $\sum_{1 \leq k \leq N} k = N(N-1)/2 \in \Theta(N^2)$.*

c.
```
private static boolean sum1(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])
        return sum1(A, S, k + 1);
    else if (sum1(A, S - A[k], k + 1) || sum1(A, S, k + 1))
        return true;
    else
        return false;
}
```

(For this one, bound the total time required to compute `sum1(A,S,0)`.)

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

In the best case, S can be 0, giving the lower bound. In the worst case, where S is sufficiently large, we get two recursive calls with k increased by one. This is identical to the "Explosive example" from the Notes, and gives an upper bound of $O(2^N)$.

2.   [4 points]

a. Fill in the blank with a single expression using only (some subset of) the operators &, | ^,
   ~, <<, >>, and >>>, so as to meet the comment.

```
/** The result of rotating X left by 4 bits.  To rotate left means
 *  that the 4 most significant bits become the 4 least significant
 *  and all other bits move left by 4.  For example,
 *  rotl4(0x12345678) is 0x23456781. */
int rotl4(int x) {

    return (x<<4) | (x>>>28);                                          ;
/* or
    return (x<<4) | ((x>>28) & 0xf);
*/
}
```

b. 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:

```java
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. */
    static final String _id;
    /** Current balance. */
    static int _balance;
}
```

Each account is supposed to have its own _balance and _id values. But Jack has declared them static, meaning that they are class variables, each with only one instance shared by all instances.

c. What's wrong with this, and what would you write instead to fulfill the programmer's probable intent?

```
class Sum {
    /** Print the sum of the first two command-line arguments in
     *  ARGS. */
    public static void main(String[] args) {
        System.out.println((int) args[0] + (int) args[1]);
    }
}
```

You mean to convert the two command-line arguments (which are Strings) into integers (ints). However, casts do not accomplish this: for any reference type, they simply return the value they are given, after checking its dynamic type. Instead, you need

```
    System.out.println(Integer.parseInt(args[0]) + Integer.parseInt(args[1]));
```

d. 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 _z is equivalent to* this._z, *and the static type of* this *is* Child, *and it is this static type, not the dynamic type, that determines which instance variable we return. Hence the result will be* $3 \cdot 2 = 6$.

3. [1 point] A ludicrously fast rocket flies past your window at 150,000 km/sec and its occupant drops a slip of paper indicating the current time according to his microsecond clock: exactly noon. (Somehow he accomplishes this without the paper instantly incinerating and without your office building being destroyed by a gigantic sonic boom). Continuing at constant speed, he then flies by another office window 150km away and drops another slip of paper indicating the current time according to his clock. What time is on the second slip?

> *The buildings are traveling at about $v = c/2$ in the frame of the rocket. Therefore, the rocket's occupant sees the distance between the buildings as $\sqrt{1 - (v/c)^2}$ times the distance in the frame of the buildings, so the time between buildings (at the same speed, v) is likewise diminished by that factor, giving about 0.00087 milliseconds past noon.*

4. [6 points] 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], L);
    }
}

/** 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;
    }
}
```

5. [7 points] A `FileList` is a kind of read-only `List<String>` whose items are words that come from a `Scanner` (see the documentation at the beginning of this test). 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` on the next page 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);

    }

}
```