1. [10 points]

a. How can you check to see if a number is less than 0 using only `==` and the bit operators (`&`, `|`, `^`, `~`, `<<`, `>>`,`>>>`)?

There are many possible answers. Here are a few:

```
(x>>>31) == 1
(x & 0x7fffffff) != x
(x & 0x80000000) != 0
(x << 1 >>> 1) != x
```

b. The following program compiles correctly. What does the main program (in `D`) print?

```
class A {
    int z = 2;
    void f () { this.g (); }
    void g () { System.out.printf ("A:%d%n", z); }
    int h () { return z; }
}

class B extends A {
    int z = 15;
    void g () { System.out.printf ("h:%d z:%d%n", h(), z); }
}

class C extends A {
    int z = 42;
    void f () { this.g (); }
}

class D {
     public static void main (String[] args) {
         A c1 = new C();
         C c2 = new C();
         A b1 = new B();
         B b2 = new B();
         c1.f ();
         c2.f ();
         b1.f ();
         b2.f ();
     }
}
```

Answer:

```
A:2
A:2
h:2 z:15
h:2 z:15
```

c. *Succinctly* describe the result of calling the following function:

```
int p (int x) {
    int n;
    n = 0;
    while (x != 0) {
       n = n ^ x;
       x = x >>> 1;
    }
    return n & 1;
}
```

Returns 1 iff there are an odd number of 1 bits in x, and 0 otherwise.

d. In the function for part (c) above, how would the result differ if we replaced `>>>` with `>>`?

You'd get an infinite loop if x is negative.

e. What is the result of compiling and executing the following? Briefly explain your answer.

```
abstract class A {
    abstract void f ();
}

class B {
    void f () { printf ("Hello, world!"); }
}

public class Main {
    public static void main (String[] args) {
       Object b = new B();
       g ((A) b);
    }

    static void g (A x) { x.f (); }
}
```

The cast `(A) b` *causes a run-time exception, because the type of* `b` *is not a subtype of* `A`*, even though it implements exactly the same methods.*

2. [10 points] Provide simple and tight asymptotic bounds for each of the following. Here, "simple" means roughly "no unnecessary terms or constants" and "tight" means "either the largest $\Omega(\cdot)$ *and* smallest $O(\cdot)$ bounds you can find or, if possible, a $\Theta(\cdot)$ bound."

a. $9x^2 + 3x + 14 \log x$

Answer: $\Theta(x^2)$

b. $\sum_{i=0}^{N} \sum_{j=0}^{i} j$

Answer: $\Theta(N^3)$

c. The running time, as a function of N, for `foo(N)`, for `foo` declared. (I'm asking for running time here, not worst-case time. So we're asking for upper and lower bounds on how long the program will run with input `N`.)

```
void foo(int N){
    int x;
    for(x = 0; x < N; x += 1) {
        int y;
        for (y = 0; y < x; y += 1) {
            bar(x,y);   // bar runs in constant time
        }
    }
}
```

Answer: $\Theta(N^2)$

More parts on the next page.

d. The running time, as some function of `low` and `high`, for `search`, declared below. That is, define a suitable function $s(\text{low}, \text{high})$, and then give a bound for $C_{\text{search}}(N)$, in terms of N, where $N = s(\text{low}, \text{high})$. (E.g., "Cost of `search`$(N) = O(N^2)$, where $N = \lg(\text{high} + \text{low})$"). (Again, I'm asking for upper and lower bounds on the time, not just a bound on the worst-case time).

```
bool search (int A[], int value, int low, int high) {
    if(high < low)
        return false;
    int mid = (low + high) / 2;
    if (A[mid] > value)
        return search(A, value, low, mid - 1);
    else if (A[mid] < value)
        return search(A, value, mid + 1, high);
    return true;
}
```

Worst-case: $O(\lg N)$, *where* $N = (high) - (low)$.
Best-case: $\Omega(1)$ *here.*

e. The running time (not just worst-case), as a function of n, for `G(`n`)`, for `G` declared

```
void G(int n) {
    if (n == 1)
        return;
    for(int i = 0; i < n; i += 1)
        G(n-1);
}
```

Answer: $\Theta(n!)$.

3. [1 point] What is an example of preadaptation?

This (rather ill-chosen) term refers to the evolution of inherited features to serve purposes much different from their originals. A classic example is that extra gill arches in ancient fish evolved to become jaw bones.

4. [10 points] Using the following class definitions:

```
class IntList {
    // 'final' means head can't be changed after the constructor
    // sets it.
    public final int head;
    public IntList tail;
    public IntList (int head, IntList tail) {
        this.head = head; this.tail = tail;
    }
}

class IntList2 {
    public final IntList head;
    public IntList2 tail;
    public IntList2 (IntList head, IntList2 tail) {
        this.head = head; this.tail = tail;
    }
}
```

fill in the methods below and on the next page to agree with their comments. Define any additional methods you'd like. HINT: don't try to make things efficient. You'll probably find it easier to create one list at a time.

```
/* a. */
/** Slice the list L into a list of N lists such that list #k contains
 *  all the items in L that are equal to k modulo N, in their original
 *  order.  For example, if N is 3 and L contains [9, 2, 7, 12, 8, 1, 6],
 *  then the result is [ [9, 12, 6], [7, 1], [2, 8] ].  The operation
 *  is destructive (it may destroy the original list) and creates no new
 *  IntList objects (it will, of course, create new IntList2 objects).
 */
static IntList2 dslice (IntList L, int N) {
    IntList[] slices = new IntList[N];  // NOTE: Creates no IntLists!
    while (L != null) {
        IntList next = L.tail;
        int k = L.head % N;
        L.tail = slices[k];
        slices[k] = L;
        L = next;
    }

    IntList2 result;
    result = null;
    for (int k = N-1; k >= 0; k -= 1)
        result = new IntList2 (dreverse (slices[k], null), result);
    return result;
}

static IntList dreverse (IntList L, IntList rest) {
    if (L == null)
      return rest;
    IntList next = L.tail;
    L.tail = rest;
    return dreverse (next, L);
}
```

```
/* b. */
/** A list of N lists such that list #k contains all the items in L
 *  that are equal to k modulo N, in their original order.  For
 *  example, if N is 3 and L contains [9, 2, 7, 12, 8, 1, 6],
 *  then the result is [ [9, 12, 6], [7, 1], [2, 8] ].  The operation
 *  is nondestructive (the original contents of L are not changed).
 */
static IntList2 slice (IntList L, int N) {
    return slice (L, N, 0);
}

static IntList2 slice (IntList L, int N, int k) {
    if (k >= N)
        return null;
    else
        return new IntList2 (select (L, N, k), slice (L, N, k+1));
}

static IntList select (IntList L, int N, int k) {
    if (L == null)
        return null;
    else if (L.head % N == k)
        return new IntList (L.head, select (L.tail, N, k));
    else return select (L.tail, N, k);
}
```

Alternative solution to part (a):

```
/* a. */
/** Slice the list L into a list of N lists such that list #k contains
 *  all the items in L that are equal to k modulo N, in their original
 *  order.  For example, if N is 3 and L contains [9, 2, 7, 12, 8, 1, 6],
 *  then the result is [ [9, 12, 6], [7, 1], [2, 8] ].  The operation
 *  is destructive (it may destroy the original list) and creates no new
 *  IntList objects (it will, of course, create new IntList2 objects).
 */
static IntList2 dslice (IntList L, int N) {
    IntList2 result;

    result = null;
    for (int k = N-1; k >= 0; k -= 1) {
        result = new IntList2 (first (L, k, N, true), result);
        IntList next = first (L, k, N, false);
        separate (L, k, N);
    }
    return result;
}

static IntList first (IntList L, int k, int N, boolean wantEqual) {
    while (L != null && (L.head % N == k) == wantEqual)
       L = L.tail;
    return L;
}

static void separate (IntList L, int k, int N) {
    IntList lastK, lastNotK;
    lastK = lastNotK = null;
    while (L != null) {
       IntList next = L.tail;
       if (L.head % N == k && lastK == null)
           lastK = L;
       else if (L.head % N == k)
           lastK = lastK.tail = L;
       else if (lastNotK == null)
           lastNotK = L;
       else
           lastNotK = lastNotK.tail = L;
       L.tail = null;
       L = next;
    }
}
```

5. The class `FilteredList` represents a read-only view of a `List` that selects only certain of its members. In this problem, you are to fill in part of its implementation. For example, if `L` is any kind of object that implements `List<String>` (that is, the standard `java.util.List`), then writing

```
List<String> FL = new FilteredList<String> (L, filter);
```

gives a list containing all items, x, in `L` for which `filter.test (`x`)` is true. Here, `filter` is of type `Predicate`:

```
interface Predicate<T> {
    boolean test (T x);
}
```

(Don't worry about that `Predicate<T>`, even though we haven't talked about it explicitly. It just means "For any type, `T`, ...") The object pointed to by `FL` above is supposed to be a *view* of `L`; when `L` changes, so does `FL` (since `FL` is a read-only view, we aren't going to worry about the other direction).

a. Fill in the indicated places below to achieve this effect. Do this "from scratch." That is, do not use the standard `AbstractList` or `AbstractSequentialList` classes.

```
public class FilteredList<T> implements List<T> {
    private List<T> L;
    private Predicate<T> filter;

    public FilteredList (List<T> L, Predicate<T> filter) {
        this.L = L; this.filter = filter;
    }

    public int size () {
        int n;  n = 0;
        for (T x : L)
            if (filter.test (x))
                n += 1;
        return n;
    }


    public T get (int k) {
        for (T x : L)
             if (filter.test (x))
                 if (k == 0)
                    return x;
                 else
                    k -= 1;
        throw new IndexOutOfBoundsException ();
    }
```

```
    public Iterator<T> iterator () {
        return new FilterIterator ();
    }

    private class FilterIterator implements Iterator<T> {
        private int n, lim;
        FilterIterator () {
            n = 0; lim = size ();
        }
        public boolean hasNext () {
            return n < lim;
        }
        public T next () {
            n += 1;
            return get (n-1);
        }
    }
}
```

b. Here is a second formulation of the `slice` problem from above. Fill it in, using the `FilteredList` abstraction from part (a) above. We suggest that the returned value be an `ArrayList<List<Integer>>`.

```
    /** A list of N lists such that list #k contains all the items in L
     *  that are equal to k modulo N, in their original order.  For
     *  example, if N is 3 and L contains [9, 2, 7, 12, 8, 1, 6],
     *  then the result is [ [9, 12, 6], [7, 1], [2, 8] ].  The operation
     *  is nondestructive (the original contents of L are not changed).
     */
    static List<List<Integer>> slice (List<Integer> L, int N) {
        ArrayList<List<Integer>> result = new ArrayList<List<Integer>> ();
        for (int k = 0; k < N; k += 1)
            result.add (new FilteredList (L, new ModFilter (k, N)));
        return result;
    }

    class ModFilter implements Predicate<Integer> {
        int k, N;
        ModFilter (int k, int N) { this.k = k; this.N = N; }
        public boolean test (Integer x) {
            return x % N == k;
        }
    }
```