1. Define the following terms
   a) analysis
   b) design
   c) postfix expression
   d) infix expression
   e) queue
2. Examine the following fragments of Java code and give a Big-Oh estimate of its running time. Assume that a is an array with N elements and that s is a string containing N characters. You may assume that both have been previously been initialized.

a) 
```java
for (int i = 0; i < 3; i++) {
    a[i] = a[a.length - (1 + i)];
}
```

b) 
```java
for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < s.length(); j++)
        if (s.charAt(j) == 'z')
            a[i]++;
}
```

c) 
```java
for (int i = 0; i < 5*N; i++) {
    for (int j = i; j >= 0; j--)
        for (int k = 0; k < a.length; k++)
            a[k] = a[k] + i * j;
}
```

d) 
```java
int x = 0;
int y = (a.length - 1)/2;
int z = a.length - 1;
while (x <= z) {
    for (int i = x; i <= z; i++)
        a[i] = y;
    if (y%2 == 0)
        x = y + 1;
    else
        z = y - 1;
    y = (z - x) / 2;
}
```
3. Assume that we have defined the following classes as the basis for a linked list:

```java
public class SimpleList {
    ListNode head;
    ListNode tail;
}

public class ListNode {
    Comparable element;
    ListNode next;
}
```

Write a method called `findLargestInList` which takes a reference to a `SimpleList` object as a parameter and which returns a `Comparable` reference to the largest element in the list. If the list is empty, your method should return a null reference. The code for the `Comparable` interface is attached to the test.
4. Any implementation of a dynamic data structure such as a Stack or Queue must address the issue of allocating the memory for the storage of data as the structure grows. We have seen two implementations of the Queue, QueueAr and QueueLi, which use an array and a linked list respectively to store data elements. Discuss how each handles the issue of allocating the space for the data elements as the Queue grows. Which approach is more efficient in terms of running time, and why? (QueueLi is attached to the take-home part of the test. The relevant parts of QueueAr are at the end of this test.)
5. Consider the design of a program to catalogue a collection of music CDs. For each disc in the collection, we must keep track of the title, the list of artists who participated, the release date, and the record company that released the CD. We want to be able to display all CDs that feature a given artist, all that were released in a given year, or all that were released by a given record company. (Use the back of the page if you need extra room.)

   a) What operations would you need to provide for such a class? Briefly describe the purpose of each method you propose.

   b) What classes would you suggest for this design? For each class describe its primary “secret,” or the organizing principle which justifies its existence.
package DataStructures;

import Exceptions.*;

// QueueAr class
public class QueueAr implements Queue
{
    /**
     * Construct the queue.
     */
    public QueueAr()
    {
        theArray = new Object[ DEFAULT_CAPACITY ];
        makeEmpty();
    }

    /**
     * Insert a new item into the queue.
     * @param x the item to insert.
     */
    public void enqueue( Object x )
    {
        if( currentSize == theArray.length )
            doubleQueue();
        back = increment( back );
        theArray[ back ] = x;
        currentSize++;  
    }

    /**
     * Internal method to expand theArray.
     */
    private void doubleQueue( )
    {
        Object[ ] newArray;
        newArray = new Object[ theArray.length * 2 ];

        // Copy elements that are logically in the queue
        for( int i = 0; i < currentSize; i++, front = increment( front ) )
            newArray[ i ] = theArray[ front ];

        theArray = newArray;
        front = 0;
        back = currentSize - 1;
    }

    private Object[ ] theArray;
    private int currentSize;
    private int front;
    private int back;

    static final int DEFAULT_CAPACITY = 10;
}
package Supporting;

/**
 * Protocol for Comparable objects.
 * @author Mark Allen Weiss
 */
public interface Comparable {
    /**
     * Compare this object with rhs.
     * @param Rhs the second Comparable.
     * @return 0 if two objects are equal;
     * less than zero if this object is smaller;
     * greater than zero if this object is larger.
     */
    int compares( Comparable rhs );

    /**
     * Compare this object with rhs.
     * @param Rhs the second Comparable.
     * @return true if this object is smaller;
     * false otherwise.
     */
    boolean lessThan( Comparable rhs );
}