Test No. 1
(Wednesday, February 27, 2002)

Part I. Mini-essays. Answer the questions as completely as you can without sacrificing conciseness. The space provided should be a good indication of the length of answer required.

1. We used the “big-O” notation to express measures of complexity. More precisely, we defined three “O” notations: $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$. Discuss how these three notations relate to each other and how they were used to measure problem and method complexity. (10 pts.)
2. Discuss the advantages and disadvantages of the two “types” of list implementation seen thus far: array-based lists vs. linked-lists. Mention time and space complexity in your answer. (10 pts.)
Part II. Matching. Match each of the terms in the first column with the best term/phrase in the second column. Record your answers on the scantron sheet that was provided for this purpose. Do not forget to write and bubble out your name and the last four digits of your SSN on the scantron sheet as instructed. (1 pt. each)

| 1. computational complexity | a. time complexity matches problem complexity |
| 2. problem complexity | b. a measure of program efficiency |
| 3. intractable problem | c. best solution has exponential time complexity |
| 4. optimal method | d. a measure of problem's intrinsic difficulty |
| 5. unsolvable problem | e. no solution known to exist |

| 6. elementary statement | a. process of determining algorithm properties |
| 7. worst-case complexity | b. complexity amortized over all possible cases |
| 8. average-case complexity | c. a lower bound on complexity of all cases |
| 9. best-case complexity | d. requires $O(1)$ steps to execute |
| 10. algorithm analysis | e. an upper bound on complexity of all cases |

| 11. array | a. contiguous sequence of variables |
| 12. array components | b. exact duplicate of original object |
| 13. `ArrayIndexOutOfBoundsException` | c. must be implemented if class supports cloning |
| 14. clone | d. thrown when array index is improper |
| 15. `Cloneable` interface | e. variables that comprise an array |

| 16. abstract constructor | a. contains instance of original as a component |
| 17. dynamic arrays | b. synonymous with factory method |
| 18. `java.util.Vector` | c. have no predefined limit on size or capacity |
| 19. wrapper or adaptor class | d. encapsulates dynamic use of arrays |
| 20. direct access to array components | e. make array-based implementations efficient |

| 21. nodes | a. last node in this list references the first |
| 22. header nodes | b. dummy node that eliminates special cases |
| 23. circular list | c. space allocation done at object creation |
| 24. doubly-linked list | d. each node references previous and next node |
| 25. dynamic storage allocation | e. basic components of linked structure |

| 26. `private class` `Node` | a. encapsulates unbounded list functionality |
| 27. `public abstract class` `LinkedList` | b. inner class that provides node functionality |
| 28. dangling leak | c. reference to space that's been reclaimed |
| 29. dangling reference | d. principal advantage of linked lists |
| 30. no a priori bound on list size | e. garbage not recognized and reclaimed |
Part III. Problem Solving. Supply the required answer for the following problems.

1. A method has been tested and clocked with the results listed in the table below. Express the time complexity of the method in terms of the $\Theta(n)$ notation. Briefly justify the answer you came up with. (10 pts.)

<table>
<thead>
<tr>
<th>Problem size ($n$)</th>
<th>Time of execution ($t(n)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>471</td>
</tr>
<tr>
<td>20</td>
<td>821</td>
</tr>
<tr>
<td>30</td>
<td>1171</td>
</tr>
<tr>
<td>40</td>
<td>1521</td>
</tr>
<tr>
<td>50</td>
<td>1871</td>
</tr>
<tr>
<td>60</td>
<td>2221</td>
</tr>
<tr>
<td>70</td>
<td>2571</td>
</tr>
<tr>
<td>80</td>
<td>2921</td>
</tr>
<tr>
<td>90</td>
<td>3271</td>
</tr>
<tr>
<td>100</td>
<td>3621</td>
</tr>
</tbody>
</table>
2. An application requires that a list of String values be available. It has been determined that a linked-list implementation is best suited for the problem at hand. Given the following formal specification of a linked-list implementation, supply the formal specification of a concrete class that provides the functionality of such a String list: (10 pts.)

```java
public abstract class LinkedList implements Cloneable {

/**
 * Create an empty LinkedList.
 */
  protected LinkedList () {
      ...
  }

...

} // end of class LinkedList
```
3. Suppose that the list of String values has been modeled by a class called StringList. Suppose further that the following piece of code has been executed:

```java
StringList wordList = new StringList(); //create an empty list of String values
int i;
for ( i = 1; i <= 10; i = i+1 )
    wordList.append( "" + i ); //append String version of the int value i
```

a. After the execution of the piece of code above, what will be the value of the expression `wordList.get( 2 )`? (5 pts.)

   ```java
   wordList.get( 2 ) ⇔ __________________
   ```

b. Suppose that the following piece of code were executed immediately after execution of the piece of code above:

```java
int n = wordlist.size();
for ( i = 0; i < n/2; i = i+1 ) {
    String tmp = wordlist.get( i );
    wordList.set( i , wordlist.get( n-i-1 ) );
    wordlist.set( n-i-1 , tmp );
}
```

Now what would be the value of the expression `wordList.get( 2 )`? (5 pts.)

```java
wordList.get( 2 ) ⇔ __________________
```
Part IV. Coding. Supply the required Java code.

In order to make the append( ) method an O(1) operation, we considered a variant of the linked list implementation that kept a reference to the last node which in turn used its otherwise null "next" component to reference the first node. A typical list is thus illustrated below:

Below may be found a partial implementation of this variant:

```java
public abstract class LinkedList implements Cloneable {

    // Constructors:
    /**
     * Create an empty LinkedList.
     */
    protected LinkedList() {
        size = 0;  //new list instance starts out empty
        last = null;
    }...

    //private components
    private int size;  //number of list elements
    private Node last;  //last node in linked structure
}  //end of class LinkedList

In this variant, the append method is implemented as follows:

```java
/**
 * Append a new list element
 */
public void append( Object obj ) {
    if ( this.isEmpty() ) {  //new element is first list element
        last = new Node( obj );
        last.next = last;
    } else {  //general case: new element becomes last element in non-empty list
        Node tmp = new Node( obj );
        tmp.next = last.next;
        last.next = tmp;
        last = tmp;
    }
    size = size + 1;
}
```

The method is clearly O(1) since it contains no loop structures no matter how many list elements are already in the list.

(more)
a. Provide the implementation of the method `get()` for this variant of `LinkedList`: (10 pts.)

```java
/**
 * The list element at given index location
 * require:
 *   i >= 0 && i < this.size()
 */

public Object get( int i ) {
```
b. Provide the implementation of the method `clone()` for this variant of `LinkedList`. The “clone” should have its own linked-node structure as illustrated below:

As in the abstract class `BoundedList`, the method `clone()` will then be used to implement the `copy()` method like so:

```java
/**
 * A copy of the current list
 */
public LinkedList copy () {
    return (LinkedList) this.clone();
}
```

We give the formal specification of the method `clone()` below. You may use the next sheet for your implementation:

```java
/**
 * A copy of the current list; returns an Object in order to
 * override clone() method inherited from class Object
 */
public Object clone() {
    ...
}
```
/**
 * A copy of the current list; returns an Object in order to
 * override clone() method inherited from class Object
 */

public Object clone() {