Question 1. The code in Figure 1 defines a `StringCollection` class that contains some useful, but very poorly named, methods. Suppose that we have executed the statements:

```java
StringCollection words = new StringCollection(10);
words.add( "parts" );
words.add( "the" );
words.add( "all" );
words.add( "than" );
words.add( "greater" );
```

Indicate the String or int value that would be produced by each of the following expressions. Also provide a brief description of what the first method invoked in each expression does in general to a `StringCollection`.

a) `words.toString()`
b) `words.stern()`
c) `words.cubits()`
d) Now, suppose we execute the statement:

```
words.deport( "all" );
```

Describe the string that would be returned by evaluating the expression

```java
words.toString()
```

after the statement that invokes `deport` and describe the general operation performed by the `deport` method.

Question 2. Consider the following set of class definitions involving inheritance:

```java
public class Child {
    public Child( ) { }
    public int descendants() {
        return children();
    }
    public int children() { return 0; }
}

public class Parent extends Child {
    private final int MAX_FAMILY = 70;
    private int familySize = 0;
    private Child[] children = new Child[ MAX_FAMILY ];
    public Parent( ) { }
    public void addChild( Child newBorn ) {
        if ( familySize < MAX_FAMILY ) {
            children[ familySize ] = newBorn;
            familySize++;
        }
    }
    public int children() { return familySize; }
}
```
public class StringCollection {

    private int size = 0;
    private String[] contents;

    public StringCollection(int maxSize) {
        contents = new String[maxSize];
    }

    public void add(String element) {
        if (size < contents.length) {
            contents[size] = element;
            size = size + 1;
        }
    }

    public String toString() {
        String result = "";
        for (int i = 0; i < size; i = i + 1) {
            result = contents[i] + " " + result;
        }
        return result;
    }

    public void deport(String target) {
        int i = 0;
        while (i < size - 1 && !contents[i].contains(target)) {
            i = i + 1;
        }
        if (contents[i].contains(target)) {
            contents[i] = contents[size - 1];
            size = size - 1;
        }
    }

    public String stern() {
        return contents[size - 1];
    }

    public int cubits() {
        int result = 0;
        for (int i = 0; i < size; i = i + 1) {
            result = contents[i].length() + result;
        }
        return result;
    }
}

Figure 1: Code for StringCollection class
public Child getChild( int birthOrder ) {
    if ( birthOrder < familySize ) {
        return children[ birthOrder ];
    } else {
        return null;
    }
}

public class GrandParent extends Parent {

    public GrandParent( ) {
    }

    public int descendants() {
        int total = 0;
        for ( int r = 0; r < children(); r++ ) {
            total = total + getChild( r ).descendants() + 1;
        }
        return total;
    }
}
Suppose that the following code using this classes has been executed in some method:

```java
Child john2 = new Child();
Child jim = new Child();
Parent joan = new Parent();
joan.addChild( jim );
joan.addChild( john2 );

Child johnjr = new Child();
GrandParent mary = new GrandParent();
mary.addChild( joan );
mary.addChild( johnjr );
```

a) What value would be produced by the invocation

```java
mary.descendants()
```

if it occurred after the code shown was executed?

b) Would the assignment

```java
Child newKid = mary;
```

be legal in this context. If so, what value would be produced by the invocation

```java
newKid.descendants()
```

if it occurred after the code shown was executed?

c) Would the assignment

```java
GrandParent grandMa = joan;
```

be legal in this context. If so, what value would be produced by the invocation

```java
grandMa.descendants()
```

if it occurred after the code shown was executed?

**Question 3.**

Table 1 shows the results of executing the main loop of Dijkstra’s shortest path algorithm several times to build a forwarding table for the router named A. Unfortunately, the table is incomplete and much of the information about the neighbors of the routers disappeared during a freak transcription accident involving cod liver oil and a small rodent.

Please follow the steps of Dijkstra’s Algorithm to complete the entries in the **Best Route Length** and **First Step** columns of the table. The rows in which the **connections to neighboring routers** rows are completely empty are the places where the information disappeared. You should leave these table entries blank. You will discover you do not need this information to complete the **Best Route Length** and **First Step** columns of the table if you simply follow algorithm 1 as described below. Assume that all edges have positive integer lengths.

<table>
<thead>
<tr>
<th>Cities</th>
<th>Best Route Length</th>
<th>First Step</th>
<th>Status</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>C</td>
<td>Adjacent</td>
<td>K: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H: 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>J: 2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>C</td>
<td>Known</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Shortest Path Table
Algorithm 1 **Dijkstra’s Shortest Path**

1. Mark starting point as *KNOWN* with length 0
2. Identify each neighbor of start as *ADJACENT*
3. Set first step of each neighbor of start to itself
4. Set route length of each neighbor to first step distance
5. While you don’t know the shortest path to all the cities:
   6. Select adjacent city with shortest 'best route' value
   7. Identify that adjacent city with shortest best route as *KNOWN*
   8. Mark neighbors of new *KNOWN* city that were *DISTANT* as *ADJACENT*
   9. Update path lengths and record first steps to *ADJACENT* neighbors of new *KNOWN* city.