Consider this implementation of the size() method for a bit vector implementation of an NSet:

```java
public class BitVecNSet implements NSet {
    private byte[] internal;    // The array of bytes, used as a bit vector. */
    private int range;          // One greater than the largest number than can be stored. */

    public int size() {
        int count = 0;
        for (int i = 0; i < internal.length - 1; i++) { // first, outer loop
            byte current = internal[i];
            for (int j = 0; j < 8; j++) { // second, inner loop
                count += current & 1;
                current >>= 1;
            }
        }
        byte current = internal[internal.length - 1];
        for (int j = 0; j < range % 8; j++) { // third loop
            count += current & 1;
            current >>= 1;
        }
        return count;
    }
}
```

Let \( N \) be the size of the universe the set is drawn from (note \( \text{range} = N \)) and let \( A \) be the conceptual set.

1. Write a loop invariant for the first, outer for loop that explains what \( \text{count} \) is and relates it to \( i \), \( \text{internal} \), and/or the conceptual set.

2. Write a loop invariant for the inner for loop that explains \( \text{current} \) and relates \( \text{current} \) to \( \text{count} \), \( i \), \( j \), \( \text{internal} \), and/or the conceptual set.

(You are not asked to do anything with the third loop)
3. Consider this implementation of the `decreaseKeyAt()` method for the implementation of a heap:

```java
public class IntHeap {

    /**
     * The array containing the internal data of the heap.
     */
    private int internal[];

    protected void decreaseKeyAt(int i) {
        int lIndex = left(i);
        int rIndex = right(i);
        while (lIndex < heapSize &&
               (internal[i] < internal[lIndex]) ||
               (rIndex < heapSize &&
                internal[i] < internal[rIndex])) {
            int greatestIndex = internal[i] < internal[lIndex] ?
                lIndex : i;
            greatestIndex =
                rIndex < heapSize && internal[greatestIndex] < internal[rIndex] ?
                rIndex : greatestIndex;
            int temp = internal[i];
            internal[i] = internal[greatestIndex];
            internal[greatestIndex] = temp;
            i = greatestIndex;
            lIndex = left(i);
            rIndex = right(i);
        }
    }
}
```

Write a precondition for this method and an invariant for the loop that explains how the variable `i` is used, specifically what assumptions are made and maintained about the heap in relation to index `i`.

4. What is the running time of counting sort (as a big-Oh category) on an array of size `n` with values in the range `[0, m)` for some whole number `m`?

5. What is an abstract data type? Specifically, what makes it abstract?

6. How do lists and stacks differ? How do sets and maps differ? For any two distinct ADTs `x` and `y` that we have defined, how to `x` and `y` differ? How do they all differ from the data structures we have studied?

7. If you needed to write a quick implementation of set and had implementations of list, map, and bag at your disposal, which would you use, and what would be the key insight of your strategy? (Recommended: Write out the code for an implementation of `WhateverSet`, and be able to articulate why you chose whatever and how you’re using it.)
8. Give one advantage each for array-based implementations and linked implementations, in comparison with each other.

9. Explain a circumstance in which an adjacency-matrix implementation has some advantages over adjacency-list.

10. Suppose we have the following interface for graphs, where vertices are identified by number:

```java
public interface Graph {
    int numVertices();
    // return the vertices adjacent to v
    Iterable<Integer> adjacents(int v);
}
```

Suppose further you have a class that implements the `Graph` interface above as a directed graph, with the following public signatures:

```java
public class DirectedGraph implements Graph {
    public DirectedGraph(int numVertices);
    public int numVertices();
    public Iterable<Integer> adjacents(int v);
    // add an edge from u to v
    public void addEdge(int u, int v);
}
```

Note that the constructor takes only the number of vertices; edges are added later using the `addEdge()` method.

Write a class that implements the `Graph` interface above as an undirected graph, reusing the class `DirectedGraph`. Like `DirectedGraph`, your class should have a constructor that takes only the number of vertices and a method `void addEdge(int u, int v)`.

11. Suppose we have the following interface for graphs, where vertices are identified by number:

```java
public interface Graph {
    int numVertices();
    // return the vertices adjacent to v
    Iterable<Integer> adjacents(int v);
}
```

One can iterate over all edges of graph `g` with the following nested loops:

```java
for (int u = 0; u < g.numVertices; u++)
    for (int v : g.adjacents(u))
        // do something with edge (u, v)
```

Assume the body of the inner loop is constant.

a. What is the running time for this code as a big-oh category in terms of `V` and `E` if `g` is implemented using adjacency lists?
b. What is the running time for this code as a big-oh category in terms of $V$ and $E$ if $g$ is implemented using an adjacency matrix?

12. In the following graph, highlight the edges that form a minimum spanning tree and label them to indicate an order in which they would be added to the tree using Kruskal’s algorithm.

13. Using the same graph as before, highlight the edges that form a minimum spanning tree and label them to indicate an order in which they would be added to the tree using Prim’s algorithm.

14. Explain how Dijkstra’s algorithm finds an SSSP tree in one round of relaxations.
public interface List<E> {
    E get(int index);
    int size();
}

public class ArrayList implements List<String> {

    private String[] internal;

    public ArrayList(String[] items) {
        internal = new String[items.length];
        for (int i = 0; i < items.length; i++)
            internal[i] = items[i];
    }

    public String get(int index) {
        return internal[index];
    }

    public int size() { return internal.length; }
}

public class LinkedList implements List<String> {

    private class Node{
        String datum;
        Node next;
        public Node(String datum, Node next) {
            this.datum = datum;
            this.next = next;
        }
    }

    Node head;

    public LinkedList(String[] items) {
        head = null;
        for (int i = items.length - 1; i >= 0; i--)
            head = new Node(items[i], head);
    }

    public String get(int index) {
        Node current = head;
        for (int i = 0; i < index; i++) current = current.next;
        return current.datum;
    }

    public int size() {
        int count = 0;
        for (Node current = head; current != null; current = current.next)
            count++;
        return count;
    }
}