Part 1: Selection Sort

The SelectionSorter class implements a selection sort on an array of strings. It is missing the minimumPosition() method that returns the index position of the smallest element in the array.

class SelectionSorter
{
    public SelectionSorter(String[] anArray)
    {
        array = anArray;
    }

    public void sort()
    {
        for (int i = 0; i < array.length - 1; i++)
        {
            int minPos = minimumPosition(i);
            swap(minPos, i);
        }
    }

    private void swap(int i, int j)
    {
        String temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private String[] array;
}

1. (10 points) Complete the coding of the minimumPosition method in the box below:

    private int minimumPosition(int from)
    {
        
    }
2. (5 points) If we found that 10,000 strings items could be sorted by a selection sort in 5 seconds, how many seconds would it take to sort 100,000 items? (This is an $O(n^2)$ algorithm.)

3. (5 points) The Merge sort is an $O(n \log n)$ algorithm. If an array of 2000 items could be sorted in 500 milliseconds, about how many milliseconds would be required to sort an array of 10,000 items?

Part 2: Binary Search

The BinarySearcher class in the box to the right implements a binary search on an array of integers. The search method finds a value in a sorted array, using the binary search algorithm. Input parameter: v = the value to search. Returns the index at which the value occurs, or –1 if it does not occur in the array.

```java
public class BinarySearcher {
    public BinarySearcher(int[] anArray) {
        array = anArray;
    }

    public int search(int v) {
        int low = 0;
        int high = array.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int diff = array[mid] - v;
            if (diff == 0)
                return mid;
            else if (diff < 0)
                high = mid + 1;
            else
                low = mid - 1;
        }
        return -1;
    }

    private int[] array;
}
```

1. (10 points) Rewrite and correct the lines in the shaded area. Write your corrected version in the box at the bottom of this page.
2. (5 points) Suppose you want to use a Binary search on an array of objects that support the
**Comparable** interface. Complete the `search()` method in the following box so that Comparable objects
can be searched. *(note: Comparable has both `compareTo()` and `equals()` methods.)*

```java
public int search(Comparable v)
{
    int low = 0;
    int high = a.length - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;

        if( a[mid].equals(v) )
            return mid;
        else if (a[mid].compareTo(v) < 0)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

**Part 3: Java Collection Classes**

1. (10 points) Write statements that do the following: Declare an ArrayList named `myList`. Create an
   Iterator named `I` for `myList`. Write a loop that displays `myList` on the console **in reverse order**, using
   the Iterator `I` to traverse the array.

```java
Part 3: Java Collection Classes

1. (10 points) Write statements that do the following: Declare an ArrayList named `myList`. Create an
   Iterator named `I` for `myList`. Write a loop that displays `myList` on the console **in reverse order**, using
   the Iterator `I` to traverse the array.
```
2. (3 points each) Write Java statements that do the following.
   a. Declare a LinkedList object named myList:

   ```java
   LinkedList myList;
   ```

   b. Create a ListIterator object named iter that references myList:

   ```java
   ListIterator iter;
   ```

   c. Add the name "Adam" to the beginning of myList:

   ```java
   myList.addFirst("Adam");
   ```

   d. Add the name "Steve" to the end of myList:

   ```java
   myList.addLast("Steve");
   ```

   e. Insert "Bill" at index position 1 in myList:

   ```java
   myList.add(1, "Bill");
   ```

   f. Write a single statement that displays "yes" on the console if the name "Steve" can be found in
      the list. Do not use a loop:

   ```java
   System.out.println(myList.contains("Steve"));
   ```

   **Part 4: Implementing a Linked List**

```java
class Term {
    Term(int coeff, int expon) {
        coefficient = coeff;
        exponent = expon;
    }

    public int compareTo(Term T2) {
        return new Integer(exponent).compareTo(
            new Integer(T2.exponent));
    }

    int coefficient, exponent;
    Term link;
}
```

```java
class Polynomial {
    void addTerm(Term T) {
    }

    Term find(Term T) {
    }

    public String toString() {
    }

    // this is the only instance field
    Term lstHead = new Term(0,0);
}
```

The Term class (shown above) represents a single term of a polynomial. Both the coefficient and
exponent are assumed to be positive integers. The Polynomial class (above) holds a linked list of Term
objects, in which the first Term is a dummy header node.
Do the following:
1. (10 points) Implement the addTerm() method so that it adds a new Term to the end of the linked list.

```java
void addTerm(Term T)
{
}
```

2. (10 points) Implement the find() method so that it searches for the first Term in the list that has a matching exponent. If the search is successful, return a reference to the Term; otherwise, return null.

```java
Term find(Term T)
{
}
```