Graded Exercise 2

CSC144 Software Architecture

06 April 2018

1. (a) Which sorting algorithm needs memory sufficient to hold two copies of the list to be sorted?
   (b) Which sorting algorithm repeatedly searches from right to left through the already sorted part of the list to find the position to which the first element in the unsorted part of the list should be moved?
   (c) Which sorting algorithm repeatedly searches from left to right through the still unsorted part of the list to find the position of the element with the smallest value?

   (a) merge sort
   (b) insertion sort
   (c) selection sort

2. Recall that...

   - $2^x \times 2^x = 2^{2x}$
   - $2^y / 2^x = 2^{y-x}$
   - $\log_2 2^x = x$
   - $2^{10} \approx$ one thousand
   - $2^{20} \approx$ one million
   - $2^{30} \approx$ one billion
   - $2^{40} \approx$ one trillion
   - one year $\approx$ 30 million seconds

   Now, suppose that...

- you have a list of 2^{30} items to sort
- your computer can execute 2^{30} instructions per second

(a) How long will it take to complete the job with an algorithm that requires the execution of N^2 instructions?

(b) How long will it take to complete the job with an algorithm that requires the execution of $N \log_2 N$ instructions?

(a) $2^{30} \times 2^{30} = 2^{60} \approx$ one million trillion instructions. 2^{60} instructions $/2^{30}$ instructions per second $= 2^{30} =$ one billion seconds. One billion seconds $\approx$ 30 years.

(b) $2^{30} \times 30 \approx$ thirty billion instructions. $30 \cdot 2^{30}$ instructions $/2^{30}$ instructions per second $= 30$ seconds.

3. Complete this stub method.

- Find the index of the smallest integer in that part of a list of integers that begins at a specified position
- *startingIndex* is the position of that part of the list in which to begin the search.
- *data* is the list in which to search.
- The method returns the index of the smallest integer in that part the list that begins the specified position.

```
public static int positionOfMinimum( int startingIndex,
        List<Integer> data ) {

    return 0;
} // positionOfMinimum( int, List<Integer> )
```

```
public static int positionOfMinimum( int startingIndex,
        List<Integer> data ) {

    int bestGuessSoFar = startingIndex;
    for( int i = startingIndex + 1; i < data.size(); i++ ) {
        if( data.get(i) < data.get(bestGuessSoFar) ) {
            bestGuessSoFar = i;
        } // if
    } // for
    return bestGuessSoFar;
} // positionOfMinimum( int, List<Integer> )
```

4. Complete this stub method.

   - This method exchanges the values of the elements at positions i and j in a list.
   - *i* is an index in the list and in the interval [0, data.size() - 1].
   - *j* is an index in the list and in the interval [0, data.size() - 1].
   - *data* is the list that contains the two elements.

```
public static void swap( int i, int j,
        List<Integer> data ) {

} // swap( int, int, List<Integer> )
```

```
public static void swap( int i, int j,
        List<Integer> data ) {

    int temp = data.get(i);
    data.set(i, data.get(j) );
    data.set(j, temp );
} // swap( int, int, List<Integer> )
```

5. Complete this stub method.

   - This method sorts a list of integers using the selection sort algorithm.
   - *data* is the list to be sorted.

```
public static void selectionSort( List<Integer> data ) {
    for( int i = 0; i < data.size(); i++ ) {

      // TO-DO: Add calls to positionOfMinimum()
      // and swap() here.

    } // for
} // selectionSort( List<Integer> )
```

```
public static void selectionSort( List<Integer> data ) {
    for( int i = 0; i < data.size(); i++ ) {
        int j = positionOfMinimum( i, data );
        swap( i, j, data );
    } // for
} // selectionSort( List<Integer> )
```

6. Complete this stub method by writing the second part of the condition for the continuation of the looping.

   - This method finds the position in the sorted part of a list in which to insert the next element.
   - This will be the last element (in a search from right to left) that is greater than or equal to a given element.
   - *index* is the position at which to begin the left to right search and is also the position of the element to which all other elements in the search will be compared.
   - *data* is the list in which to search.
   - This method returns the index of the insertion point.

```
public static int positionOfGE( int index, List<Integer> data ) {
    int nextValue = data.get( index );
    int i = index;
    while( i > 0 /* TO-DO: Write second part of condition. */ ) {
        i = i - 1;
    } // while
    return i;
} // positionOfGE( int, List<Integer> )
```

```
public static int positionOfGE( int index, List<Integer> data ) {
    int nextValue = data.get( index );
    int i = index;
    while( i > 0 && data.get(i - 1) > nextValue ) {
        i = i - 1;
    } // while
    return i;
} // positionOfGE( int, List<Integer> )
```

7. Complete this stub method.

   - This method creates a hole in a list into which to insert an element in the next step of an insertion sort.
   - i is the index of the element to be inserted ($0 <= j <= i <= data.size - 1$).
   - j is the position at which the element is to be inserted ($0 <= j <= i <= data.size - 1$).

```
public static void createHole( int i, int j,
        List<Integer> data ) {

    for( int k = i; k > j; k-- ) {

      // TO-DO: Assign a new value to the k-th element of data.

    } // for
    data.set( j, null );
} // createHole( int, int,  List<Integer> )
```

```
public static void createHole( int i, int j,
        List<Integer> data ) {

    for( int k = i; k > j; k-- ) {
        data.set( k, data.get(k - 1) );
    } // for
    data.set( j, null );
} // createHole( int, int,  List<Integer> )
```

8. Complete this stub method with these statements:

   (a) data.set(j, temp);
   (b) createHole(i, j, data);
   (c) **int** j = positionOfGE(i, data);
   (d) **int** temp = data.get(i);

   - This method sorts a list using the insertion sort algorithm.
   - *data* is the list to be sorted.

```
public static void insertionSort( List<Integer> data ) {
    for( int i = 1; i < data.size(); i++ ) {

      // TO-DO: Place 4 statements in the right order here.

    } // for
} // insertionSort( List<Integer> )
```

```
public static void insertionSort( List<Integer> data ) {
    for( int i = 1; i < data.size(); i++ ) {
        int j = positionOfGE( i, data );
        int temp = data.get( i );
        createHole( i, j, data );
        data.set( j, temp );
    } // for
} // insertionSort( List<Integer> )
```

9. Complete this stub method.

   - Merge the two parts of a list (when those two parts are already sorted).
   - The two parts of the list will contain consecutive elements elements of the list.
   - The two parts need not make up the whole list—there may be other elements before and/or after these two parts.
   - *prefixStart* is the index of the first element in the first part of the list.

- *suffixStart* is the index of the first element in the second part of the list.
- *suffixEnd* is the end of the list element in the second part of the list.
- *data* is the list.
- This method returns the original list with the specified two parts now sorted.

```
public static void merge( int prefixStart,
        int suffixStart, int suffixEnd, List<Integer> data ) {
    int i = prefixStart;
    int j = suffixStart;

    List<Integer> result = new ArrayList<>();

    while( i < suffixStart && j <= suffixEnd ) {
        if( data.get(i) < data.get(j) ) {
            result.add( data.get(i) );
            i++;
        } // if
        else {
            result.add( data.get(j) );
            j++;
        } // else
    } // while

    // TO-DO: Write two more while loops here.

    for( int k = 0; k < result.size(); k++ ) {
        data.set( k + prefixStart, result.get(k) );
    } // for

} // merge( int, int, int, List<Integer> )
```

```
public static void merge( int prefixStart,
        int suffixStart, int suffixEnd, List<Integer> data ) {
    int i = prefixStart;
    int j = suffixStart;

    List<Integer> result = new ArrayList<>();
```

```
        while( i < suffixStart && j <= suffixEnd ) {
            if( data.get(i) < data.get(j) ) {
                result.add( data.get(i) );
                i++;
            } // if
            else {
                result.add( data.get(j) );
                j++;
            } // else
        } // while

        while( i < suffixStart ) {
            result.add( data.get(i) );
            i++;
        } // while

        while( j <= suffixEnd ) {
            result.add( data.get(j) );
            j++;
        } // while

        for( int k = 0; k < result.size(); k++ ) {
            data.set( k + prefixStart, result.get(k) );
        } // for

    } // merge( int, int, int, List<Integer> )
```

10. When does the recursion stop in the execution of this method?

```
    public static void mergeSortHelper( int i, int j,
            List<Integer> data ) {
        if( i == j ) {
        } // if
        else if( j - i == 1 ) {
            merge( i, j, j, data );
        } // else if
        else {
            int k = (i + j)/2;
            mergeSortHelper( i, k - 1, data );
            mergeSortHelper( k, j, data );
            merge( i, k, j, data );
        } // if
    } // mergeSortHelper( int, int, List<Integer> )
```

The method repeatedly divides a list into two smaller lists. It then calls itself, first with one of the sublists, and then with the other. When these two recursive calls return, the method merges the two (now sorted) sublists.

Recursion stops when the divisions result in a sublist that has two (or fewer) elements.