JAVA COURSEWORK 2 (REPORT)

1838838 (MARLEY SUDBURY)

1. Introduction

In this report I will present an overview of the design and implementation of the programs created in this coursework, an overview of the efficiency of the insertion sort and merge sort algorithms which were implemented for part B, a user guide to those implementations and instructions on how to run the code I've written.

2. Design and Implementation

The implementation for part A was very simple. In `main(String[] args)` I loaded the two files which were supplied on Learning Central, `GPT2.txt` and `stopwords.txt`. Then I fed these files into a method I wrote called `removeStopwords(File GPT2, File stopwords)`, as per the instructions. This method in turn calls another method I wrote called `File_to_List(File file)`, which takes the file given and divides the contents into an `ArrayList<String>`. This then gets returned and used by the `removeStopwords()` method to remove any words in `GPT2.txt` that are contained in `stopwords.txt`. This is done using the `ArrayList.contains()` method which returns a `Boolean` variable.

For part B, the first step was to create an implementation of the insertion and merge sort algorithms. I did this using pseudo code from the lecture slides as a reference.

The insertion sort was much simpler to implement as it only involved one method, which didn't use recursion. Once both algorithms were implemented correctly, I had to implement a counter for all moves and swaps occurring and also a timer for every 100 items sorted. Again, this was very simple for the insertion sort, but quite complicated for the merge sort. Therefore, I used global variables for the merge sort which weren't required for the insertion sort. For the merge sort, the sorted elements were counted when they were all merged together at the end.

```
ArrayList<String> words = new ArrayList<String>(S);
int length = words.size();
for (int i = 1; i < length; i++) {
    String item = words.get(i);
    words.remove(i);
    int j = i - 1;
    while (j >= 0 && words.get(j).toLowerCase().compareTo(item.toLowerCase()) > 0) {
        swaps++;
        words.add(j+1, words.get(j));
        words.remove(j);
        j--;
    }
}
```

The above is a snippet from the function `Insertion_Sort(ArrayList<String> S)`. `S` is copied to another `ArrayList<String>` called `words`. Then it iterates over every element in the list and moves it to the right until it find an element bigger than itself. This comparison is done using the `String.compareTo(String)` method.

```
public static ArrayList<String> Merge_Sort(ArrayList<String> S, int p, int r) {
    // Sorts an ArrayList of type String using the merge sort algorithm
    if (p < r) {
        int q = (p + r) / 2;
        ArrayList<String> list1 = new ArrayList<String>(Merge_Sort(S, p, q));
        ArrayList<String> list2 = new ArrayList<String>(Merge_Sort(S, q+1, r));
        return Merge(list1, list2);
    } else {
        ArrayList<String> list1 = new ArrayList<String>();
        list1.add(S.get(r));
        return list1;
    }
}
```

The above is the function `Merge_Sort(ArrayList<String> S, int p, int r)`. The function is recursive, with a base-case of returning a list with only one element if `p >= r)` or otherwise calling itself with different values of `p` and `q` and then returns the result of `Merge(list1, list2)`.

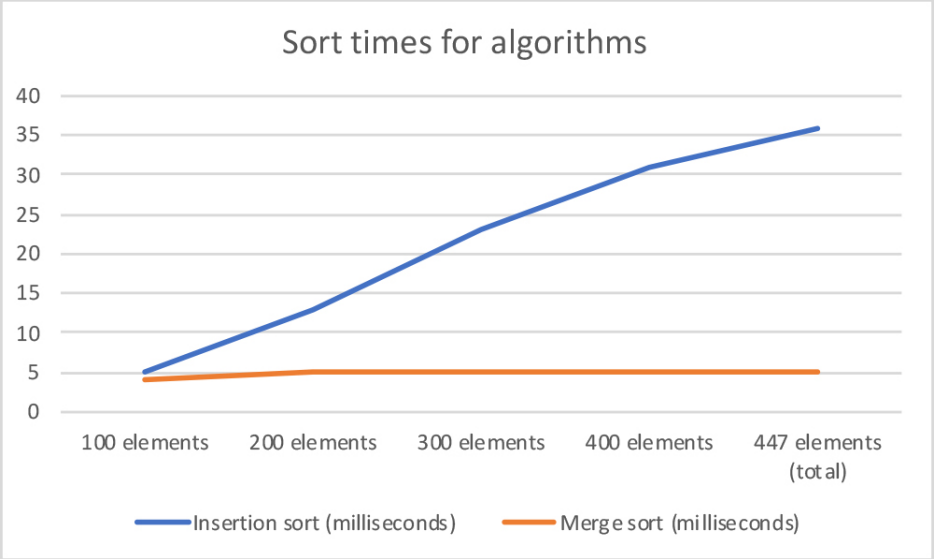
```
public void enqueue(Object theElement)
{
   if ((rear + 2) % queue.length == front) {
      Object[] newQueue = new Object[queue.length*2];
      int i = 1;
      while (!isEmpty()) {
         newQueue[i] = dequeue();
         i++;
      }
      front = 0;
      rear = queue.length - 1;
      queue = newQueue;
   } else {
      rear = (rear + 1) % queue.length;
   }
   queue[rear] = theElement;
}
```

The above is the function `enqueue(Object theElement)`. If the queue is full, it creates a new array with twice the length, adds all the elements from the previous array, and then replaces the old one with the new. Otherwise it moves the rear pointer along one and then in either case it adds `theElement` to the end of the array.

```
public Object dequeue()
{
   if (isEmpty()) {
      return null;
   } else {
      front = (front + 1) % queue.length;
      Object toReturn = queue[front];
      queue[front] = null;
      return toReturn;
   }
}
```

The above is the function `dequeue()`. If the array is empty then it returns `null`, otherwise it increments the `front` pointer and returns the object pointed to. It also sets the contents of that index to `null`.

3. Timing Results



No. of elements sorted	*100 elements*	*200 elements*	*300 elements*	*400 elements*	*447 elements (total)*
Insertion sort (milliseconds)	5	13	23	31	36
Merge sort (milliseconds)	4	5	5	5	5

Above you can see the time taken to sort the elements with each algorithm. Merge sort is quicker than insertion sort in each case, although they are very similar at 100 elements.

Algorithm	Swaps
Insertion sort	53809
Merge sort	1674

Above you can see the swaps used by each algorithm to sort the 447 elements. Insertion sort performs many more swaps than merge sort. In fact the merge sort took about 3.11% of the swaps by insertion sort, showing it is more efficient in this case.

4. User Guide for Algorithms

When using these implementations of these algorithms, here are some things you should know.

The insertion sort takes as a parameter a single `ArrayList<String>`, which is `words`. This list is then sorted using the insertion sort algorithm. This function returns a sorted version of the list; the original list should not be affected.

The merge sort takes as parameters the `ArrayList<String> S`, as well as an `int` for the start of the list (`p`) and for the end of the list (`r`). These values must be given to the function because it works recursively by calling itself with different values for these two parameters. This function also returns an `ArrayList<String>`. Again, the original list order should not be affected.

5. How to Run the Code

From the command line, you can compile the work for parts A and B with `javac StopWords.java` and run it with `java StopWords`. This will then display the timing results of the two sorting algorithms, and the swaps they made.

For part C, compile with `javac MyArrayQueue.java` and run with `java MyArrayQueue`. This will then display the results of the instructions that were present in the skeleton code I adapted, i.e. the elements being removed.