The merge sort algorithm

James Stanier

April 12, 2008

Contents

1 In-depth commenting for MergeSort.java

Section 1: ***Background***

The *merge sort* algorithm is a well-known, highly used algorithm for sorting in computer science, mostly due to its ease to understand and its $O(n\ log\ n)$ running time. The algorithm consists of two parts, which I have split into two classes – Merge and MergeSort – to reinforce the fact that they are two separate parts. First, we'll look at merge sort.

```
public ArrayList<Integer> mergeSort(ArrayList<Integer> l) {
```

The method takes an ArrayList as *input*. This is the (in this implementation) l of integers that we want to sort into the correct order.

Section 2: ***Variable initialisation***

Some variables are declared before we start. *split1* and *split2* refer to the two sub-arrays that we will break our input into. *sorted1* and *sorted2* are two arrays that will contain our sorted permuations. Don't worry – this will be explained as we go along! Finally, *sorted* is the array in which the result will be stored, and this will be returned.

```
ArrayList<Integer> split1 = new ArrayList<Integer>();
ArrayList<Integer> split2 = new ArrayList<Integer>();
ArrayList<Integer> sorted1 = new ArrayList<Integer>();
ArrayList<Integer> sorted2 = new ArrayList<Integer>();
ArrayList<Integer> sorted = new ArrayList<Integer>();
```

With these variables initialised, it is now possible to look at the main algorithm.

Section 3: ***An important check***

Below is an important piece of code:

```
if(l.size() <= 1) {
   return l;
}
```

If the array that has been input is either empty or of size 1, then we can just return it – it is already sorted! This isn't just common sense; it prevents errors from occurring should this have been left unchecked.

Section 4: ***Splitting the input***

The first part of the algorithm involves splitting the input into two parts. There is no sorting going on here – just simply halving it down the middle, as can be seen by checking to see if $i < l \div 2$.

```
else {
   for(int i = 0; i < l.size(); i++) {
      int x=l.get(i);
```

```
if(i < l.size()
```

At the end of this for loop, the whole of *l* will have been iterated over, with the first half residing in *split1* and the second in *split2*.

Section 5: *The magic of recursion*

This is the part that makes merge sort so powerful: *recursion*. Here, the algorithm calls itself on the two halves of the input. These will be halved again, and again, and again...until the check at the top of the algorithm fails. This was that the input must be greater than 1 in size, remember?

```
sorted1 = mergeSort(split1);
sorted2 = mergeSort(split2);

sorted = new Merge().merge(sorted1,sorted2);

return sorted;
```

At the time we get to the *return* statement in the top-most merge sort method, recursive calls have occurred so that the original input has been broken down into singular lists just one element. Pairs at each level in the *computation tree*, starting at the bottom, are then merged by the merge algorithm into the correct order. This occurs all the way up the tree until we arrive at the top, with all elements in one list in the correct order.

1.1 Variable grid for MergeSort.java

Every identifer used in the program appears here in this section, along with the line number upon which it was declared.

Type	Name	Line number
ArrayList<Integer>	l	13
ArrayList<Integer>	split1	29
ArrayList<Integer>	split2	30
ArrayList<Integer>	sorted1	31
ArrayList<Integer>	sorted2	32
ArrayList<Integer>	sorted	33
int	i	59
int	x	60

1.2 Keyword index for MergeSort.java

The keyword index contains a list of all defined keywords in in-depth comments, with the number of the section they appear in.

2 In-depth commenting for Merge.java

Section 1: ***Introduction***

The *merge* algorithm is a crucial part of the merge sort algorithm; it couldn't function without it. The idea behind behind merge is that it takes two data structures – in this case *ArrayList* – which are called l and m.

```
public ArrayList<Integer> merge (ArrayList<Integer> l, ArrayList<Integer> m) {
```

Above is the header of the actual merge method, showing the two inputs and its return type: another ArrayList.

Section 2: ***Variable initialisation***

We need to initialise three additional variables in this particular implementation. Firstly, another ArrayList is required called *result*, in which sorted elements will be added as the algorithm runs. The two integers, i and j, will be used to keep track of the current position in list l and m respectively.

```
ArrayList<Integer> result = new ArrayList<Integer>();
int i = 0;
int j = 0;
```

Now that these have been declared, we can move on to looking at the main merge algorithm.

Section 3: ***The merge algorithm***

The purpose of this while loop is to do the following. Firstly, we must only enter while our result array is smaller than the sum of the sizes of the two input arrays. This makes sense, and ensures we get no *null pointer* s. The outer if construct checks that the two indexes have not incremented past the size of our two input arrays. Again, this prevents *null pointer* exceptions. If the pointers are less, then we compare the i th and j th elements in both arrays. The smallest gets added to the result array and the index of that array is incremented.

```
while(result.size() < (l.size() + m.size())) {
   if(l.size() > i && m.size() > j) {
      if(l.get(i) < m.get(j)) {
         result.add(l.get(i++));
      }
      else {
         result.add(m.get(j++));
      }
   }
   else if (l.size() > i) {
      result.add(l.get(i++));
   }
   else if (m.size() > j) {
      result.add(m.get(j++));
```

```
    }
}
```

When the outer if guard fails, either $i \geq result.size()$ or $j \geq result.size()$. Therefore, this means that the other input array will have some elements left that need to be copied into *result*. The while loop continues executing, and each remaining iteration will add another 'left over' element to the result array, until the while's outer guard fails.

Section 4: *Returning and speed*

At this point in the code, the algorithm has finished. All that remains to do is *return* the result.

```
return result;
```

The overall efficiency of the merge algorithm is $O(n)$. This is because every time that merge will be called, it will be working on some divided portion of the tree – let is call the size of that portion m. At each level in the tree, all portions of $O(m)$ will always add up to $O(n)$. Therefore, $O(m) + O(m) + ... + O(m) = O(n)$.

2.1 Variable grid for Merge.java

Every identifer used in the program appears here in this section, along with the line number upon which it was declared.

Type	Name	Line number
ArrayList<Integer>	l	10
ArrayList<Integer>	m	10
ArrayList<Integer>	result	23
int	i	24
int	j	25

2.2 Keyword index for Merge.java

The keyword index contains a list of all defined keywords in in-depth comments, with the number of the section they appear in.

3 Code index for MergeSort.java

```
import java.util.*;
public class MergeSort {

    public ArrayList<Integer> mergeSort(ArrayList<Integer> l) {

      ArrayList<Integer> split1 = new ArrayList<Integer>();
      ArrayList<Integer> split2 = new ArrayList<Integer>();
      ArrayList<Integer> sorted1 = new ArrayList<Integer>();
      ArrayList<Integer> sorted2 = new ArrayList<Integer>();
      ArrayList<Integer> sorted = new ArrayList<Integer>();
      if(l.size() <= 1) {
        return l;
      }
      else {
        for(int i = 0; i < l.size(); i++) {
          int x=l.get(i);
          if(i < l.size() 2) {
            split1.add(x);
          }
          else {
            split2.add(x);
          }
        }
        sorted1 = mergeSort(split1);
        sorted2 = mergeSort(split2);
        sorted = new Merge().merge(sorted1,sorted2);
        return sorted;

      }
    }
}
```

4 Code index for Merge.java

```
import java.util.*;
public class Merge {

    public ArrayList<Integer> merge (ArrayList<Integer> l, ArrayList<Integer> m) {
      ArrayList<Integer> result = new ArrayList<Integer>();
      int i = 0;
      int j = 0;
      while(result.size() < (l.size() + m.size())) {
        if(l.size() > i && m.size() > j) {
          if(l.get(i) < m.get(j)) {
            result.add(l.get(i++));
          }
          else {
            result.add(m.get(j++));
          }
        }
        else if (l.size() > i) {
          result.add(l.get(i++));
        }
        else if (m.size() > j) {
          result.add(m.get(j++));
        }
      }
      return result;
    }
}
```