Dm507 – project 1

Emil Sjulstok Rasmussen - 050391

Hold: M1

Problem 1

In assignment 1 I have used an ArrayList to store the int's from the text file, while in assignment 2 I have just used an int array. I did this to show that both methods can be used, but I found it easier to use the int array instead of ArrayLists. In order to make the array the appropriate size, since normal arrays can't be expanded automatically, I traversed the input file an extra time to count the number of integers it contains. In the end we are trying to create an algorithm that runs in O(n lg(n)) time so it can be justified to traverse the list twice, since it only takes linear time.

The SimpleInv algorithm finds the inversions in an array by just compare each element in the list, which makes it very easy to implement. But this straightforward way doesn't make for a very good running time, since it uses two nested loops, it has a running time of $\Theta(n^2)$. In the following problems we should end up with an algorithm with a much better running time.

Problem 2

To implement MergeSort I pretty much just followed the pseudo code from the book. The only thing one needs to keep in mind is that in the pseudo code arrays start at index 1 and in java they start at index 0. There is also another thing that causes somewhat of a problem, in the pseudo code int's of value infinity is stored in the end of the two arrays but in java we can't use infinity. Therefore I use the command

integer.MaX_VALUE.

This sets the integer to the largest possible, which is something around 2,147,483,647. As long as the we don't deal with numbers this size we should be good.

In MergeSort line 89 I have used the command

int q =(int) Math.floor((p+r)/2);

I did this to make sure the value (p+r)/2 was rounded down properly. One could argue that just typecasting (p+r)/2 as an integer like so

int q = int (p+r)/2;

Would be enough, but I wasn't totally sure this would round down in the way I wanted. So I did it in a more tedious way but now I'm sure that q is rounded down properly.

Problem 3

To modify the merge algorithm to also count the number of inversions between the sub arrays L and R, can easily be done when we now that the two sub arrays are sorted. Since the two arrays are sorted, we know that if L[i]>R[i] then the rest of the elements in L will also be greater then R[i] and there will be L.length-1-i elements left in the array, with this in mind we can implement the counter zCounter in the following way.

We start by initiating the new variable.

Int zCounter = 0;

Then after line 17 in the pseudo code from the book we insert the following

zCounter = zCounter + (L.length-1-i);

Lastly I also want the algorithm to return the number of inversions, so after the last line in Merge I add

return zCounter;

This effectively gives us the number of inversions between the subarrays L and R, I will show that this implementation is correct in the following section.

Problem 4

If we use the suggested loop invariant, i.e. zCounter = z - inv(s,t), we start by showing that this loop invariant is true before the loop is initiated.

Initialization:

Before we enter the loop we need to show that zCounter is zero, to do this we use that z is always the same number because it is just the number of inversions between the two arrays and this number does not change, the number of inversions between s and t however does. We now the following about s and t

$p+i-1 \le s \le q$ $q+j \le t \le r$

Before we enter the loop i=1 and j=1 that is

$p \le s \le q$ $q+1 \le t \le r$

But this means that s spans the entire array L and t spans the entire array R so the number of inversions between these two is exactly z. This leaves us with zCounter = z-z = 0. Which is what we wanted to show.

Maintenance:

If we move a number from L into A it will form no inversion pairs with the numbers in R, because they are all higher and have a greater index. When we move a number from R into A it forms inversion pairs with all the remaining numbers in L therefore we increase zCounter with L.length-1-I, which is all the numbers in L. This way we are insured that all the inversions between the two arrays are found.

So when the loop increases j, t in the loop invariant decreases, that is zCounter gets bigger, because we subtract a smaller amount (inv(s,t)) from z, but the number of inversions that inv(s,t) have lost are now stored in zCounter.

Termination:

When the loop terminates we would expect that zCounter = z, because in the loop invariant we would have that inv(s,t) = 0 so zCounter = z -0

This is true because when we exit the loop we have moved all the numbers in the two arrays and therefore we have all the inversions.

The running time of the algorithm will be the same as before, i.e. $\theta(n \lg(n))$ since I only added a counter.

Problem 5

To create the desired algorithm we only need to use the tools we have already described. That is the modified merge, and for mergesort to return some integer, which will be the number of inversions in the given array.

If we implement the modified version of merge in MergeSort.java, it will find the inversions between all the sub arrays as I have explained I the previous problem. If we add up all these inversion it turns out that we get all the inversions in the array.

In order for this to work properly, we will have to return zCounter for all the subarrays, not just the last one. Modifying mergesort in the following way solves that problem.

```
public static int mergesort(int[] A,int p,int r){
        if(p<r){
          int q =(int) Math.floor((p+r)/2);
          int l = mergesort(A,p,q);
                int k = mergesort(A,q+1,r);
                int z = merge(A,p,q,r);
        return z+l+k;}
        return 0;
```

Here the variables l, k and z are just used to store the zCounter values for all the recursive calls. This I needed so I can add them up and return them.
When this is done, we need to print the output of mergesort, so instead of just calling MergeSort on the given array. This is quite easily done.

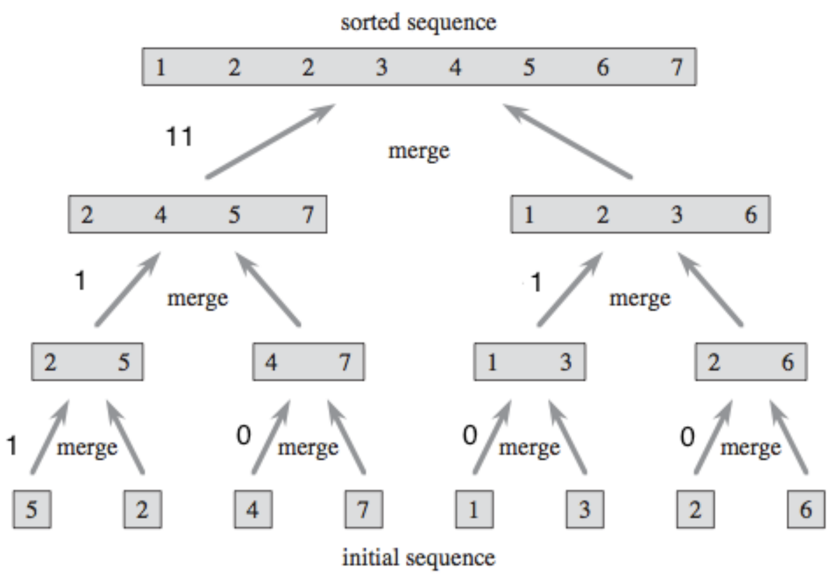
```
System.out.println(MergeSort(A,0,A.length-1));
```

This way the output will be the number of inversions in the array A.

The correctness of this algorithm will be shown in the next problem.

Problem 6

To show how my modified version of MergeSort works I have included this picture that shows how mergesort first breaks down the initial array, [5,2,4,7,1,3,2,6] into sub arrays of size one and then merging them all together via the modified merge procedure. Merge has been modified to count the inversions between two subarrays when merging them, as I have argued in the previous assignments. The number besides the two arrows that shows when two arrays are merging is the inversions between the two subarrays, i.e. zCounter.



The number of inversions in the shown array is 14, and if I add up all the inversions between the different sub arrays I get 11+1+1+1 = 14, which is what I expected. Now that I have “shown” that the algorithm works, I will try to use induction to prove it.

We now that when A.length≤1 there can be no inversions in it because it is either empty or just have one element. If A.length≤1 in FastInv.java then mergesort will return zero, because *p* would be greater then or equal to *r*. This takes care of the case where there is 1 or fewer elements in A.

I now assume that FastInv returns the correct number of inversions when A.length< k, where k = 2,3,4 … n-1, now we consider the case where A.length = n. FastInv will create two new sub arrays R and L, L consisting of the first half of A in sorted order and R consisting of the second half of A in sorted order. We can see that L<n and R<n by using induction we can conclude that the number of inversions calculated and stored in l and k are correct.

To show that we get all the inversions that are in A, we look at the three cases of inversions that are possible, all have that L[i]>R[j].

If i and j are in the first half of A then the inversions are calculated and stored in the variable l in the mergesort procedure

If i and j are in the second half of A then the inversions are calculated and stored in the variable k in the mergesort procedure

If i and j are in the first half and second half respectively then they are calculated by the modified merge procedure and stored in z.

Therefore this algorithm will find all the inversions in the array A, and return them, since it return l+k+z.

The running time of this algorithm will still be θ(n lg(n)), since we only add constant work, which is absorbed in the constant related to the big theta notation.

Problem 7

Here I have implemented the changes in mergeSort i found in problem 6 so it finds the number of inversions in the given array, A. Again I have used a simple int array to store the numbers, and found the appropriate size of the array by traversing the list ones before adding the numbers.

After implementing this algorithm I timed the two algorithms SimpleInv.java and FastInv.java when their input was an approximately 50.000 digit reversely sorted array. This yields a running time for SimpleInv that is around 40 seconds and a running time for FastInv that is around 30 miliseconds. This really shows how big a difference there is between the two algorithms, and how important choosing the right algorithm is.

Appendix

SimpleInv:

```
import java.util.*;
import java.io.*;
public class SimpleInv{
public static void main(String[] args){
ArrayList<Integer> A = new ArrayList<Integer>();
// Creates a new ArrayList named A that will be used to store the integers from
the given text file.
        Scanner sc = new Scanner(System.in);
        // Creates a new scanner called sc that will be used to handle the
input from the text file.
        File infile = new File(args[0]);
        try{
        Scanner s = new Scanner(infile);

        while(s.hasNext()){
            A.add(s.nextInt());
        }
    } catch(FileNotFoundException ex){
    System.out.println("File not found!");
    }
    // The above adds each integer from the tekst file, given as an argument
in the cmd, to the ArrayList A, if the given file does not exist an error message
will be thrown.

        int counter = 0;
        // Here a variable called counter is created and set to be zero. This
variable will count the number of inversions in the tekst file.


        for(int i=0; i<A.size()-1; i++){
            for(int j = i+1 ;j<A.size(); j++){
                if(A.get(i)>A.get(j)){
                counter++;
            }
            }

        }

        System.out.println(counter);
    // The two for loops runs through every number in A and checks if
A[i]>A[j] i.e. if there is an invertion, if there the counter is increased by one.
    }
}
```

MergeSort:

```
import java.util.*;
import java.io.*;
public class MergeSort{
        public static void main(String[] args){

        File infile = new File(args[0]);
        int counter= 0;
        int x = 0;
        try{
        Scanner s = new Scanner(infile);
        while(s.hasNextInt()){
                s.nextInt();
                counter++;
        }
} catch(FileNotFoundException e){
        System.out.println("File not found");
}
// The above traverses the file given as an argument in the cmd, and counts the
number of integers in it in the parameter counter


int A[] = new int[counter];
//Here the array of ints, A, that should hold the elements from the tekst file, is
created.


        try{
                Scanner sc = new Scanner(infile);
                while(sc.hasNextInt()){
                        A[x] = sc.nextInt();
                        x++;
                }
        } catch(FileNotFoundException ex){
                System.out.println("File not found");
        }
        // The above adds each element from the tekst file to the array A. If the
name given in the cmd is not a tekst file, an exception is thrown.



         mergesort(A,0,A.length-1);
        // Here the mergesort procedure is called with the array A, the start of A
and the end of a, as arguments.



        for(int i=0;i<A.length;i++){
                System.out.print(A[i] + " ");
        }
        //This for loop prints the sorted array A in one line.
         }

public static void merge(int[] A, int p ,int q, int r){
        int n1 = q-p+1;
        int n2 = r-q;
        int[] L = new int[n1+2];
        int[] R = new int[n2+2];
        // Creates the variables nedded in the procedure merge.

        for(int i=1; i<=n1; i++){
                L[i] = A[p+i-1];
                }
                // Adds the appropriate numbers to the subarray L

                for(int j=1; j<=n2; j++){
                        R[j] = A[q+j];
                }
                // Adds the appropriate numbers to the subarray R

                L[n1+1] = Integer.MAX_VALUE;
                R[n2+1] = Integer.MAX_VALUE;
        // Here we add, what corresponds to infinity, to the end of each of
the subarrays L and R, so the procedure nows when the arrays end.


        int i = 1;
        int j = 1;
        // Here the variables i and j are initially set to be one, which we
will need later
        for(int k = p; k<=r; k++){
            if(L[i] <= R[j]){
                A[k] = L[i];
                i = i+1;
            }
            else{
                A[k] = R[j];
                j = j+1;
            }
        }
    }



// The above for loop checks if L[i]<=R[j], and adds the smallest back to the
original array A, such that A will consist of its original elements in a sorted order.



public static void mergesort(int[] A,int p,int r){
    if(p<r){
      int q =(int) Math.floor((p+r)/2);
        mergesort(A,p,q);
        mergesort(A,q+1,r);
        merge(A,p,q,r);
    }
    // Here the procedure mergesort is created. It calls itself in such a way
that it creates two subproblems each of the original size divided by two.
}
}
```

FastInv:

```
import java.util.*;
import java.io.*;
public class FastInv{
        public static void main(String[] args){

        File infile = new File(args[0]);
        int counter= 0;
        int x = 0;
        try{
        Scanner s = new Scanner(infile);
        while(s.hasNextInt()){
                s.nextInt();
                counter++;
        }
} catch(FileNotFoundException e){
        System.out.println("File not found");
}
// The above traverses the file given as an argument in the cmd, and counts the
number of integers in it in the parameter counter


int A[] = new int[counter];
//Here the array of ints, A, that should hold the elements from the tekst file, is
created.


        try{
                Scanner sc = new Scanner(infile);
                while(sc.hasNextInt()){
                        A[x] = sc.nextInt();
                        x++;
                }
        } catch(FileNotFoundException ex){
                System.out.println("File not found");
        }
        // The above adds each element from the tekst file to the array A. If the
name given in the cmd is not a tekst file, an exception is thrown.


         System.out.println( mergesort(A,0,A.length-1));
        // Here the mergesort procedure is called with the array A, the start of A
and the end of a, as arguments, and is printed so we get the number of inversions
of in A as the output.
         }
```

```
public static int merge(int[] A, int p ,int q, int r){
        int n1 = q-p+1;
        int n2 = r-q;
        int[] L = new int[n1+2];
        int[] R = new int[n2+2];
        // Creates the variables nedded in the procedure merge.

        for(int i=1; i<=n1; i++){
                L[i] = A[p+i-1];
                }
                // Adds the appropriate numbers to the subarray L

                for(int j=1; j<=n2; j++){
                        R[j] = A[q+j];
                }
                // Adds the appropriate numbers to the subarray R



                L[n1+1] = Integer.MAX_VALUE;
                R[n2+1] = Integer.MAX_VALUE;
                // Here we add, what corresponds to infinity, to the end of each of
the subarrays L and R, so the procedure nows when the two arrays end.

                int zCounter = 0;
                int i = 1;
                int j = 1;
                // Here the variables i, j and zCounter are initially set to be one,
which we will need later
                for(int k = p; k<=r; k++){
                        if(L[i] <= R[j]){
                                A[k] = L[i];
                                i = i+1;
                        }
                        else{
                                A[k] = R[j];
                                j = j+1;
                                zCounter = zCounter + (L.length-1-i);
                        }
                }
return zCounter;        }
```

// The above for loop checks if L[i]<=R[j], and adds the smallest back to the original array A, such that A will consist of its original elements in a sorted order.
// Here zCounter counts the number of inversions between the two arrays that are being merged, as explained in assignment 5-6

```
public static int mergesort(int[] A,int p,int r){
        if(p<r){
          int q =(int) Math.floor((p+r)/2);
          int l = mergesort(A,p,q);
                int k = mergesort(A,q+1,r);
                int z = merge(A,p,q,r);
        return z+l+k;}
        // Here the procedure mergesort is created. It calls itself in such a way
that it creates two subproblems each of the original size divided by two, until
each sub array is of size one, then it calls merge on them.
        // l, k and z are here variables that hold zCounter for all the subproblems,
therefor we add them together and return them to get all the inversions in A.
return 0;
}
}
```