DM507 - Algoritmer og Datastrukturer Project 1

Christian Skjøth

Mat-Øk 280588

Morten Olsen

Mat-Øk 090789

19. marts 2012

Task 1 - Double for-loop

So, first we needed to create an instance of the class File using the filename passed as an argument when the program was called. Then we needed to build an array from this file. And finally we had to count the number of inversions in the array. So our main function ended up looking like this:

```
public static void main(String[] args) throws Exception{

/** create an instance of the class File from the given
 *  filename when the function is called  such that we are
 *  able to read data from this file
 */
        File fil= new File(args[0]);

/** create an array containing the integers in the given file
 */
        int[] dr= buildArrayFromFile(fil);

/** print the number of inversions in the array to the command prompt
*/
        System.out.println(countInversions(dr));
}
```

So, we need to make two methods in this program, that is "buildArrayFromFile"and "countInversions". The first of the two is quite simple, however since we do not know the size of the array in the file in advance, we have chosen to create an ArrayList into which the integers from the file provided as parameter are initially stored. Then an array with the same "size"as the ArrayList is created and the values from ArrayList are copied to the array which is then returned. Thus "buildArrayFromFile"ended up looking like this:

```
private static int[] buildArrayFromFile(File fil){
        List<Integer> list= new ArrayList<Integer>();
        try{
                Scanner s = new Scanner(fil);
                while (s.hasNextInt()){
                        int i=s.nextInt();
```

```
                        list.add(i);
                }

        }catch(FileNotFoundException e){

                System.out.println("File Not Found!");
                System.exit(-1);
        }
        int[] array=new int[list.size()];
        for (int i=0;i<list.size();i++){
                array[i]=list.get(i);
        }
        return array;
}
```

"countInversions"is extremely simple. Using the double for-loop, if n is the number of integers in the array, we iterate though the first $n-1$ integers of the array and for each integer k_i $(i = 1, 2, .., n)$, we iterate through the remaining $n - i$ integers checking whether these are bigger than k_i. If that is the case, the counter is incremented by 1. When the array has been fully processed, the counter is then returned to the main function and printed to the screen. This is the code for the countInversions function:

```
private static int countInversions(int[] dr){
        int count=0;

        try{
                for (int i=0; i<dr.length-1; i++){
                        for (int j=i+1; j<dr.length; j++){
                                if (dr[j]<dr[i]){
                                        count++;
                                }
                        }
                }

        }catch(Exception e){
                System.out.println("There were no numbers in the given file");
                System.exit(-1);
        }

        return count;
}
```

Task 2 - Using Pseudo Code

Our main function is very similar to the previous task and the buildArrayFromFile is exactly the same, so we are not going to elaborate on these. So we have 2 functions, mergeSort and Merge which are based on the pseudo code in the book. Let us start with mergeSort. The biggest challenge with this function was the secure that it actually returned an array to the main function. So we had to make Merge return an array which is the returned to main. Also, note that when the function is called recursively at some point one will end up processing only 1 integer. This is obviously already sorted and is therefore returned directly to the previous call of function mergesort. Finally, since we have chosen q to be an integer, we will get a number which is already rounded down following division.

```
private static int[] mergeSort(int[] dr, int p, int r){
        if (p<r){
```

```
                int q=(p+r)/2;
                mergeSort(dr, p, q);
                mergeSort(dr, q+1, r);
                return Merge(dr, p, q, r);
        } else {
                return dr;
        }
}
```

Then we had to implement Merge. This was pretty easy. The only thing we really had to do was to account for 0-indexing in java whereas the book uses 1-indexing and of course we had to make sure the sorted array is returned to mergeSort where Merge is called. The code is as stated below:

```
private static int[] Merge(int[] dr, int p, int q, int r){
        int llength = q - p + 1;
        int rlength = r - q;

        int[] L = new int[llength+1];
        int[] R = new int[rlength+1];

        for (int i=0; i<llength; i++){
                L[i]=dr[p+i];
        }
        for (int j=0; j<rlength; j++){
                R[j]=dr[q+j+1];
        }

        L[llength]=Integer.MAX_VALUE;
        R[rlength]=Integer.MAX_VALUE;

        int i=0;
        int j=0;

        for (int k=p; k<=r; k++){
                if (L[i]<=R[j]){
                        dr[k]=L[i];
                        i++;
                } else {
                        dr[k]=R[j];
                        j++;
                }
        }
        return dr;
}
```

Task 3

So the initial array is divided into 2 subarrays. Every time an integer from the R-subarray is lower than a non-∞ integer from the L-subarray, there is an inversion. Then the zCounter will be incremented by $q+1-(p+i)$. If we would want to state it in code it would be:

```
        int i=0;
        int j=0;
        int zCounter=0;

        for (int k=p; k<=r; k++){
                if (L[i]<=R[j]){
                        dr[k]=L[i];
```

```
                i++;
            } else {
                dr[k]=R[j];
                j++;
                zCounter += q+1-(p+i);
                }
            }
    }
```

Task 4

This algorithm is of course correct, since we were the ones to design it!
Anyways, as Initialization: Prior to the first iteration of the loop, we have $zCounter = i = j = 0$ which is the same as z minus the the number of inversions (s,t) in $A[p..r]$ with endpoints in $A[(p+i=p)..q]$ and $A[(q+j+1=q+1)..r]$ (i.e. the number of inversions in the entire array).

To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$, then $zCounter$ will not be incremented because $L[i]$ is thus smaller than or equal to every element in $A[(q+j+1)..r]$, so there are no inversions between $A[(p+i)]$ and $A[(q+j+1)..r]$. So i increments by 1 and thus $zCounter$ is still equal z minus the inversions (s,t) in $A[p..r]$ with endpoints in $A[(p+i)..q]$ and $A[(q+j+1)..r]$.
On the other hand suppose that $L[i] > R[j]$, then $zCounter$ will be incremented because all the elements in $A[(p+i)..q]$ are thus smaller than every element in $A[(q+j+1)..r]$ (because $A[(p+i)..q]$ is already sorted), so there are $q+1-(p+i)$ inversions between $A[(p+i)..q]$ and $A[(q+j+1)]$. So j increments by 1 and $zCounter$ increments by $q+1-(p+i)$ and thus $zCounter$ is still equal z minus the inversions (s,t) in $A[p..r]$ with endpoints in $A[(p+i)..q]$ and $A[(q+j+1)..r]$ and the loop invariant is maintained.

At termination $i = q-p+1$ and $j = r-(q+1)+1$ and thus $zCounter$ equals z minus the number of inversions (s,t) in $A[p..r]$ with endpoints in $A[(p+i = p+q-p = q+1)..q]$ which makes no sense and $A[(q+(r-(q+1))+1 = r+1)..r]$ which also does not make any sense. But we do know that $zCounter$ by this point is the number of inversions (s,t) in $A[p..r]$ with endpoints in $A[p..q]$ and $A[(q+1)..r]$ which is the number of inversions in the entire array $A[p..r]$ and so equals z because there are no remaining inversions.

The second part of this task (the runtime) is pretty well described on the bottom of page 30 and the top of page 31. Suppose we have n integers, the worst case maximum increase in runtime as opposed to the runtime of the version without z will be $c_1 + \frac{n}{2}(c_2)$ with c_1 being the cost of initializing $zCounter$, and c_2 the cost of incrementing $zCounter$ by $q+1-(p+i)$. But an additional runtime of $c_1 + \frac{n}{2}(c_2)$ does not change the Θ runtime stated in the book of the simplified version which is $\Theta(n)$.

Task 5

Okay, so since we already know how to calculate z by using Merge, we need only compute x and y and the sum of these 3 values will be the number of inversions

in the array. Now since every subarray can be treated as an independent array, we can continue to call mergeSort recursively on these subarray until we have arrays of length 1. Obviously these are sorted, so our "sub-x"and "sub-y"are set to 0. We then use Merge to calculate our "sub-z"and the sum of these 3 is the number of inversions in an array of length 2. This will again become a new "sub-x"and together with the corresponding "sub-y"and "sub-z", the sum of these 3 will become the total number of inversions in a subarray of size 3 or 4. If we keeping pairing these sums and calculating the corresponding z's, we should end up with the total number of inversions in the entire array. The algorithm would look like

1. check whether the given array has more than 1 element, if not return 0
2. divide the array into 2 subarrays
3. find the sum of inversions in each of these two subarrays by going to step 1 for each subarray
4. find the number of inversions between the two subarrays using the improved Merge
5. return the sum from step 4 + the number from step 5

Task 6

We will use induction. For the base case, consider an array of length 1. This is already sorted and does not contain any inversions, so the base case is correct. For the induction step, suppose that MergeSort will sort any array with less than n elements and return the correct number of inversions in these. So if we call MergeSort on an array of length n, it will then recursively call MergeSort on two arrays of length $n/2$. These are sorted correctly and the correct number of inversions are returned according to the induction hypothesis. So after recursion the two subarrays $A[p..q]$ and $A[q+1..r]$ are both sorted and the correct number of inversions in both are returned. We have already argued that our Merge works, and thus after this is executed, the array is sorted correctly and the sum of inversions in each subarray plus the number of inversions between the 2 is the total number of inversion in the entire array A. This concludes that this algorithm must be correct.

In terms of runtime. We have only added a constant amount of work to each procedure call and to each iteration of the last for-loop in the Merge procedure. Therefore the total running of the algorithm is the same as for merge sort, i.e. $\Theta(n \log n)$. But it is probably required to use the Master Theorem. So we will also do this. Lets denote the total runtime $T(n)$. Since each array will be recursively subdivided into 2 subarrays and then merged, we may write

$$T(n) = 2T(\frac{n}{2}) + f(n)$$

with $f(n)$ being the time of merging, i.e. $\Theta(n)$. We now see that the master theorem can be applied on $T(n)$ with $a = b = 2$. We now realize that

$$f(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$$

and therefore

$$T(n) = \Theta(n^{\log_2(2)} \log_2(n)) = \Theta(n \log_2(n))$$

Task 7

Everything in this implementation has already been discussed or is trivial, so we refer to the source code in the appendix.

Appendix

SimpleInv.java

```
import java.util.*;
import java.io.*;

public class SimpleInv{

/** main class. counts the number of inversions in an array contained
 *  in a file whose name is given when the function is called
 */
        public static void main(String[] args) throws Exception{

        /** create an instance of the class File from the given
         *  filename when the function is called  such that we are
         *  able to read data from this file
         */
                File fil= new File(args[0]);

        /** create an array containing the integers in the given file
         */
                int[] dr= buildArrayFromFile(fil);

        /** print the number of inversions in the array to the command prompt
        */
                System.out.println(countInversions(dr));
        }

/** creates an array containing the integers in the given file
 *  in the same order they were listed in the file, by parsing
 *  through the file and storing every integer encountered into an
 *  ArrayList.
 *  Then an array is created with the same size as the
 *  ArrayList into which the values of the ArrayList are copied.
 *  Finally the array containing the values of the files are returned
 */
        private static int[] buildArrayFromFile(File fil){
                List<Integer> list= new ArrayList<Integer>();
                try{
                        Scanner s = new Scanner(fil);
                        while (s.hasNextInt()){
                                int i=s.nextInt();
                                list.add(i);
                        }

                }catch(FileNotFoundException e){

                        System.out.println("File Not Found!");
                        System.exit(-1);
                }
                int[] array=new int[list.size()];
                for (int i=0;i<list.size();i++){
                        array[i]=list.get(i);
                }
                return array;
        }

/** returns the number of inversions using a double for loop
 */
        private static int countInversions(int[] dr){
                int count=0;
```

```
                try{
                        for (int i=0; i<dr.length-1; i++){
                                for (int j=i+1; j<dr.length; j++){
                                        if (dr[j]<dr[i]){
                                                count++;
                                        }
                                }
                        }

                }catch(Exception e){
                        System.out.println("There were no numbers in the given file");
                        System.exit(-1);
                }

                return count;
        }

}
```

MergeSort.java

```
import java.util.*;
import java.io.*;

public class MergeSort{
        public static void main(String[] args) throws Exception{
                File fil= new File(args[0]);
                int[] dr= buildArrayFromFile(fil);
                dr=mergeSort(dr, 0, dr.length-1);
                for (int i=0; i<dr.length; i++){
                        if (i==dr.length-1){
                                System.out.println(dr[i]);
                        }
                        else{
                                System.out.print(dr[i]+" ");
                        }
                }
        }

        private static int[] buildArrayFromFile(File fil){
                List<Integer> list= new ArrayList<Integer>();
                try{
                        Scanner s = new Scanner(fil);
                        while (s.hasNextInt()){
                                int i=s.nextInt();
                                list.add(i);
                        }

                }catch(FileNotFoundException e){

                        System.out.println("File Not Found!");
                        System.exit(-1);
                }
                int[] array=new int[list.size()];
                for (int i=0;i<list.size();i++){
                        array[i]=list.get(i);
                }
                return array;
        }

/** Divides the entire array using recursion and sorts the subarrays */
```

```
        private static int[] mergeSort(int[] dr, int p, int r){
                if (p<r){
                        int q=(p+r)/2;
                        mergeSort(dr, p, q);
                        mergeSort(dr, q+1, r);
                        return Merge(dr, p, q, r);
                } else {
                        return dr;
                }
        }

/** Sorts an array by dividing it into 2 subarray and then compares the elements to see which
  * smaller */

        private static int[] Merge(int[] dr, int p, int q, int r){
                int llength = q - p + 1;
                int rlength = r - q;

                int[] L = new int[llength+1];
                int[] R = new int[rlength+1];

                for (int i=0; i<llength; i++){
                        L[i]=dr[p+i];
                }
                for (int j=0; j<rlength; j++){
                        R[j]=dr[q+j+1];
                }

                L[llength]=Integer.MAX_VALUE;
                R[rlength]=Integer.MAX_VALUE;

                int i=0;
                int j=0;

                for (int k=p; k<=r; k++){
                        if (L[i]<=R[j]){
                                dr[k]=L[i];
                                i++;
                        } else {
                                dr[k]=R[j];
                                j++;
                        }
                }
                return dr;
        }
}
```

FastInv.java

```
import java.util.*;
import java.io.*;

public class FastInv{
        public static void main(String[] args) throws Exception{
                File fil= new File(args[0]);
                int[] dr= buildArrayFromFile(fil);
                System.out.println(mergeSort(dr, 0, dr.length-1));
        }

        private static int[] buildArrayFromFile(File fil){
                List<Integer> list= new ArrayList<Integer>();
                try{
                        Scanner s = new Scanner(fil);
```

```
                while (s.hasNextInt()){
                        int i=s.nextInt();
                        list.add(i);
                }

        }catch(FileNotFoundException e){

                System.out.println("File Not Found!");
                System.exit(-1);
        }
        int[] array=new int[list.size()];
        for (int i=0;i<list.size();i++){
                array[i]=list.get(i);
        }
        return array;
    }

/** returns the number of inversions in an array using recursion to divide the array
  * into subarrays from whom the number of inversions are returned
  */


    private static int mergeSort(int[] dr, int p, int r){
        if (p<r){
                int q=(p+r)/2;
                int ztot = mergeSort(dr, p, q) + mergeSort(dr, q+1, r);
                ztot += Merge(dr, p, q, r);
                return ztot;
        } else {return 0;}
    }

/** returns the number of inversion in an array */

    private static int Merge(int[] dr, int p, int q, int r){
        int llength = q - p + 1;
        int rlength = r - q;

        int[] L = new int[llength+1];
        int[] R = new int[rlength+1];

        for (int i=0; i<llength; i++){
            L[i]=dr[p+i];
        }
        for (int j=0; j<rlength; j++){
            R[j]=dr[q+j+1];
        }

        L[llength]=Integer.MAX_VALUE;
        R[rlength]=Integer.MAX_VALUE;

        int i=0;
        int j=0;
        int zCounter=0;

        for (int k=p; k<=r; k++){
                if (L[i]<=R[j]){
                        dr[k]=L[i];
                        i++;
                } else {
                        dr[k]=R[j];
                        j++;
                        zCounter += llength-i;
```

```
                        }
                    }
                    return zCounter;
                }
            }
```