COM2031 Advanced Algorithms, Autumn Semester 2019

Lab 3: Counting Inversions

Purpose of the lab

Your task is to code a solution to the problem of counting inversions (pairs of values out of order) in a list of values. You can use any programming language you like. A sample solution will be given in Java.

For the implementation, you should implement the core algorithm(s), but probably you will also need to implement the data structures that the algorithm(s) calculate on. You will most likely also want to test your algorithms – so will also need to write code that generates some sample data to test the algorithm(s), or manually produce unit tests.

You will notice that it is a fairly long way from the pseudo code in the lectures / book to a real implementation as you will need to create all the data structures and take care of special cases, parameter checking etc that a pseudo code representation dismisses hand-wavingly.

You will probably not finish the coding for this exercise within the 2hrs allocated to the lab. This is not unexpected. You will need to work on finalising the algorithms also after the lab to fully profit from the exercise. The lab serves to get you started, settle any questions and put you on the right track.

Always work through the exercise sheet in conjunction with the lecture slides / textbook!

Preamble

In the lecture we discussed that a list L of numbers has an inversion whenever i < j, but L[i] > L[j]. A straightforward brute force implementation of this is:

```java
/**
* Brute force implementation of inversion count Runtime O(n^2)
*
* Inversion are the number of swaps that bubbleSort has to execute in order
* to sort its input data.
*
* @param L
* @return
*/
public static int countInversionsBruteForce(final int[] L) {
	int inversions = 0;
	// for loops are such that always i < j
	for (int i = 0; i < L.length - 1; i++) {
		for (int j = i + 1; j < L.length; j++) {
			if (L[i] > L[j]) {
				inversions++;
			}
		}
	}
	return inversions;
}
```

Another implementation is directly based on BubbleSort, just by adding a variable that counts the number of swaps:

```
public static int bubbleSort(int[] L) {

      int count = 0;

      for (int j = L.length - 1; j > 0; j--) {
            for (int i = 0; i < j; i++) {
                  if (L[i] > L[i + 1]) {
                        swapNeighbours(L, i);
                        count++;
                  }
            }
      }
      return count;
}
```

Both these implementations have a running time of O(n^2).

Lab task: Counting Inversions

Your task for this lab is to implement an algorithm the counts inversions in O(n log n) and is based on MergeSort as presented in the lecture. You are provided with template code of a class "***InversionCount***". Your main task is to implement the "***countInversionsMergeSort***" method. You can start coding either by modifying your own implementation of MergeSort in a previous lab or by using the sample solution for that lab.

I suggest to proceed as follows (based on the sample solution for MergeSort), not necessary in the same linear order:

1. (Fundamental) Think carefully about at which place the actual counting of inversion across the two halves should happen within the MergeSort code, and how to implement it. Remember from the slides that the count of inversions for each item in the second list is the number of elements still in the first list "to jump over" (as per the lecture slides). This requires updates to the ***merge*** and ***MSort*** Method in the provided ***MergeSortGeneric*** class and the ***countInversionsMergeSort*** method in the ***InversionCount*** class template.

2. (Fundamental) Current methods merge and MergeSort return only the sorted arrays. Eventually however we now need them to also return the count of inversions for their respective snippet of the list: You could for eg define a nested class ***ReturnValue*** that both contains the list and the count value and let ***merge*** and ***MSort*** methods return such an object. You can update the provided ***MergeSortGeneric*** class to create an inner return class, and to update the ***merge*** and the ***MSort*** methods to return the defined class in task 2 to be eventually used in task 1.

3. (Fundamental) For a sanity check, amend the testing code so that it now also tests whether inversion counts from this MergeSort-based implementation agree with the brute force and the BubbleSort-based version. This is provided in the main method of ***InversionCount*** class template.