Optimization Project 1 - Comparing rankings

December 11, 2015

Students

Felix Gutmann
Marco Fayet
Max van Esso

1 Discussion of the problem

The problem is as follows: n items are ranked according to two lists. The first list is ordered 1 to n by convention. The second ranking is a permutation of 1,2, ... , n which we label as list A = a_1, a_2, ... , a_n. We want to count the number of inversions in A, which is the number of pairs of indices such that $i < j$ but $a_i > a_j$. The number of inversions is taken as a measure of dissimilarity between the two rankings.

In order to solve this problem, we will use a Divide-and-Conquer design principle, which consists of 3 parts:

1. Divide the input into smaller parts
2. Recursively solve the same problem on each of the smaller parts
3. Combine the solutions computed by the recursive calls to obtain the final solution.

This is essentially a reduction technique which splits the original problem into a combination of steps, with the overall effect of reducing complexity. Steps 1 and 2 follow an algorithm labeled Sort-and-Count, while step 3 follows an algorithm referred to as Merge-and-Count.

2 Proposed Algorithm

2.1 Sort and Count

The rationale for this method stems from the fact that counting the number of inversions by looking at each individual pair of ranking (i.e. by following the Naive Algorithm) would take exponential time with complexity O(n^2). In order to circumvent this, we divide the list A into two components of equal size $A_1 = a_1, \ldots, a_m$ and $A_2 = a_{m+1}, \ldots, a_n$. Each component is recursively sorted within itself so that the elements are ordered in increasing order. We count the number of inversions taking place in each sub-list and add them to a global counter r_t. This process gives us two sorted "sub-lists".

2.2 Merge and Count

This process will go through the sorted sub-lists A_1 and A_2. We use the notion of a pointer for each list, in order to identify which elements are being analyzed, and declare the front of a list to be its smallest element. We start with pointers at the beginning of each list.

1. If the first element of A_1 is smaller than the first element of A_2, we remove it from A_1, add it to a final list C, and move the A_1 pointer to the next element.
2. If the first element of A_1 is bigger, we remove the first element of A_2 to include it in C. Furthermore, we increase the inversion counter r_t by the number of remaining elements in A_1. This derives from the fact that A_1 is already sorted: if an element in A_2 is bigger than another element in A_1, it is bigger than all the other elements in A_1. We move the pointer to along to the next element of A_2.
3. We move along the elements of A_1 and A_2 until both sub-lists are cleared. As a final output we get a sorted list C as well a count for the number of inversions.

2.3 Complexity

If we label T(n) as the worst-case running time on input lists of size n, the algorithm will spend O(n) time dividing the list into two sublists, T($\frac{n}{2}$) time ordering each sub list (at most) and O(n) time combining the solutions into a final sorted list. Put together, the initial division of our procedure and the final merging step take linear time bounded by an overall complexity of O(n log n).

3 Proof of Correctness

The discussion that follows apply to lists with at least two elements. If the list contains only one element it has no inversion by definition and the algorithm will return that element.

Let us first start by differentiating between a total proof of correctness and a partial proof of correctness. The partial proof of correctness simply requires that if an answer is returned it will be correct. The total proof of correctness will perform the same function, while additionally requiring that the algorithm terminates.

Let's examine the possibility that our algorithm may not terminate, and show that this is not realistic:

Our algorithm will first divide our list into two sublists of $n \times \frac{1}{2}$ values (or $(n-1) \times \frac{1}{2} + 1$ and $(n-1) \times \frac{1}{2}$ values respectively in the event that n is an odd number). We then sort the two sub-lists within themselves in order to have all the values in a sub-list ordered. At this stage it is impossible for the algorithm to enter an infinite loop because the number of values in each sub-list is finite and the order classification has no ambiguities.

The algorithm will then start comparing the top values for each sub-list and place them in a sorted order into a third, final list via the above-mentioned process. Here too, it is impossible for the algorithm to get stuck in an endless loop because the comparison between two elements is essentially a binary decision where there are only two possible outcomes. Given that the two sub-lists are finite, the algorithm will eventually terminate. We can now turn our attention to whether the algorithm will provide the expected results or not.

Let us denote the original list of all unsorted numbers Z, the first ordered sub-list A, and the second ordered sub-list B.

I.

We consider the case when the first element of sub-list A, a_i, is bigger than the first element of sub-list B, b_j. When the Merge-and-Count algorithm compares a_i with b_j, it adds the number of elements of A (which we will denote as $|A|$) to the global counter of inversions. We claim that at this point, every element in A including and following a_i is indeed involved in an inversion pair with b_j.

To see this, we note that every element in A after a_i is bigger or equal than a_i because the sub-list was previously sorted. Equivalently, b_j is smaller than every following element in the sub-list B. Since b_j is smaller than a_i, it is also smaller than every following element of A.

Furthermore, we note that each element of A comes before each element of B in the original unsorted list Z. This follows directly from the Sort-and-Count process during which Z is divided in two parts, with A being on the left and B on the right.

Therefore, when $a_i > b_j$, and the Merge-and-Count adds $|A|$ to the total count of inversions, it is because there are that many inversion pairs involving b_j as the smaller element of the pair.

II. We claim that every inversion pair gets counted at least once in the algorithm.

The Merge-and-Count algorithm counts every inversion at least once. Suppose there is an inversion in the original unsorted list Z between elements in position i and j; that is, $Z_i > Z_j$. Let Z_i = a and Z_j = b.

When we divide Z into the sub-lists A and B, all elements in A are to the left of all the elements of B, and the elements in the union of the sub-lists A and B are the same elements that are in the entire set Z. At some point in the Merge-and-Count algorithm, the "pointer" (which denotes the element being examined) in sub-list A will be on a and the pointer in sub-list B will be on b. We note that the pointer marks the biggest element of the sub-list at that point (see previous section). We also note that the Merge-and-Count algorithm cannot exhaust all of the elements in sub-list A before b becomes the top element of B because $a > b$.

We then look at what happens when b is the top element in the sub-list B and a is still in A: b will be compared to an element a' belonging to A, where a' > b and a' < a. At that point, b is removed, and $|A|$ is added to the count accumulated in Merge-and-Count. The inversion pair {a, b} counts as a plus one in this update of the accumulated count in the call of Merge-and-Count for A and B. Note that any element a" that were to the left of a' in A is smaller than b (and as argued above, a" is to the left of b in Z) so {a", b} is not an inversion. Therefore, every inversion pair gets counted at least once in the algorithm.

III. We claim that no inversion pair gets counted more than once.

An inversion pair {a,b} can only contribute to a count when a belongs to A and b belongs to B in some invocation of Merge-and-Count. Since A and B are merged into a final list C during that invocation, the pair {a,b} can contribute to the inversion count in exactly one invocation of Merge-and-Count. When an inversion pair {a,b} does not contribute to the count during that inversion, b is immediately removed from B resulting in the pair a,b never being counted in that invocation of Merge-and-Count. Therefore, no inversion gets counted more than once.

4 Auxilliary remarks and output report

We wrote a piece of code which can be found in the file "Group7_Count_Inversion.R". In order to test it, we ran in on both a sorted array going from 100 to 1, as well as on the data provided in "permutation.txt". Both outputs are displayed on the screengrabs below.

The file "Group7_Count_Inversion.R" creates two functions: Naive.count.inversion() and DC.count.inversion(). The first one provides as an output the number of inversions based on the naive algorithm in $O(n^2)$ time. The DC.count.inversion() function initializes both the Sorting Count and Merging Count

algorithms before applying them to the input array to *in fine* return a list with the sorted array and the inversion count. When we append $inversion to the function call, the function outputs the number of inversions determined by the Divide and Conquer Algorithm in $O(n\ log\ n)$ time. The sorted list can be accessed by appending $array to the function call.
For the sake of compactness we chose not to display the sorted list of 1000 items in this document.

Figure I Output from sequence from 100 to 1

```
>
> sequence <- 100:1
>
> Naive.count.inversion(sequence)
[1] 4950
>
> DC.count.inversion(sequence)$inversion
[1] 4950
>
```

Figure II Output from permutation.txt

```
>
> permutation <- as.numeric(scan("Permutation.txt",what="",sep=","))
Read 1000 items
>
> Naive.count.inversion(permutation)
[1] 247504
>
> DC.count.inversion(permutation)$inversion
[1] 247504
>
```

5 Resources

Gusfield, Daniel - Counting Inversions in a Permutation, UC Davis: http://web.cs.ucdavis.edu/ gusfield/cs122f10/inversioncount

Kingsford, Carl - CMSC 451: Divide and Conquer, Department of Computer Scienc University of Maryland, https://www.cs.umd.edu/class/fall2009/cmsc451/lectures/Lec08-inversions.pd

Kleinberg, Jon and Tardos, Eva - Algorithm Design (2006), Cornell University

Precup, Doina - MergeSort proof of correctness, and running time, McGill University: http://www.cs.mcgill.ca/ dprecup/c lecture16.pdf

Zeh, Norbert - Design and Analysis of Algorithms class notes, Dalhousie University: https://web.cs.dal.ca/ nzeh/Teaching/3110/Notes/dnc.pdf