CS 3510 Algorithms Last updated: September 5, 2023

Lecture 5: Divide and Conquer Continued, Exam Review

Lecturer: Sahil Singla *Scribe(s): Allen Chang*

1 Exam Announcements

Exam 1 will be on **Thursday**, September 7th. The following topic lists will be covered:

1. Big-$\mathcal{O}$ notation
2. Recursions (the tree method, induction, and the Master theorem)
3. The divide and conquer strategy, (sorting, median finding, multiplication and matrix multiplication)

2 Number of Inversions

Problem. Given an array `a[0],a[1], ... ,a[n-1]`, the goal is to find the number of pairs `i<j` such that `a[i]>a[j]`.

Naively, we have an algorithm that computes the solution in $\mathcal{O}(n^2)$ time by checking all pairs (i, j) and counting the number of pairs in which $i < j$. However, we can do better: in fact, there exists an algorithm that solves this problem in $\mathcal{O}(n \log n)$ operations.

Given our desired runtime and our knowledge of divide and conquer algorithms, we can reverse engineer the algorithm. We can guess that the recursion is

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Thus, we would like to break up the problem into two pieces:

Figure 1: Dividing array into two subarrays of length $n/2$

Given the recurrence, we would like to only spend $\mathcal{O}(n)$ time on the "combine" step of the recurrence relation. One strategy we will attempt is to store more information (i.e. doing more work, not just counting the number of inversions!) which may help us achieve the desired running time.

In fact, one thing that we would like to is to also sort the array. Let the new algorithmic problem be: $T(n)$ is the worst-case time to calculate the number of inversions and sort n elements. Of course, we use the same recursion

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

except $\mathcal{O}(n)$ is now the time it takes to calculate the number of cross inversions and the time it takes to combine to sorted arrays. [1] It turns out that you can also count the number of cross inversions for two sorted arrays in $\mathcal{O}(n)$ time. [2]

The takeaway of this problem is that sometimes it is faster to do more than what is asked of the problem, which gives us other information (or extra assumptions, such as the subarrays being sorted).

[1] You may remember this as the merge operation in mergesort, which we have seen is $\mathcal{O}(n)$ time.

[2] This is left an an exercise for the reader, but the gist of the solution is to maintain two pointers across the two subarrays and count while incrementing the pointers.