Divide and conquer algorithms

M. Hassell

February 26, 2017

1 Introduction

These are the notes for the divide and conquer chapter in CLRS.

2 Maximum subarray problem

The maximum subarray problem seeks to find the maximum sum of contiguous elements in a given array. The naive approach is to compute all possible sums of subarrays where the starting element is "to the left" of the other element in the subarray. Since there are approximately

$$\binom{n}{2}$$

ways to choose valid starting and ending points for subarrays to consider, we'd end up with an $\mathcal{O}(n^2)$ algorithm for this problem. Using a recursive strategy we can cut this down to a $\mathcal{O}(n \log n)$ algorithm. Roughly, we can split the array into two subarrays of roughly equal sizes and then search for the maximum subarray in each. We have one rub with this: a maximum subarray may cross the "midpoint" of the original array, and therefore we would not pick it up with a simple recursive method. Instead, we need to devise a method to find if a maximum subarray crosses this artificial boundary we've introduced by recursing. We simply need to find a method to check for these "crossing" subarrays in a $\mathcal{O}(n \log n)$ manner as to not consume the speedup we gain by recursing in the first place. Fortunately, this is doable in linear time in the simplest possible way. I'll briefly describe this piece now.

To find a maximum crossing subarray, we simply do the obvious thing and pass the array under consideration, cleft it in twain, and then look at the subarrays that end at the midpoint (by counting backwards from the midpoint) and keeping track of their sums. We do similarly for arrays that start at the midpoint and work forward. This function returns the maximum sum it encountered for arrays that crossed the midpoint. Each for loop takes constant time (constant w.r.t. the size of the array), and therefore, for an array with n elements, we can expect this routine to run in $\mathcal{O}(n)$ time. Combining this with a recursion that is $\mathcal{O}(n \log n)$, our asymptotic complexity is $\mathcal{O}(n \log n)$, so we've improved

over the brute force approach. I'm not going to place the pseudocode here, as it is already in CLRS.

3 Strassen's Algorithm for matrix multiplication