Divide and conquer algorithms

M. Hassell

March 7, 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

$$\begin{pmatrix} n \\ 2 \end{pmatrix}$$

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). The total number of operations we perform in an array of size n is thus

$$\begin{aligned} &mid - low + 1 && \text{first loop,} \\ &high - mid && \text{second loop,} \end{aligned}$$

which in total gives $mid - low + 1 + high - mid = high - low + 1 = n$. So this bit runs 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

We now can move on to a means to improve the speed of matrix multiplication. The product of an $m \times n$ and $n \times p$ matrix is given by

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

To solve this in a recursive manner, we can split the matrices A, B, and C into quadrants and recursively multiply those four submatrices each generated by A and B and place them in C in the correct locations. For $n \times n$ arrays, the cubic complexity of recursive matrix multiplication comes from the need to multiply eight subarrays of size $n/2 \times n/2$. This can be seen from the following equations for the submatrices in $C = AB$:

$$C_{11} = A_{11}B_{11} + A_{12}B_{22}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B21, \quad C_{22} = A_{21}B12 + A_{22}B_{22}.$$

Our recurrence for this method is thus

$$T(n) = 8T(n/2) + \mathcal{O}(n^2),$$

where the lower order terms come from adding up the arrays. Solving the recurrence (with a yet-to-be-studied method!) gives us cubic complexity in the number of rows in the array. Strassen's method reduces the complexity with intermediate computations of the S_i and P_i arrays, which takes only $\mathcal{O}(n^2)$ time since they are largely additions and subtractions of matrices of size n. In computing the P_i, we recursively multiply seven matrices (instead of eight), which gives the recurrence

$$T(n) = 7T(n/2) + \mathcal{O}(n^2).$$

This brings the complexity of the matrix multiplication down to $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$. I'd love to understand the motivation for all of these S_i and P_i matrices, but it isn't covered in the book. I'll add this to the pile of things to improve on a second pass.

Some comments on my code. Right now it only works for square matrices that have a number of rows that is a power of two. I'll put this in the pile of things to improve in a second pass. Also, it seems that there is a fair amount of effort required on bookkeeping, such as extracting sub matrices of a given larger matrix for applying the recursion, as well as joining the sub matrices together after the recursion. I'm wondering if there is

a cleverer way to do this with only pointers, rather than building matrices and then operating on those (i.e. can we work in-place with the matrices). I also opted to work with vanilla arrays instead of using the std::array template just to keep up with C++ array functionality. The decay to a pointer is a bit of a headache, and using std::array would simplify some of the function calls (by reducing the number of arguments). This may be nice to do as well in a later version. Something else that would be nice would be the ability to overload the + operator to be able to directly add arrays. It seems this can't be done in C++ with built-in types in general. So perhaps a future iteration will define an array class that allows overloading basic arithmetic operations. It would be amazing to be able to overload the $*$ operator for matrix and scalar multiplication, and have the matrix multiplication overloading automatically deduce the types of the arrays (full, sparse, real, complex) and apply the most efficient method. But, at this point, we're basically re-inventing Matlab.