Divide and Conquer

We have seen one general paradigm for finding algorithms: the greedy approach. We now consider another general paradigm, known as divide and conquer.

We have already seen examples of divide and conquer algorithms: mergesort and Karatsuba’s algorithm. The idea behind mergesort is to take a list, divide it into two smaller sublists, conquer each sublist by sorting it, and then combine the two solutions for the subproblems into a single solution. These three basic steps – divide, conquer, and combine – were also the crux of Karatsuba’s algorithm, and in general lie behind most divide and conquer algorithms.

With mergesort, we kept dividing the list into halves until there was just one element left. In general, we may divide the problem into smaller problems in any convenient fashion. Also, in practice it may not be best to keep dividing until the instances are completely trivial. Instead, it may be wise to divide until the instances are reasonably small, and then apply an algorithm that is fast on small instances. For example, with mergesort, it might be best to divide lists until there are only four elements, and then sort these small lists quickly by insertion sort.

Maximum/minimum

Suppose we wish to find the minimum and maximum items in a list of numbers. How many comparisons does it take?

A natural approach is to try a divide and conquer algorithm. Split the list into two sublists of equal size. (Assume that the initial list size is a power of two.) Find the maxima and minima of the sublists. Two more comparisons then suffice to find the maximum and minimum of the list.

Hence, if $T(n)$ is the number of comparisons, then $T(n) = 2T(n/2) + 2$. (The $2T(n/2)$ term comes from conquering the two problems into which we divide the original; the 2 term comes from combining these solutions.) Also, clearly $T(2) = 1$. By induction we find $T(n) = (3n/2) - 2$, for $n$ a power of 2.

Median finding

Here we present a linear time algorithm for finding the median a list of numbers. Recall that if $x_1 \leq x_2 \leq \ldots \leq x_n$, 

9-1
the median of these numbers is $x_{\lceil n/2 \rceil}$. In fact we will give a linear time algorithm for the more general selection problem, where we are given an unsorted array of numbers and a number $k$, and we must output the $k$th smallest number (the median corresponds to $k = \lceil n/2 \rceil$).

Given an array $A[1, 2, \ldots, n]$, one algorithm for selecting the $k$th smallest element is simply to sort $A$ then return $A[\lceil n/2 \rceil]$. This takes $\Theta(n \log n)$ time using, say, MergeSort. How could we hope to do better? Well, suppose we had a black box that gave us the median of an array in linear time (of course we don’t – that’s what we’re trying to get – but let’s do some wishful thinking!). How could we use such a black box to solve the general selection problem with $k$ as part of the input? First, we could call the median algorithm to return $m$, the median of $A$. We then compare $A[i]$ to $m$ for $i = 1, 2, \ldots, n$. This partitions the items of $A$ into two arrays $B$ and $C$. $B$ contains items less than $m$, and $C$ contains items greater than or equal to $m$ (we assume item values are distinct, which is without loss of generality because we can treat the item $A[i]$ as the tuple $(A[i], i)$ and do lexicographic comparison). Thus $|B|, |C| \leq \lceil n/2 \rceil$ since $m$ is a median. Now, based on whether $k \leq |B|$ or $k > |B|$, we either need to search for the $k$th smallest item recursively in $B$, or the $(k - |B|)$th smallest item recursively in $C$. The running time recurrence is thus $T(n) \leq T(\lceil n/2 \rceil) + \Theta(n)$ (getting $m$ from the black box and comparing $m$ with every element takes $\Theta(n)$ time). This recurrence solves to $\Theta(n)$ and we’re done!

Of course we don’t have the above black box. But notice that $m$ doesn’t actually have to be a median to make the linear time analysis go through. As long as $m$ is a good pivot, in that it partitions $A$ into two arrays $B,C$ each containing at most $cn$ elements for some $c < 1$, we would obtain the recurrence $T(n) \leq T(cn) + \Theta(n)$, which also solves to $T(n) = \Theta(n)$ for $c < 1$. So how can we obtain a good such pivot? One way is to be random: simply pick $m$ as a random element from $A$. In expectation $m$ will be the median, but also with good probability it will not be near the very beginning or very end. This randomized approach is known as QuickSelect, but we will not cover it here. Instead, below we will discuss a linear time algorithm for the selection problem, due to [BFP+73], which is based on deterministically finding a good pivot element.

Write $A = a_1, a_2, \ldots, a_n$. First we break these items into groups of size 5 (with potentially the last group having less than 5 elements if $n$ is not divisible by 5): $(a_1, a_2, \ldots, a_5), (a_6, a_7, \ldots, a_{10}), \ldots$. In each group we do InsertionSort then find the median of that group. This gives us a set of group medians $m_1, m_2, \ldots, m_{\lceil n/5 \rceil}$. We then recursively call our median finding algorithm on this set of size $\lceil n/5 \rceil$, giving us an element $m$. Now we claim that $m$ is a good pivot, in that a constant fraction of the $a_i$ are smaller than $m$, and a constant fraction are bigger. Indeed, how many elements of $A$ are bigger than $m$? There are $\lceil n/5 \rceil / 2 \geq n/10 - 1$ of the $m_i$’s greater than $m$, since $m$ is their median. Each of these $m_i$’s are medians of a group of size 5 and thus have two elements in their group larger than even them (except for potentially the last group which might be smaller). Thus at least $3(n/10 - 2) + 1 = 3n/10 - 5$ elements
of $A$ are greater than $m$; a similar figure holds for counting elements of $A$ smaller than $m$. Thus the running time satisfies the recurrence

$$T(n) \leq T\left(\lceil n/5 \rceil \right) + T(7n/10 + 5) + \Theta(n)$$

which can be shown to be $\Theta(n)$ (exercise for home!). A slightly easier computation which doesn’t take the “plus 5” and ceiling into account that gives intuition for why this is linear time is the following. Suppose we just had

$$T(n) \leq T(n/5) + T(7n/10) + C_1 n.$$  Ignoring the base case and just focusing on the inductive step, we guess that $T(n) = C_2 n$. Then to verify inductively, we want

$$C_2 (n/5 + 7n/10) + C_1 n \leq C_2 n,$$

which translates into wanting $C_1 n \leq C_2 (n - 9n/10) \leq$, so we can set $C_2 = 10C_1$. The key thing that made this analysis work is that $1/5 + 7/10 < 1$.

**Strassen’s algorithm**

Divide and conquer algorithms can similarly improve the speed of matrix multiplication. Recall that when multiplying two matrices, $A = a_{ij}$ and $B = b_{jk}$, the resulting matrix $C = c_{ik}$ is given by

$$c_{ik} = \sum_j a_{ij} b_{jk}.$$  

In the case of multiplying together two $n$ by $n$ matrices, this gives us an $\Theta(n^3)$ algorithm; computing each $c_{ik}$ takes $\Theta(n)$ time, and there are $n^2$ entries to compute.

Let us again try to divide up the problem. We can break each matrix into four submatrices, each of size $n/2$ by $n/2$. Multiplying the original matrices can be broken down into eight multiplications of the submatrices, with some additions.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Letting $T(n)$ be the time to multiply together two $n$ by $n$ matrices by this algorithm, we have $T(n) = 8T(n/2) + \Theta(n^2)$. Unfortunately, this does not improve the running time; it is still $\Theta(n^3)$.

As in the case of multiplying integers, we have to be a little tricky to speed up matrix multiplication. (Strassen deserves a great deal of credit for coming up with this trick!) We compute the following seven products:

- $P_1 = A(F - H)$
- \( P_2 = (A+B)H \)
- \( P_3 = (C+D)E \)
- \( P_4 = D(G-E) \)
- \( P_5 = (A+D)(E+H) \)
- \( P_6 = (B-D)(G+H) \)
- \( P_7 = (A-C)(E+F) \)

Then we can find the appropriate terms of the product by addition:

- \( AE + BG = P_5 + P_4 - P_2 + P_6 \)
- \( AF + BH = P_1 + P_2 \)
- \( CE + DG = P_3 + P_4 \)
- \( CF + DH = P_5 + P_1 - P_3 - P_7 \)

Now we have \( T(n) = 7T(n/2) + \Theta(n^2) \), which give a running time of \( T(n) = \Theta(n \log^7 n) \).

Faster algorithms requiring more complex splits exist; however, they are generally too slow to be useful in practice. Strassen’s algorithm, however, can improve the standard matrix multiplication algorithm for reasonably sized matrices.

References