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 an example of divide and conquer algorithms: mergesort. 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 – 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.

Integer Multiplication

The standard multiplication algorithm takes time $\Theta(n^2)$ to multiply together two n digit numbers. This algorithm is so natural that we may think that no algorithm could be better. Here, we will show that better algorithms

exist (at least in terms of asymptotic behavior).

Imagine splitting each number x and y into two parts: $x = 10^{n/2}a + b, y = 10^{n/2}c + d$. Then

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

The additions and the multiplications by powers of 10 (which are just shifts!) can all be done in linear time. We have therefore reduced our multiplication problem into four smaller multiplications problems, so the recurrence for the time $T(n)$ to multiply two n-digit numbers becomes

$$T(n) = 4T(n/2) + O(n).$$

The $4T(n/2)$ term arises from conquering the smaller problems; the $O(n)$ is the time to combine these problems into the final solution (using additions and shifts). Unfortunately, when we solve this recurrence, the running time is still $\Theta(n^2)$, so it seems that we have not gained anything.

The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum ad + bc.* Now note that

$$(a+b)(c+d) = (ad + bc) + (ac + bd).$$

So if we calculate ac , bd, and $(a+b)(c+d)$, we can compute $ad + bc$ by the subtracting the first two terms from the third! Of course, we have to do a bit more addition, but since the bottleneck to speeding up this multiplication algorithm is the number of smaller multiplications required, that does not matter. The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2) + O(n),$$

and we find that $T(n) = n^{\log_2 3} \approx n^{1.59}$, improving on the quadratic algorithm.

If one were to implement this algorithm, it would probably be best not to divide the numbers down to one digit. The conventional algorithm, because it uses fewer additions, is probably more efficient for small values of n. Moreover, on a computer, there would be no reason to continue dividing once the length n is so small that the multiplication can be done in one standard machine multiplication operation!

It also turns out that using a more complicated algorithm (based on a similar idea) the asymptotic time for multiplication can be made arbitrarily close to linear– that is, for any $\varepsilon > 0$ there is an algorithm that runs in time $O(n^{1+\varepsilon})$.

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 $T(n) = \Theta(n^{\log 7})$.

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, as we will see in our second programming assignment.