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.

FFT

We will now consider another divide and conquer algorithm, the Fast Fourier transform (FFT). The Fast Fourier transform is a classic algorithm that has proven incredibly useful across a wide spectrum of applications, and as such remains in widespread use today.

The algorithm is based on a clever idea that is worth emphasizing before we begin. The key point is that it is very important to consider how to represent your data. In this case, he key to the Fast Fourier transform is an exceedingly clever and non-intuitive representation of polynomials.

To motivate the FFT, suppose we are keeping track of traffic low along a highway. We have a hardware device that samples the traffic every five minutes. That is, it outputs the number of cars that have passed in the last five minutes at five minute intervals. Let us call these samples $a_0, a_1, a_2, \ldots$. We may want to keep a running average of the traffic over the last half an hour. In that case, we would want to compute

$$\sum_{i=n}^{n+5} \frac{1}{6} a_i.$$

Alternatively, we may want to keep a *weighted average* of the traffic over the last half an hour. If we are judging traffic flow, it makes sense that the last five minutes are perhaps more important than a whole half hour ago. At the same time, it would be a mistake to discount the previous information entirely; if we look only at five minute intervals, we might overreact to random traffic fluctuations.

Suppose we use weighting coefficients $b_0, b_1, \ldots, b_5$. The most recent time interval will be weight by b_0, and so on. Then after a_n is output we wish to compute the weighted average c_n given by

$$c_n = \sum_{i=0}^{5} b_0 a_{n-i}.$$

This sum of products should look familiar. It is the coefficient of x^5 in the product of

$$\left(\sum_{i=0}^{5} a_i x^i\right)\left(\sum_{i=0}^{5} b_i x^i\right).$$

Indeed, more generally, if we look at the various coefficients of

$$\left(\sum_{i=0}^{k} a_i x^i\right)\left(\sum_{i=0}^{5} b_i x^i\right),$$

we can obtain several values of c_n at once.

This sort of processing occurs often, especially in digital signal processing. With this motivation, we see that it is of great interest to compute the product of two given polynomials extremely quickly.

We therefore focus on determining an efficient algorithm for multiplying two polynomials. That is, given $A(x)$ and $B(x)$, we wish to compute the coefficients of $C(x) = A(x)B(x)$. Without loss of generality, we will assume that C has degree $n-1$, where n is a power of 2. Also, we will think of A and B of also being degree $n-1$; the coefficients of some terms of A and B will just be 0. It is clear that we can compute the coefficients of C in time $\Theta(n^2)$ using the standard algorithm. But can we do better?

Thus far, we have been thinking of a polynomial as being represented by its coefficients. We think of the polynomial $A(x)$ as $A(x) = \sum_{t=0}^{n-1} a_t x^t$. But there are other ways to represent a polynomial. In particular, a polynomial of degree $n-1$ is completely defined by n points. For example, a line (a polynomial of degree one) is defined by any two points it passes through; a quadratic is defined by any three points it passes through.

Thus, instead of representing $A(x)$ by the sequence of coefficients $(a_0, a_1, \ldots, a_{n-1})$ (call this the *coefficient representation*), we could represent it by the sequence of values at points $x_0, x_1, x_2, \ldots, x_{n-1}$, namely $(A(x_0), A(x_1), A(x_2), \ldots, A(x_{n-1}))$ (call this the *point representation*). Now multiplying is easy in the point representation: we can compute $C(x_i) = A(x_i) \cdot B(x_i)$. Since to represent the polynomial C uniquely we will need to evaluate C at n points, we must have A and B evaluated at n points. But assuming that we use this representation, we can multiply $A(x)$ and $B(x)$ with just $O(n)$ multiplications!

The problem is that we may not want to keep the polynomial in the point representation. For example, it is hard to evaluate C at any new points when we represent it in this way, while this is a straightforward operation in the coefficient representation. Also, as we saw in our motivation section, we actually want as output the coefficients of C. Again, we see that different representations are useful in different situations! What we really want now is a quick way to move back and forth between these two representations.

In general, going back and forth between the two representations is not apparently quicker than $\Omega(n^2)$. To do

better, we need to be even trickier. *We must carefully choose which points to use in the point representation.* By choosing these points cleverly, we will be able to move back and forth between representations in time $O(n \log n)$.

The right points to evaluate A and B at turn out to be the solutions to $x^n = 1$. These solutions are the called the nth roots of unity. Now, in the real numbers, of course, there are only 2 solutions to this equation, namely 1 and -1. In the complex numbers, however, there are $2n$ different solutions. In fact, these solutions are all powers of a generator w, so the solutions can be written as $w, w^2, w^3, \ldots, w^{n-1}, w^n = 1$. Note also that $w^{n/2} = -1$! Finally, a very important fact is that for all roots w^i other than 1, when we add its powers $1 + w^i + w^{2i} + \ldots w^{(n-1)i}$, we must get 0. This is because

$$(1 + w^i + w^{2i} + \ldots w^{(n-1)i}) \cdot (1 - w^i) = 1 - w^{ni} = 1 - (w^n)^i = 0.$$

Since the first term in the product is not 0 (as long as $w^i \neq 1$), the first term is.

(Interestingly, we do not even have to work over complex numbers– we can work over other mathematical structures, such as finite fields. The point is we just need a generator– a w with $w^n = 1$, such that $w^k \neq 1$ for any k with $1 \leq k < n$.)

We now want to find a fast way to compute the values

$$A(w^j) = \sum_{i=0}^{n-1} a_i w^{ij}, \; j = 0, \ldots, n-1.$$

It turns out we can use a divide and conquer scheme. We divide the coefficients of A into two subsequences: the even subsequence $a_0, a_2, \ldots$ and the odd subsequence $a_1, a_3, \ldots$. Then

$$\begin{aligned} A(w^j) &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} w^{2ij} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} w^{2ij+j} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} (w^2)^{ij} + w^j \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} (w^2)^{ij} \\ &= E((w^2)^j) + w^j O((w^2)^j) \end{aligned}$$

Now we have reduced the problem of computing $A(w^j)$ to computing two smaller problems of size $n/2$ of the same kind – namely computing $E(w_2^j)$ and $O(w_2^j)$, where $w_2 = w^2$! Notice that this recursion will work because $1, w^2, w^4, \ldots$, which are the points where we will compute O and E, are $n/2$nd roots of unity. (Again, note that there are $n/2$ points; we have indeed reduced to the problem to two subproblems of half the size.) To obtain $A(w^j)$ from the subproblems, we do a multiply (by w^j) and an addition.

This is standard divide and conquer, with the recurrence:

$$T(n) = 2T(\frac{n}{2}) + cn,$$

where the work linear in n comes from putting the parts back together via multiplications and additions. Of course we immediately have $T(n) = O(n \log n)$. This process is known as the *Fast Fourier Transform*, or FFT.

(Aside: Another way of expressing this is we want to compute $A(x)$ at the n roots of unity. Note that

$$A(x) = A_0(x^2) + xA_1(x^2)$$

where A_0 is the polynomial with the "even" coefficients and A_1 is the polynomial with the odd coefficients. We need to find A_0 and A_1 at the points $1, w^2, w^4, \ldots = 1, v, v^2, \ldots$ where v is an $(n/2)$nd root of unity. This gives us the recursion above.)

We've shown how to go quickly from the coefficient representation to a point representation. How do we go back? Amazingly, this can be done with another FFT!

We start with a point representation $(C(1), C(w), C(w^2), \ldots, C(w^{n-1}))$. To invert, we think of these values as coefficients! (This is a tricky thought!) Then we do an FFT on these coefficients, using w^{-1} instead of w in the FFT. The jth value we obtain is

$$\begin{aligned} \sum_{i=0}^{n-1} C(w^i) w^{-ij} &= \sum_{i=0}^{n-1} (\sum_{k=0}^{n-1} c_k w^{ik}) w^{-ij} \\ &= \sum_{k=0}^{n-1} c_k (\sum_{i=0}^{n-1} w^{i(k-j)}). \end{aligned}$$

Now, consider the inner sum in the last line. If j and k are the same, the inner sum is just n. Otherwise, we get the sum of all the powers of w– but we know this sum equals 0! Hence we have

$$\sum_{i=0}^{n-1} C(w^i) w^{-ij} = n c_k,$$

so doing this inverse FFT returns to us the proper coefficients (scaled up by a factor of n).

To summarize, to multiply two polynomials A and B in $O(n \log n)$ time, we do the following:

- We first compute the FFT on the coefficients $(a_0, a_1, \ldots, a_{n-1})$ for A, and similarly for B. This puts us in the point representation.
- We compute $C(w^i) = A(w^i) \cdot B(w^i)$ for $i = 0, \ldots, n-1$. Note that these are multiplicatons on complex numbers!
- We treat $(C(w^0), C(w^1), \ldots, C(w^{n-1}))$ as coefficients, and by computing the inverse FFT (using w^{-1}!) and dividing the results by n, we obtain the coefficient representation $(c_0, c_1, \ldots, c_{n-1})$.