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.

Median finding

Here we present a linear time algorithm for finding the median a list of numbers. Recall that if $x_1 \leq x_2 \leq \ldots \leq x_n$, 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 kth smallest number (the median corresponds to $k = \lceil n/2 \rceil$).

Given an array $A[1,2,\ldots,n]$, one algorithm for selecting the kth 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 kth 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 $\lfloor n/5 \rfloor/2 \geq n/10 - 1$ 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

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

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

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

which translates into $C_2(n - 9n/10) \leq C_1 n$, 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 $T(n) = \Theta(n^{\log 7})$.

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.

Polynomial Multiplication (Fast Fourier Transform)

In this problem we have two polynomials

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{n-1}x^{n-1}$$
$$B(x) = b_0 + b_1x + b_2^2 + b_3x^3 + \ldots + b_{n-1}x^{n-1}$$

We assume n is a power of 2 and that $a_{n/2} = a_{n/2+1} = \ldots = a_{n-1} = 0, b_{n/2} = b_{n/2+1} = \ldots = b_{n-1} = 0$. This is without loss of generality, since if the actual degrees are d then we can round $2(d+1)$ up to the nearest power of 2 and let that be n, at most roughly quadrupling the degree (coefficients of $x^{d+1}, x^{d+2}, \ldots, x^{n-1}$ will just be set to 0). The goal in this problem is to compute the coefficents of the product polynomial

$$C(x) = c_0 + c_1x + c_2x^2 + \ldots + c_{n-2}x^{n-1}.$$

where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}, \text{ treating } a_i, b_i \text{ as 0 for } i \geq n.$$

Note that by our choice of rounding degrees up to n, the degree of C is less than n (it is at most $n-2$ in fact).

The straightforward algorithm is to, for each $k = 0, 1, \ldots, n-1$, compute the sum above in $2k+1$ arithmetic operations. The total number of arithmetic operations is thus $\sum_{k=0}^{n-1}(2k+1) = \Theta(n^2)$. Can we do better? We shall, by making use of the following interpolation theorem!

Theorem 4.1 *(Interpolation) A degree-N polynomial is uniquely determined by its evaluation on $N+1$ points.*

Proof: Suppose $P(x)$ has degree N and for $N+1$ distinct points $x_1, \ldots, x_{N+1}$ we know that $y_i = P(x_i)$. Then we have the following equation:

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^N \\ 1 & x_2 & x_2^2 & \ldots & x_2^N \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & x_{N+1} & x_{N+1}^2 & \ldots & x_{N+1}^N \end{pmatrix}}_{V} \underbrace{\begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_N \end{pmatrix}}_{p} = \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N+1} \end{pmatrix}}_{y}$$

That is, $Vp = y$. Thus, if V were invertible we would be done: p would be uniquely determined as $V^{-1}y$. Well, is V invertible? A matrix is invertible if and only if its determinant is non-zero. It turns out this matrix V is well studied and is known as a *Vandermonde matrix*. The determinant is known to have the closed form $\prod_{1 \leq i < j \leq N+1}(x_i - x_j) \neq 0$. The proof of this determinant equality is by induction on N, using elementary row operations. We leave it as an exercise (or you can search for a proof online). ■

Thus we will employ a different strategy! Rather than directly multiply $A \times B$, we will pick n points $x_1, \ldots, x_n$ and compute $\alpha_i = A(x_i), \beta_i = B(x_i)$ for each i. Then we can obtain values $y_i = C(x_i) = \alpha_i \cdot \beta_i$, which by the interpolation theorem uniquely determines C. Given all the α_i, β_i values, it just takes n multiplications to obtain the y_i's, but there is of course a catch: computing a degree $n-1$ polynomial such as A on n points naively takes $\Theta(n^2)$ time. We also have to translate the y_i into the coefficients of C. Doing this naively requires computing V^{-1} then multiplying $V^{-1}y$ as in the proof of the interpolation theorem, but matrix inversion on an arbitrary $n \times n$ matrix itself must clearly take $\Omega(n^2)$ time (just writing down the inverse takes this time!), and in fact the best known algorithm for matrix inversion is as slow as matrix multiplication.

Our approach for getting around this is to not pick the x_i arbitrarily, but to pick them to be algorithmically convenient, so that multiplying by V, V^{-1} can both be done quickly. In what follows we assume that our computer

is capable of doing infinite precision complex arithmetic, so that adding and multiplying compex numbers takes constant time. Of course this isn't realistic, but it will simplify our presentation (and it turns out this basic approach can be carried out in reasonably small precision).

We will pick our n points as $1, w, w^2, \ldots, w^{n-1}$ where $w = e^{2\pi i/n}$ (here $i = \sqrt{-1}$) is a "primitive nth root of unity". While this may seem magical now, we will see soon why this choice is convenient. Also, a recap on complex numbers: a complex number is of the form $a + ib$ where a, b are real. Such a number can be drawn in the plane, where a is the x-axis coordinate, and b is the y-axis coordinate. Then we can also represent such a number in polar coordinates by the radius $r = \sqrt{a^2 + b^2}$ and angle $\theta = \tan^{-1}(b/a)$. A random mathematical fact is that $e^{i\theta} = \cos\theta + i\sin\theta$, and thus we can write such an $a + ib$ as $re^{i\theta}$. Note that if one draws $1, w, w^2, \ldots, w^{n-1}$ in the plane, we obtain equispaced points on the circle (and note the next power in the series $w^n = (e^{2\pi i/n})^n = e^{2\pi i} = cos(2\pi) + i\sin(2\pi) = 1$ just returns back to the start).

Now, I claim that divide and conquer lets us compute all the evaluations $A(1), A(w), \ldots, A(w^{n-1})$ in $O(n \log n)$ time via divide and conquer! This is via an algorithm known as the Fast Fourier Transform (FFT). Let us write

$$A(x) = \underbrace{(a_0 + a_2 x^2 + a_4 x^4 + \ldots + a_{n-2} x^{n-2})}_{A_{even}(x^2)} + x\underbrace{(a_1 + a_3 x^2 + a_5 x^4 + \ldots + a_{n-1} x^{n-2})}_{A_{odd}(x^2)}$$

Note the polynomials A_{even}, A_{odd} each have degree $n/2 - 1$. Thus evaluating a degree-$(n-1)$ polynomial on n points reduces to evaluating two degree-$(n/2-1)$ polynomials on the n points $\{1^2, w^2, (w^2)^2, (w^3)^2, \ldots, (w^{n-1})^2\}$. Note however that w^2 is just $e^{4\pi i/n} = e^{2\pi i/(n/2)}$ an $(n/2)$th root of unity! Thus, this set of "n" points is actually only a set of $n/2$ points, since $(w^2)^{n/2}, (w^2)^{n/2+1}, \ldots, (w^2)^{n-1}$ again equals $1, w^2, (w^2)^2, \ldots, (w^2)^{n/2-1}$. Thus what we have is that the running time to compute $A(x)$ satisfies the recurrence

$$T(n) = 2T(n/2) + \Theta(n).$$

This is the same recurrence we've seen before for MergeSort, and we know it implies $T(n) = \Theta(n \log n)$. Evaluating B at these points takes the same amount of time.

Ok, great, so we can get the evaluations of C on n points. But how do we translate that into the coefficients of c? Basically, this amounts to computing $V^{-1}y$. So what's V^{-1}?

Lemma 4.2 *For $x_1, \ldots, x_{n+1} = 1, w, \ldots, w^{n-1}$ for w an nth root of unity, we have that for V the corresponding Vandermonde matrix, $(V^{-1})_{i,j} = \frac{1}{n} \cdot w^{-ij}$.*

Proof: First of all, we should observe that $V_{i,j} = w^{ij}$. Define Z to be the matrix $Z_{i,j} = \frac{1}{n} \cdot w^{-ij}$. We need to verify that $Z = V^{-1}$, i.e. that VZ is the identity matrix, so we should have $(VZ)_{ij}$ being 1 for $i = j$ and 0 otherwise.

$$(VZ)_{ij} = \frac{1}{n} \sum_{k=0}^{n-1} w^{ik} w^{-kj} = \frac{1}{n} \sum_{k=0}^{n-1} (w^{i-j})^k$$

If $i = j$ then this is $\frac{1}{n} \cdot n = 1$ as desired. If $i \neq j$, then we have a geometric series which gives us

$$\frac{1}{n} \cdot \frac{(w^{i-j})^n - 1}{w^{i-j} - 1} = \frac{1}{n} \cdot \frac{(w^n)^{i-j} - 1}{w^{i-j} - 1} = \frac{1}{n} \cdot \frac{1-1}{w^{i-j} - 1} = 0$$

as desired. ■

Now, recall for a complex number $z = a + ib = re^{i\theta}$, its *complex conjugate* $\bar{z}$ is defined as $\bar{z} = a - ib = re^{-i\theta}$. Let us define for a vector $x = (x_1, \ldots, x_N)$ of complex numbers its complex conjugate $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_N)$. Then the above lemma implies that $V^{-1} y = \overline{V \bar{y}}$ (check this as an exercise!). The complex conjugation operator can be applied to y in linear time, then we can multiply by V in linear time (this is just the FFT again, on a polynomial whose coefficients are given by $\bar{y}$!), then we can apply conjugation to the result again in linear time.

This completes the description and analysis of the FFT.

Applications of the FFT

Integer multiplication. Suppose we have two integers a, b which are each n digits and we want to multiply them. For example, we might have $a = 1254, b = 2047$. We can treat an integer as simply a certain polynomial evaluated at 10. For example, define $A(x) = 4 + 5x + 2x^2 + x^3$ and $B(x) = 7 + 4x + 2x^3$. Then $a \cdot b = c$ is just $(A \cdot B)(10)$. Thus we just need to compute $C = A \cdot B$ and evaluate it at the point 10, which we can using $O(n \log n)$ arithmetic operations via the FFT.

Note that we have not discussed precision at all, which actually should be carefully considered. First, there is the issue of the precision to which we represent w in the FFT and do our FFT arithmetic, so that our accumulated errors are sufficiently small; we do not analyze that here though it is known that it is possible to do this reasonably (see [S96]). There is also the issue though that, while the coefficients of A, B are each digits 0–9 and can be concatenated to form the integers a, b, this is not true for obtaining c from the coefficients of C. This is because if we write

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$

and similarly for B, then we have

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

That is, the coefficients of c are not in the range 0–9. In fact they can be as big as $9^2 \cdot n = 81n$. Thus when we evaluate $C(10)$, we cannot simply concatenate the coefficients of C to obtain c: we have to deal with the carries. The key thing to note is that if a number is at most $81n$, then writing it base 10 takes at most $D = \lceil \log_{10}(81n) \rceil = O(\log n)$ digits. Thus, evaluating $C(10)$ even naively requires $O(nD) = O(n \log n)$ arithmetic operations.

Pattern matching. Suppose we have binary strings $P = p_0 p_1 \cdots p_{m-1}$ and $T = t_0 t_1 \cdots t_{n-1}$ for $n \geq m$, and the p_i, t_i each being 0 or 1. We would like to report all indices i such that the length-m string starting at position i in T matches the smaller string P. That is, $T[i, i+1, \ldots, i+m-1] = P$. The naive approach would try all possible $n-m+1$ starting positions i and check whether this holds character by character, taking time $\Theta((n-m+1)m) = O(nm)$. It turns out that that we can achieve $O(n \log n)$ using the FFT!

First we double the lengths of P, T by the following encoding: for each letter which is 0, we map it to 01, and for each that is 1 we map it to 10. So for example, to find all occurrences of $P = 101$ in $T = 10101$, we would actually search for all occurrences of $P' = 100110$ in $T' = 1001100110$. Write $P' = a_0 a_1 \cdots a_{2m-1}, T' = b_0 b_1 \cdots b_{2n-1}$. We now define the polynomials

$$A(x) = a_{2m-1} + a_{2m-2}x + a_{2m-3}x^2 + \ldots + a_0 x^{2m-1}$$

and

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{2n-1} x^{2n-1}.$$

(Note that the coefficients in A are reversed from the order they appear in P'!) Now let us see what $C(x) = A(x) \cdot B(x)$ gives us. If $i \geq 2m-1$, let us write $i = 2m-1+j$ for $j \geq 0$. Then we have that the coefficient of x^i in $C(x)$ is

$$c_i = \sum_{k=0}^{2m-1} a_{2m-1-k} b_{2m-1-k+j}.$$

That is, we are taking the dot product of $T'[j, j+1, \ldots, j+2m-1]$ with P'. Because of our encoding of 0 as 01 and 1 as 10, note that two matching characters in T, P contribute 1 to the dot product, and mismatched characters contribute 0. Thus the sum above for c_i will be exactly m if $T[j/2, j/2+1, \ldots, j/2+m-1]$ matches P and will be strictly less than m otherwise (the division by 2 is since we blew up the lengths of T, P each by a factor of 2). Thus by simply checking which c_{2m-1+j} for even j are equal to exactly m, we can read off the locations of occurrences of P in T. The time is dominated by the FFT, which is $O(n \log n)$.

References

[1] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan. Time Bounds for Selection. *J. Comput. Syst. Sci.* 7(4): 448-461, 1973.

[2] James C. Schatzman. Accuracy of the Discrete Fourier Transform and the Fast Fourier Transform. *SIAM J. Sci. Comput.* 17(5), 1150–1166, 1996.