CS 341: ALGORITHMS (S18) — LECTURE 4
FAST MULTIPLICATION

ERIC BLAIS

The MergeSort algorithm we studied in the last lecture is a classic example of the power of the *divide and conquer* technique. This technique is a general approach for solving problems with a three-step approach:

Divide: the original problem into smaller subproblems,
Conquer: each smaller subproblem separately, and
Combine: the results back together.

With MergeSort, we divided the initial array in two, conquered the sorting problem on the two smaller arrays with recursive calls to MergeSort, and combined the results using Merge. In the next few lectures, we will see other problems where this approach is applicable.

1. Integer multiplication

Let's consider one of the most fundamental arithmetic problems around: how to multiply two positive integers.

Definition 4.1 (Integer multiplication problem)**.** Given two n-bit positive integers x and y, output their product xy.

Humans invented an algorithm for this problem thousands of years ago. It is now known as the grade school algorithm. With this algorithm, we multiply x by y_i and shift the result $n-i$ positions to the left. As an example, when we run this algorithm to solve 13×11, we obtain

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ +\ 1101 \\ \hline 10001111 \end{array} \qquad (= 143)$$

Notice that in the computations, we produces $O(n^2)$ intermediate bits, and this is indeed the time complexity of this algorithm. This begs the question: can we use the Divide & Conquer approach to obtain a more efficient algorithm?

To apply the Divide & Conquer approach, we need to first determine how we can break up the original multiplication problem into problems on smaller inputs. The natural way to do this is to split the n-bit integer x into two $n/2$-bit integers x_L (containing the $n/2$

most-significant bits) and x_R (containing the least-significant bits) and to do the same with y. Then

$$x = 2^{n/2}x_L + x_R,$$
$$y = 2^{n/2}y_L + y_R,$$

and

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

This is exactly what we were hoping to see! We have divided up the problem of multiplying two n-bit integers into four instances of the smaller problem of multiplying two $n/2$-bit integers. The resulting algorithm is as follows.

Algorithm 1: MULTIPLY $(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ SPLIT (x);
$(y_L, y_R) \leftarrow$ SPLIT (y);
$P_{LL} \leftarrow$ MULTIPLY (x_L, y_L);
$P_{LR} \leftarrow$ MULTIPLY (x_L, y_R);
$P_{RL} \leftarrow$ MULTIPLY (x_R, y_L);
$P_{RR} \leftarrow$ MULTIPLY (x_R, y_R);
return $2^n P_{LL} + 2^{n/2}(P_{LR} + P_{RL}) + P_{RR}$;

At first, it might be worrisome to see that we have also added extra multiplications by 2^n and $2^{n/2}$, but in fact these are just shifts (by n and $n/2$ bits, respectively), so these operations have time complexity $O(n)$ and the total time complexity of the MULTIPLY algorithm is defined by the recursion

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

We can apply the Master Theorem with parameters $a = 4$, $b = 2$, and $c = 1$ and the observation that $1 = c < \log_b a = 2$ to obtain

$$T(n) = O(n^2).$$

This is exactly the same as the grade school algorithm! So while the Divide & Conquer approach gives an elegant algorithm, it does not appear to have led to any efficiency improvement.

2. Fast integer multiplication

...And yet if that was the end of the story, we probably wouldn't be covering the integer multiplication problem at this point in the class! Recall that

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

There's a deceptively simple observation that can be traced back to Gauss which will be immensely useful to us: the middle term $x_L y_R + x_R y_L$ satisfies the identity

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

We can use this identity to express the term $x_L y_R + x_R y_L$ with a difference of three multiplications. This would be a step back for us (we only need 2 multiplications to compute the same term directly!) except that two of the multiplications, $x_L y_L$ and $x_R y_R$ are multiplications that we already need to do in the multiplication algorithm anyways! As a result, we can now implement a Divide & Conquer multiplication algorithm that performs only 3 multiplications instead of 4.

Algorithm 2: FastMultiply $(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n)$

if $n = 1$ **return** xy;
$(x_L, x_R) \leftarrow$ Split (x);
$(y_L, y_R) \leftarrow$ Split (y);
$P_{LL} \leftarrow$ FastMultiply (x_L, y_L);
$P_{RR} \leftarrow$ FastMultiply (x_R, y_R);
$P_{\text{sum}} \leftarrow$ FastMultiply $(x_L + x_R, y_L + y_R)$;
return $2^n P_{LL} + 2^{n/2}(P_{\text{sum}} - P_{LL} - P_{RR}) + P_{RR}$;

The time complexity of this algorithm now satisfies

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

We can again apply the Master Theorem, this time with parameters $a = 3$, $b = 2$, and $c = 1$. Since $1 = c < \log_b a = \log_2 3 < 1.59$, we obtain

$$T(n) = O(n^{1.59}).$$

This result is a great illustration of the power of algorithmic thinking: despite countless mathematicians (and students of all ages) using multiplication algorithms over thousands of years, it was only in 1960 that Karatsuba showed with the above algorithm that it was possible to solve the integer multiplication in subquadratic time.

3. Fast matrix multiplication

A similar approach can also be used to obtain a fast matrix multiplication algorithm. Given two $n \times n$ matrices A and B, their product is the matrix $C = AB$ whose entries satisfy

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}.$$

The algorithm that uses this definition directly has time complexity $O(n^3)$. Again, we can try to use the Divide & Conquer method to obtain a more efficient algorithm. In this case, it is natural to divide each matrix in four:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

We then observe that the 4 submatrices of C can each be obtained by taking products of the submatrices of A and B:

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

The Divide & Conquer algorithm obtained using these identities makes 8 multiplications on $\frac{n}{2} \times \frac{n}{2}$ matrices and requires $O(n^2)$ extra work (since the matrices it adds together have n^2 entries) so its time complexity satisfies

$$T(n) = 8\,T(n/2) + O(n^2).$$

With the Master Theorem, we see that this time complexity is $T(n) = O(n^3)$, the same as the naïve matrix multiplication algorithm.

As was the case with the integer multiplication problem, however, it is possible to be more clever in how we choose the multiplications of the submatrices so that we compute the product C with only 7 (instead of 8) multiplications. Doing so yields a time complexity

$$T(n) = 7\,T(n/2) + O(n^2),$$

which is $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$. The resulting algorithm is known as *Strassen's algorithm* and has been extremely influential in practice as well as in theory: there are numerous computational problems today that require multiplication of enormous matrices, and Strassen's algorithm is provides significant efficiency improvements over the naïve algorithms in those settings.

As a good algorithm designer, however, there is one question that you should be asking yourself at this point: can we do even better than Strassen's algorithm? Indeed we can. At the moment, the best known algorithm for matrix multiplication has time complexity $O(n^{2.37286\ldots})$. It is conjectured that matrix multiplication can be done in time $O(n^2)$ which, if true, would clearly be optimal since there are n^2 entries in an $n \times n$ matrix.