The Divide and Conquer Algorithm Design Technique

The rest of the course will mostly be loosely organized around different algorithm design techniques, each accompanied by some examples of its use. Where appropriate, you will be asked to use the techniques to design new algorithms for given problems.

Actually we have already seen one technique: brute force.

Algorithms designed using our second technique—divide and conquer—end up being recursive and hence requiring efficiency analysis like “the Master Theorem”. The divide and conquer technique can lead to algorithms that are more efficient than other, usually simpler, algorithms.

> Most of the other design techniques we will study lead to algorithms whose efficiency is either easily analyzed, or extremely difficult to analyze.

We begin with the divide and conquer technique, which says that to solve a problem, we find a way to view the problem as several sub-problems, such that we can solve the original problem by somehow using the solutions to the sub-problems.

These algorithms can, if properly designed, be more efficient than one might expect.

You have probably already seen this technique in action in CS 2050, say in the the merge sort and quick sort algorithms.

In this chapter and the next we will apply the divide and conquer design technique to several interesting problems. One of these is of some practical use (Karatsuba’s method), one is theoretically interesting and fun to derive, but not all that practical (Strassen’s algorithm), and one—the Fast Fourier Transform—has been characterized by some experts as the most important algorithm in human history for its practical applications (and so it gets its own chapter).

Karatsuba’s Method for Multiplying Integers

For a new problem/algorithm to tackle by the divide and conquer technique, consider the problem of multiplying two integers symbolically, using ordinary “by hand” techniques.

Most people (I hope!) had to learn in elementary school an algorithm for multiplying two integers by hand. One version of this algorithm is demonstrated here—each row is produced by multiplying the first number by one digit of the second number, suitably shifted, and then the columns have to be added up:

$$\begin{array}{cccccccc}
 & & & & 3 & 1 & 7 & 8 \\
 & & \times & & 5 & 6 & 9 & 4 \\
\hline
 & & & 1 & 2 & 7 & 1 & 2 \\
 & & 2 & 8 & 6 & 0 & 2 & \\
 & 1 & 9 & 0 & 6 & 8 & & \\
1 & 5 & 8 & 9 & 0 & & & \\
\hline
1 & 8 & 0 & 9 & 5 & 5 & 3 & 2
\end{array}$$

To be fair in analyzing the new algorithm, we will count both the operation of multiplying two digits and the operation of adding two digits.

It is clear, with a little thought, that if we want the product of two n digit integers, and we're using the standard algorithm, the number of digit multiplications is clearly in $\Theta(n^2)$, because each of the n digits in the first number has to be multiplied by each of the n digits in the second number.

It is a little harder to see that the number of digit additions is in $\Theta(n^2)$. We can note that there are n rows—4 in our example, and each row has n or $n+1$ digits (the product of two single digit integers has either 2 or 3 digits in its answer, depending on how big they are). So, the number of digits that gets involved in an addition is clearly in $\Theta(n^2)$ (there will be $\Theta(n)$ extra additions to handle the extra digits on some rows and the carries).

According to Levitin, in 1960 Anatoly Karatsuba developed a more efficient algorithm for this elementary school problem! We will now develop this algorithm using the divide and conquer technique.

Karatsuba used the divide and conquer idea of thinking about this multiplication not as a bunch of digit by digit products, but as a number of half-size problems. He approached the problem of multiplying two n-digit numbers by the strategy of doing some $n/2$ multiplication problems and then adding.

Specifically, suppose that x and y are n-digit numbers, and you want to form xy (the multiplication operator is often omitted as usual in algebra). So, we break x into its first $n/2$ digits, a, and its last $n/2$ digits b (assuming that n is even for convenience), then $x = 10^{n/2}a + b$, and similarly $y = 10^{n/2}c + d$, where all of a, b, c, and d are $n/2$ digits long.

For our earlier example, this idea means to compute

$$(31 \cdot 100 + 78) \cdot (56 \cdot 100 + 94)$$

Now, by simple algebra,

$$xy = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = ac\,10^n + (ad + bc)10^{n/2} + bd,$$

which shows that xy can be formed by doing four multiplications of size $n/2$, namely ac, ad, bc, and bd, and then doing a bunch of shifting and adding. Note, though, that the work to add and shift is in $\Theta(n)$ and hence negligible.

We can analyze the efficiency of this algorithm by letting $T(n)$ be the number of single digit multiplications needed to multiply together two n digit integers, and noting that

$$T(n) = 4T(n/2)$$

for the algorithm we have just designed.

Using the Master Theorem (really just substituting and extrapolating), where as usual we take $n = 2^m$, we see that

$$T(2^m) = 4T(2^{m-1}) = 4^2T(2^{m-2}) = \cdots = 4^mT(1) = 4^{\log_2 n} = n^{\log_2 4} = n^2$$

so $T(n) \in \Theta(n^2)$.

But, Karatsuba went on to notice something clever, namely that he could form the product of two n digits numbers by doing *only three* half-size multiplications! Instead of computing the 4 products ac, ad, bc, and bd, we can instead begin by computing $p_1 = (a+b) \cdot (c+d)$ (an apparently mildly insane action). Then we go ahead and compute $p_2 = ac$ and $p_3 = bd$, and, after only 3 size $n/2$ multiplications, we have enough information to compute xy.

The crucial observation is that since

$$p_1 = (a+b)(c+d) = ac + ad + bc + bd,$$

if we have computed $p_2 = ac$, and $p_3 = bd$, then we can compute $ad + bc$ by doing

$$p_1 - p_2 - p_3 = (a+b)(c+d) - ac - bd = ad + bc.$$

The key idea is that we don't actually need ad and bc separately, we only need $ad + bc$.

Example

Let's solve our earlier example using this idea. To compute $3178 \cdot 5694$, we have $a = 31$, $b = 78$, $c = 56$, and $d = 94$, so

$$p_1 = (a+b)(c+d) = (31+78)(56+94) = 109 \cdot 150 = 16350,$$

$$p_2 = ac = 31 \cdot 56 = 1736,$$

and

$$p_3 = bd = 78 \cdot 94 = 7332.$$

Now we can do some subtracting, obtaining

$$ad + bc = p_1 - p_2 - p_3 = 16350 - 1736 - 7332 = 7282.$$

Then we just have to take our three values ac, $ad + bc$, and bd, and shift them correctly (noting that $n = 4$), and add them, getting

$$ac \cdot 10000 + (ad + bc)100 + bd = 17360000 + 728200 + 7332 = 18095532.$$

In doing this by hand, it is important to note that the subtraction step and the shifting and adding step all take time in $\Theta(n)$—they are quite easy to do by hand, without a calculator, even.

$\Rightarrow$ The previous example does 4-digit multiplication by use three 2-digit multiplications. The actual recursive algorithm would do all three of those by Karatsuba's algorithm as well. Let's do them, just for more practice.

Now we want to analyze the efficiency of this algorithm. Of course, the complete divide and conquer algorithm uses the main idea recursively. For example, to multiply two 8-digit numbers, we use the main idea to form this product using just three 4-digit by 4-digit multiplications. But, then to do each of those 4-digit by 4-digit multiplications, we use the same idea to do each of them using just three 2-digit by 2-digit multiplications. And, finally, we do all of the 2-digit by 2-digit multiplications using the same main idea to require just three 1-digit by 1-digit multiplications. So, how many total 1-digit by 1-digit multiplications does that make? It's not so easy to see, which is why we use our standard analysis technique, as follows.

So, we let $S(n)$ be the total number of operations—single digit multiplications, single digit additions, and shiftings—required to multiply two n-digit numbers, assume as usual that $n = 2^m$, and observe that

$$S(n) = S(2^m) = 3S(2^{m-1}) + a2^m + b,$$

where a and b are smallish positive integers, with these terms accounting for extra work being done to add, subtract, and shift various numbers of length 2^m, and any constant overhead.

In our usual way, we can show that

$$S(n) \in \Theta(3^{\log_2(n)}) = \Theta(n^{\log_2(3)}).$$

So, Karatsuba's algorithm takes $n^{\log_2(3)} \approx n^{1.5849625}$. For example with $n = 512$, Karatsuba's algorithm takes $3^{\log_2(512)} = 3^9 = 19683$ single-digit multiplications,
whereas the traditional algorithm takes $512^2 = 262144$ single-digit multiplications.

According to the documentation for the Java `BigInteger` class, Karatsuba's method is actually used for the `multiply` method.

Exercise 8 [10 points] (target due date: Tuesday, September 8)

Demonstrate how to use Karatsuba's method to compute

$$3157624897154208 \cdot 1342796457403786$$

Assume you are using a calculator that can only compute products up to 4 digits times 4 digits, and do all additions, subtractions, and shifts, of any size, "by hand." Note that this will require you to do Karatsuba's idea multiple times, first for computing some 8 digit by 8 digit products, and then doing each of those by some 4 digit by 4 digit products (on the calculator), and then putting it all together by hand.

Actually, sometimes you'll have a fifth digit, so allow yourself the ability to multiply 4 or 5 digit numbers on the calculator.

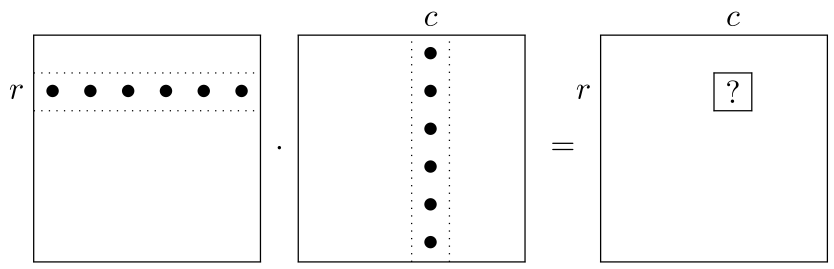
Strassen's Algorithm for Multiplying Square Matrices

Next you'll learn how to design—by the divide-and-conquer approach—an algorithm that can multiply two square matrices (as usual with size that is a power of 2) more efficiently than the standard algorithm.

Matrix Multiplication

Recall from matrix algebra the standard algorithm for multiplying two matrices. To compute $C = AB$, where A is an m by p matrix and B is a p by n matrix, each element C_{rc} is computed as row r of A "times" column c of B, where a "row times a column" is done by multiplying all corresponding spots in the two lists and adding them up.

It is easy to see that for n by n matrices, a "row times a column" takes n multiplications and $n-1$ additions, which is in $\Theta(n)$, and there are n^2 elements of C to be computed, so traditional matrix multiplication is in $\Theta(n^3)$.



Deriving Strassen's Algorithm

Now we are going to develop Strassen's algorithm. I will do most of it, leaving you in Exercise 9 to finish and do a demonstration of the method.

> We are studying Strassen's algorithm for practice on divide and conquer, not because it is very useful in practice.

Consider multiplication of two 2 by 2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}.$$

This algorithm takes 8 multiplications and 4 additions. Strassen's algorithm is based on the following clever way to get the same result at a cost of 7 multiplications and a bunch of additions, which, when we do the efficiency analysis, turns out to be better asymptotically.

> Of course, to be more efficient, Strassen's algorithm applies the same idea that we will be deriving recursively. For example, if you wanted to multiply two 1024 by 1024 matrices, a through h would each be matrices of size 512 by 512, and Strassen's algorithm would be applied to compute each 512 by 512 product. To avoid going mildly insane, we will only study the clever algebra for Strassen's algorithm when a through h are single numbers.

The idea (a fancier version of what Karatsuba did) is to get the 8 scalars such as ae, bg, and so on, by multiplying sums of terms. For example, $(a+b)(e+h) = ae + ah + be + bh$, so at a cost of one multiplication we can get two terms that we want, namely ae and bh. Of course, we need those terms separately, not added together, and not added to the other terms.

To understand Strassen's idea, we use a 4 by 4 chart to visualize the terms obtained by a product of this sort. Here, for example, is the chart for $(a+b)(e+h) = ae + ah + be + bh$:

$$\begin{array}{c|c|c|c|c|}
 & e & f & g & h \\ \cline{2-5}
a & + & & & + \\ \cline{2-5}
b & + & & & + \\ \cline{2-5}
c & & & & \\ \cline{2-5}
d & & & & \\ \cline{2-5}
\end{array}$$

We can do the same technique with subtraction. For example, $(b-d)(-e+g) = -be + bg + bd - dg$ can be pictured with this chart:

$$\begin{array}{c|c|c|c|c|}
 & e & f & g & h \\ \cline{2-5}
a & & & & \\ \cline{2-5}
b & - & & + & \\ \cline{2-5}
c & & & & \\ \cline{2-5}
d & + & & - & \\ \cline{2-5}
\end{array}$$

Strassen figured out that if we use these quantities involving the four original numbers:

$$m_1 = (a+d)(e+h),$$

$$m_2 = (c-a)(e+f),$$

$$m_3 = (b-d)(g+h),$$

then we can form $m_1 + m_3$ and $m_1 + m_2$ and then find four terms with one original number times the sum of two others, like $(a+b)h$, and can finally form the 4 desired elements of the matrix product at a cost of just 7 multiplications.

⇒ Fill in the + and − symbols in these charts for the 4 *target* expressions in the matrix product—these are the ones we need for our final answer:

$ae + bg$:

	e	f	g	h
a				
b				
c				
d				

$af + bh$:

	e	f	g	h
a				
b				
c				
d				

$ce + dg$:

	e	f	g	h
a				
b				
c				
d				

$cf + dh$:

	e	f	g	h
a				
b				
c				
d				

⇒ Now fill in the symbols for the three key products figured out by Strassen, namely m_1, m_2, and m_3:

$m_1 = (a + d)(e + h)$:

	e	f	g	h
a				
b				
c				
d				

$m_2 = (c - a)(e + f)$:

	e	f	g	h
a				
b				
c				
d				

$m_3 = (b - d)(g + h)$:

	e	f	g	h
a				
b				
c				
d				

⇒ Fill in the symbols for these two suggested sums:

$m_1 + m_2$:

	e	f	g	h
a				
b				
c				
d				

$m_1 + m_3$:

	e	f	g	h
a				
b				
c				
d				

At this point you have been given some big hints from what Strassen did, namely to use 3 of the allotted 7 multiplications to form m_1, m_2, and m_3, and to add them up in this way—forming $m_1 + m_2$ and $m_1 + m_3$.

Exercise 9 [4 points] (target due date: Tuesday, September 8)

Your job now is to cleverly figure out how to spend the remaining 4 multiplications in order to get charts that you can add or subtract with other charts to obtain the 4 target charts.

The idea is to look at the patterns in the three charts you have and compare them to the patterns in the four target charts. You will see the target cells appearing in the three given charts, but there will be unwanted cells—figure out how to do just 4 multiplications allowing you to get rid of the unwanted cells.

Once you have figured this out, write down (somewhere you can cherish them forever, or at least until Test 1 is over) formulas for the four target expressions using sums and differences of just the 7 quantities you have found using 7 multiplications.

> You don't need to write this up as part of your Exercise, but verify that your formulas use 18 additions/subtractions of matrices of size $n/2$ along with 7 multiplications of matrices of size $n/2$, which shows that the recurrence relation given in the last chapter for the number of multiplications and the number of additions in Strassen's algorithm is correct.
>
> Thus, Strassen's algorithm can multiply two n by n matrices in $\Theta(n^{\log_2(7)}$ total arithmetic operations, which is slightly better than the $\Theta(n^3)$ time for the traditional algorithm. According to Wikipedia, Strassen's algorithm is not better in practice until n gets above 100, and the algorithm is more susceptible to rounding error problems than the traditional algorithm.
>
> People (Coopersmith-Winograd and LeGall) have developed a matrix multiplication algorithm with efficiency in $\Theta(n^{2.373})$. This algorithm is apparently not practical, due to a large constant (which we ignore in asymptotic analysis, but is still important in practical terms). It is an open problem whether there are algorithms that get arbitrarily close to $\Theta(n^2)$ time. Obviously, since the algorithm has to look at all $\Theta(n^2)$ numbers in the two matrices being multiplied, no algorithm can do better than $\Theta(n^2)$. so (or so I've read). So, not practical, but philosophically important.

Demonstrate use of your formulas to compute

$$\begin{bmatrix} 5 & 7 \\ 4 & 2 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 3 & 6 \end{bmatrix}.$$

Be sure to check your result against the traditional algorithm so you can catch any errors in your formulas or process.

The Integers Mod N

We need to understand the mathematical entity Z_n (known as "the integers mod n") for our upcoming work on the Fast Fourier Transform, and for our future work on RSA and elliptic curve encryption.

This material is a total digression from the topic of divide and conquer algorithms. Perhaps you have seen it in your discrete math course, but we will assume you haven't, just in case.

Define Z_n to be the integers where arithmetic is done using this simple rule:

> do the operation (addition, subtraction, and multiplication) as usual, and then reduce the result by doing integer division by n and taking the remainder as the result. This operation is known as "reducing a number 'mod n'. "

If you did see this topic in a discrete math course, it was probably couched in terms of equivalence relation and equivalence classes. We are saying the same thing here, just being more concrete.

We consider two integers as being "the same" (in the same equivalence class) if they differ by a multiple of n. Note that all integers belong to Z_n, but we prefer 0 through $n-1$ as representatives. When reduced mod n, all integers are seen to be equal to one of these *canonical* values.

This is a lot like fractions—we prefer $\frac{1}{2}$ to the infinitely many other ways to express this same number as a ratio of two integers, like, say, $\frac{17}{34}$.

For example, Z_5 has the canonical values 0 through 4, with these addition and multiplication tables:

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

⇒ Verify some of the values in this table.

For any prime number n, it turns out that Z_n is a *field*, which is an algebraic entity that satisfies all the algebraic properties of the real numbers that don't involve comparisons.

⇒ Note that for every number x, there is a number y such that $x + y = 0$, and for every non-zero number x, there is a number z that that $xz = 1$. Also, $x + 0 = x$ and $x \cdot 1 = x$ for every x, so 0 and 1 are additive and multiplicative identity elements. Since arithmetic is done as usual, followed by reduction mod n, both addition and multiplication satisfy the associative, commutative, and distributive properties.

Exercise 10 [10 points] (target due date: Tuesday, September 8)

To practice with Z_n, figure out the answers to these questions:

- compute $8 \cdot 15$ in Z_{17}

  Some calculators are fancy enough to reduce a number mod 17 as a built-in operation, but if not, you can divide the number by 17, hit =, subtract the whole number part, hit =, and then multiply by 17 to get the remainder. Also, when working in Z_{17}, it can be useful to start by quickly writing down a bunch of multiples of 17, namely 17, 34, 51, 68, ..., 170. These values are all 0 in Z_{17}, which let's you quickly reduce numbers by 17 in your head, like $40 = 6$ because $40 - 34 = 6$.

- Find the number in Z_{17} you can add to 8 to give 0 (this number can be referred to as -8, but we want it as one of the canonical numbers, i.e, from 0 through 16).

- Find the number in Z_{17} you can multiply by 11 to give 1.

  Note that finding the multiplicative inverse is considerably harder than finding the additive inverse—later we'll learn an efficient algorithm for doing it, but for now you'll have to use brute force (just keep trying different numbers until you find one that works).

- Does 13 have a square root in Z_{17} (a number x such that $x^2 = 13$)? If so, find it.

- It will turn out that there is a very useful fact about the powers of any given value in Z_n. To explore this, fill in this table of the various powers of $\alpha = 3$, in Z_{17}. Be sure to do this efficiently by noting that $\alpha^k = \alpha^{k-1} \cdot \alpha$ for $k \geq 1$—do not compute α^k in Z_{17} using your calculator and then reduce the result mod 17 (this approach will, later, quickly overwhelm your calculator, even for relatively small values of n):

k	0	1	2	3	4	5	6	7	8	9	10
3^k											

k	11	12	13	14	15	16	17	18	19	20
3^k										
