Discrete Mathematics | **U. Waterloo ECE 103, Spring 2010**
Ashwin Nayak | **May 17, 2010**

Recursion

During the past week, we learnt about inductive reasoning, in which we broke down a problem of "size" n, into one or more problems of size smaller than n. For example, to establish a factorisation of an integer n into primes, we expressed it (if possible) as a product of two integers, a, b, both smaller than n. *Assuming* that we had a factorisation for a, b, we constructed one for n. If we were asked to *produce* a prime factorisation for n, the same idea would work: we would test if n is a prime. If it is, our task is done. If not, we would find some divisor a, with $1 < a < n$, so that $n = ab$. Then, we would repeat the same procedure on a, to get a factorisation for a, and similarly for b. Together, these would give us the factorisation for n. This method of solution is called *recursion*.

In the next few lectures, we will see several such instances of problems we can solve using recursion. In addition, we will see how induction helps prove properties (including correctness) of our solutions.

Exponentiation

Suppose we are given numbers a, n, where $n > 0$ is an integer. We wish to calculate the number a^n. What is the quickest way to do this? How many multiplication operations do we have to perform?

Of course, we may compute 19^8 by calculating $19 \times 19 = 361$, then calculating $19^3 = 361 \times 19 = 6859$, then $19^4 = 6859 \times 19 = 130321$, and so on, until we get 19^8. This takes seven multiplications in total. Is this the quickest possible?

Note that $8 = 2 \times 4$, so we can also write $19^8 = 19^4 \times 19^4$. If we compute 19^4 first, and then square it, we need only one more multiplication. The straightforward method would require four more multiplications: $19^8 = 19^4 \times 19 \times 19 \times 19 \times 19$. Similarly, $19^4 = 19^2 \times 19^2$. So if we calculate $19^2 = 361$ with one multiplication, $19^4 = 361^2 = 130321$ with one more, we get $19^8 = 130321^2 = 16983563041$ with the third multiplication. This cleverer method requires only *three* multiplications.

The method above seems to work when the exponent n is even. What do we do when it is odd? Say, we would like to calculate 19^7. We may write $7 = 6+1$, so $19^7 = 19^6 \times 19$, then $19^6 = 19^3 \times 19^3$, and finally $19^3 = 19^2 \times 19$. So $19^3 = 361 \times 19 = 6859$, $19^6 = 6859^2 = 47045881$, and $19^7 = 47045881 \times 19 = 893871739$. The straightforward method of calculation requires 6 multiplications, and we needed only 4 here. Curiously, the number is more than we needed for 19^8. We will see the reason for this below.

We can combine the ideas from the two examples above to get a procedure to calculate the power a^n for any pair a, n. This kind of procedure, which specifies a sequence of steps to complete a given task is called an *algorithm*.

Algorithm 1: POWER(a,n), a recursive procedure to calculate a power.

```
input  : Real number $a$, integer $n \geq 1$.
output: The number $a^n$.
if $n = 1$ then
    return $a$;
if $n$ is even then
    Let $b$ = POWER$(a, n/2)$ ;
    return $b^2$;
else
    // n is odd
    Let $b$ = POWER$(a, (n-1)/2)$ ;
    return $a \times b^2$ ;
```

Let us see how this algorithm computes 2^{13}.

Example 1 *Calculate 2^{13} using algorithm* POWER, *and state the number of multiplications used.*

Solution : The following table summarizes the calculations made by POWER$(2, 13)$. Let $a = 2$, and b denote the value returned by the recursive call to POWER in one step.

Step	n	b	calculation	value
1	13	a^6	$a \times (a^6)^2$	8192
2	6	a^3	$(a^3)^2$	64
3	3	a	$a \times (a)^2$	8
4	1			2

The number of multiplications used is 5. ■

The algorithm POWER(a, n) is correctly calculates a^n, as long as it correctly calculates the relevant smaller powers of a. We may prove by induction over n that this happens for any number a.

Theorem 1 *The algorithm* POWER(a, n) *returns a^n for every real number a and integer $n \geq 1$.*

Proof : The base case is $n = 1$, when the algorithm correctly returns a. Assume that the algorithm is correct for all k, with $1 \leq k \leq n - 1$. Consider POWER(a, n). If n is even, the algorithm returns the value b^2, where $b =$ POWER$(a, n/2)$. Since $n > 1$ and is even, $n/2$ is an integer, and $1 \leq n/2 \leq n - 1$. By induction hypothesis, POWER$(a, n/2)$ correctly returns $a^{n/2}$, so the algorithm returns $b^2 = a^n$. If n is odd ($n > 1$), then $(n - 1)/2$ is an integer, and $1 \leq (n - 1)/2 \leq n - 1$. By the induction hypothesis, $b =$ POWER$(a, (n - 1)/2) = a^{(n-1)/2}$, and the algorithm correctly returns $a \times b^2 = a^n$ in this case as well. So the algorithm is correct by the second principle of mathematical induction. ■

How many multiplications does the above procedure need to compute a^n? Running through our earlier examples, we see that the procedure calculates 19^8 and 19^7 just as we did. So it takes three and four multiplications in these cases. It did five multiplications to calculate 2^{13}. How does the number of multiplications scale with n?

In every step of the recursion, the exponent n decreases by a factor of at least 2, and we perform at most 2 multiplications. We stop when the exponent is 1. If there are k recursive steps in all, the total number of multiplications is at most $2k$. Since the exponent decreases by a factor of two in every step, the final exponent (i.e., 1) is at most $n/2^k$. This gives us $1 \leq n/2^k$, i.e., $k \leq \log_2 n$, and that the number of multiplications is at most $2 \log_2 n$. Let us prove all of this more rigorously.

Let $M(n)$ be the number of multiplications that the procedure POWER(a, n) uses to compute a^n. We have

$$M(1) = 0 \tag{1}$$
$$M(n) = M(n/2) + 1, \qquad \text{if } n \text{ is even,} \tag{2}$$
$$M(n) = M((n-1)/2) + 2, \qquad \text{if } n \text{ is odd and } n > 1. \tag{3}$$

Theorem 2 *$M(n) \leq 2 \log_2 n$ for all integers $n \geq 1$.*

Proof : We prove this using strong induction. The base case is $n = 1$, when $M(1) = 0 = 2 \log_2 1$. Assume that the statement is true for all k such that $1 \leq k \leq n - 1$. Consider n. If n is even, we have $1 \leq n/2 \leq n - 1$, so that $M(n) = M(n/2) + 1 \leq 2 \log_2(n/2) + 1 = 2 \log_2 n - 1$. If n is odd, $1 \leq (n - 1)/2 \leq n - 1$ and $M(n) = M((n - 1)/2) + 2 \leq 2 \log_2((n - 1)/2) + 2 \leq 2 \log_2(n - 1)$. In both cases, $M(n) \leq 2 \log_2 n$. ■

In fact, the recurrence for $M(n)$ may be solved exactly to get

$$M(n) = k_n + \mathrm{h}(n - 2^{k_n}), \tag{4}$$

where $k_x = \lfloor \log_2 x \rfloor$ is the largest integer k such that $2^k \leq x$, and $\mathrm{h}(x)$ denotes the number of 1s in the base 2 (i.e., binary) representation of the non-negative integer x. This latter term, which is the number of *bits* required to represent x, is bounded by $\log_2 x$, so $M(n) \leq 2 \log_2 n$. We however do not attempt to derive this exact solution here.

The straightforward (sequential) method for calculating a^n would take $n-1$ multiplications. The difference between n and $\log_2 n$ is enormous. If $n = 2^{500}$, a 500-bit integer, the algorithm $\text{POWER}(a, n)$ needs at most 1000 multiplications (actually at most 500), whereas the sequential method needs $2^{500} - 1$. Even assuming the current generation of computers can multiply two numbers in one clock cycle, i.e., in $10^{-9}s$, the sequential algorithm will take more than the estimated lifetime of the universe! The recursive procedure however will give us the answer within a second.

In the theory of computation, algorithms such as POWER are deemed to be *efficient*, as they take time that is polynomial (in this case, linear) in the bit-representation of the input. Precisely which problems admit efficient algorithms, and which may not, is a subject of intensive study in computer science.

Multiplication

In the our discussion on powering, we intentionally regarded multiplication as a basic operation, and estimated the "cost" of powering in terms of this operation. It is clear, though, even from the small examples we studied above $(19^7, 19^8)$ that the numbers encountered by the algorithm grow rapidly in size. The result of multiplying two n digit integers is up to $2n$ digits long. Evidently, the "cost" of multiplying these numbers grows as the number of digits increases, and should be taken into account.

Recall the method for multiplication we learnt in school. Let us square the integer 130321 using this method:

$$\begin{array}{r}
130321 \\
130321\times \\
\hline
130321 \\
260642 \\
390963 \\
000000 \\
390963 \\
130321 \\
\hline
16983563041
\end{array}$$

We multiply the first integer 130321 by the digits of the second, starting from the right, one at a time, writing the results in a staggered fashion, shifting each successive result one digit to the left. We then add up all the six integers to get our answer.

In our example above, we started with two 6-digit integers, whose product is 12-digit long. To obtain the product, we added six integers, each up to 12-digit long. (The shifting to the left corresponds to extra 0-digits we append on the right.) In total, we added roughly $6 \times 12 = 72$ digits to get our answer.

In general, this method involves n^2 single digit multiplications and the addition of n integers with at most $2n$ digits to calculate the product of two n-digit integers. So its "cost" is $3n^2$. Can we do better?

In 1960, Andrey Kolmogorov, an eminent mathematician at Moscow State University, posed this question in a seminar. (Among other famous pieces of work, he laid the foundations of modern probability theory.) He conjectured that no procedure could multiply two n-digit integers with significantly fewer than n^2 elementary operations. Within a week, Anatolii Karatsuba, then a 23-year old student in the seminar, found a recursive algorithm that accomplishes this in a constant times $n^{\log_2 3} \approx n^{1.585}$ elementary operations, thus disproving Kolmogorov's conjecture. Apparently, the term "divide and conquer", which refers to a commonly used recursive technique, was first used for this algorithm.

To understand the Karatsuba algorithm, let us look at another example. Suppose we would like to compute 130321×123131. We may write

$$\begin{aligned}
\overbrace{130}^{a}\,\overbrace{321}^{b} &= 130 \times 10^3 + 321 \\
&= a10^3 + b, \qquad \text{and} \\
\overbrace{123}^{c}\,\overbrace{131}^{d} &= 123 \times 10^3 + 131 \\
&= c10^3 + d.
\end{aligned}$$

In other words, we may express a 6 digit integer in terms of two 3-digit integers. (Note that multiplication by a power of 10^3 corresponds to shifting an integer to the left by three digits.) Now

$$\begin{aligned}
130321 \times 123131 &= (130 \times 10^3 + 321)(123 \times 10^3 + 131) \\
&= (130 \times 123)10^6 + (130 \times 131 + 321 \times 123)10^3 + 321 \times 131 \\
&= (ac)10^6 + (ad + bc)10^3 + bd. \qquad (5)
\end{aligned}$$

In effect, we have reduced the problem to that of multiplying four pairs of 3-digit integers, and the addition of three pairs of integers with at most 12 digits. (Multiplying by a power of 10, say 10^k, corresponds to simply shifting the integer by k digits.)

In general, the multiplication of two n-digit integers is reduced to four multiplications of $(n/2)$-digit integers, and three additions of $2n$ digit integers. If we work out the number of elementary operations in this scheme, using the estimate from the earlier method for multiplication, we get $4 \times 3(n/2)^2$ operations for the multiplications, and $3 \times (2n)$ for the additions, for a total of $3n^2 + 6n$. The dominant term here is $3n^2$ for large n, which gives us no advantage over the earlier method.

Karatsuba's idea was to reduce the number of multiplications of the smaller integers from four to three, while increasing the number of additions. He suggested computing the middle term in Equation (5) as follows:

$$\begin{aligned}
ad + bc &= (a + b)(c + d) - ac - bd, \qquad \text{i.e., as} \\
130 \times 131 + 321 \times 123 &= (130 + 321)(123 + 131) - (130 \times 123) - (321 \times 131).
\end{aligned}$$

Since the products we subtract above are the first and the last products in Equation (5), he effectively reduced two multiplications and an addition to one multiplication and four additions. (Recall that subtraction is the addition of a negative number).

If we use Karatsuba's idea for multiplying n-digit integers, the number of elementary operations we need is $3 \times 3(n/2)^2$ for the multiplications, and $6 \times (2n)$ for the additions, for a total of $(9/4)n^2 + 12n$. The dominant term here is $(9/4)n^2$, a factor of $3/4$ smaller than in the first attempt. Evidently, this improvement comes about from our reduction of four multiplications to three.

Strictly speaking, our estimate above is not entirely correct, as the sum of the two $(n/2)$-digit integers may have a carry over, and give us an $(n/2 + 1)$-digit integer. However, we can verify that even with this correction, the dominant term remains $(9/4)n^2$.

The rewards of this constant factor savings become clearer when we apply it *recursively* on the multiplication of smaller, $(n/2)$-digit integers. Let us see how this algorithm looks.

Let x, y be two n-digit integers. For the moment, assume that n is even. We break each integer into a sum coming from the first $(n/2)$ and the last $(n/2)$ digits. Call the integer represented by the first $(n/2)$ digits of x as x_{L} and the integer represented by the last $(n/2)$ digits as x_{R}. Similarly we get the integers $y_{\mathrm{L}}, y_{\mathrm{R}}$ from y. Then

$$\begin{aligned}
x &= x_{\mathrm{L}} \times 10^{n/2} + x_{\mathrm{R}} \\
y &= y_{\mathrm{L}} \times 10^{n/2} + y_{\mathrm{R}},
\end{aligned}$$

the product

$$\begin{aligned} xy &= (x_{\mathrm{L}}10^{n/2} + x_{\mathrm{R}})(y_{\mathrm{L}}10^{n/2} + y_{\mathrm{R}}) \\ &= (x_{\mathrm{L}}y_{\mathrm{L}})10^{n} + (x_{\mathrm{L}}y_{\mathrm{R}} + x_{\mathrm{R}}y_{\mathrm{L}})10^{n/2} + x_{\mathrm{R}}y_{\mathrm{R}}, \end{aligned}$$

and, as before, we may calculate the middle term using one multiplication as

$$x_{\mathrm{L}}y_{\mathrm{R}} + x_{\mathrm{R}}y_{\mathrm{L}} = (x_{\mathrm{L}} + x_{\mathrm{R}})(y_{\mathrm{L}} + y_{\mathrm{R}}) - (x_{\mathrm{L}}y_{\mathrm{L}}) - (x_{\mathrm{R}}y_{\mathrm{R}}),$$

so that we may compute xy with only three multiplications. The multiplication of the $(n/2)$-digit integers is done recursively, using the same kind of splitting, until we arrive at integers with a constant number of digits. (We choose this constant to be 3, so that we are guaranteed that the number of digits decreases in every recursive step.) At this point, we directly multiply the integers.

If the integer n is not even, we split x, y into two integers each, with $\lceil n/2 \rceil = (n+1)/2$ and $\lfloor n/2 \rfloor = (n-1)/2$ digits, and the rest of the operations are modified appropriately. Here, $\lfloor a \rfloor$ denotes the result of rounding *down* the integer a to the closest integer, and $\lceil a \rceil$ denotes the result of rounding *up* a to the nearest integer. If n is even, $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$. Regardless of whether n is even or odd, we have $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$. For $n \geq 4$, we have $\lceil n/2 \rceil + 1 < n$, so that the algorithm computes xy by calculating the product of numbers with strictly fewer digits.

The resulting recursive algorithm for multiplication is summarised below.

```
Algorithm 2: MULTIPLY($x, y$), the Karatsuba multiplication algorithm
  input  : Positive integers $x, y$, each with $n \geq 1$ decimal digits.
  output: The product $xy$.
  if $n \leq 3$ then return $xy$;
  Let $x_{\mathrm{L}}, x_{\mathrm{R}}$ be the leftmost $\lceil n/2 \rceil$ and the rightmost $\lfloor n/2 \rfloor$ digits of $x$, respectively;
  Let $y_{\mathrm{L}}, y_{\mathrm{R}}$ be the leftmost $\lceil n/2 \rceil$ and the rightmost $\lfloor n/2 \rfloor$ digits of $y$, respectively;
  Let $P_1 = $ MULTIPLY($x_{\mathrm{L}}, y_{\mathrm{L}}$);
  Let $P_2 = $ MULTIPLY($x_{\mathrm{R}}, y_{\mathrm{R}}$);
  Let $P_3 = $ MULTIPLY($x_{\mathrm{L}} + x_{\mathrm{R}}, y_{\mathrm{L}} + y_{\mathrm{R}}$);
  return $P_1 \times 10^{2\lfloor n/2 \rfloor} + (P_3 - P_1 - P_2) \times 10^{\lfloor n/2 \rfloor} + P_2$;
```

As we did for the recursive algorithm for exponentiation, we may prove the Karatsuba algorithm to be correct using strong induction on the number of digits. We leave this as an exercise.

An important difference between the recursion here and the recursion in POWER is that we get three smaller subproblems at every step, rather than one. So the algorithm cannot easily be unraveled into an iterative procedure.

Example 2 *Multiply the numbers* 1010203 *and* 3020101 *using the Karatsuba multiplication algorithm.*

Solution : We summarise the calculations made by the algorithm in a table:

Step	x	y	$P_1 = x_{\mathrm{L}} \times y_{\mathrm{L}}$	$P_2 = x_{\mathrm{R}} \times y_{\mathrm{R}}$	$(x_{\mathrm{L}} + x_{\mathrm{R}})$	$(y_{\mathrm{L}} + y_{\mathrm{R}})$
1	1010203	3020101	1010×3020	203×101	1213	3121
2	1010	3020	10×30	10×20	20	50
3	1213	3121	12×31	13×21	25	52

In this table, we have not included separate lines for steps in which the algorithm directly multiplies two numbers with at most 3 digits (e.g., for 203×101 or 10×30). Recall that $P_3 = (x_{\mathrm{L}} + x_{\mathrm{R}})(y_{\mathrm{L}} + y_{\mathrm{R}})$. The calculations we make for xy in every step are:

Step	$P_1 \times 10^{2\lfloor n/2 \rfloor} + (P_3 - P_1 - P_2) \times 10^{\lfloor n/2 \rfloor} + P_2$	xy
1	$3050200 \times 10^6 + (3785773 - 3050200 - 20503)10^3 + 20503$	3050915090503
2	$300 \times 10^4 + (1000 - 300 - 200)10^2 + 200$	3050200
3	$372 \times 10^4 + (1300 - 372 - 273)10^2 + 273$	3785773

So we have $1010203 \times 3020101 = 3050915090503$. ■

Let us turn to the number of elementary operations performed by the algorithm. Let $T(n)$ be (a bound on) the number of elementary operations performed by the algorithm MULTIPLY(x, y) on inputs x, y with n digits. We get the following recurrence for $T(n)$:

$$\begin{aligned} T(3) &= 27 \\ T(n) &= 2\,T(n/2) + T(n/2+1) + 12(n+1), \qquad \text{if } n \text{ is even and } n > 3 \\ T(n) &= 2\,T((n+1)/2) + T((n+1)/2+1) + 12(n+1), \qquad \text{if } n \text{ is odd and } n > 3, \end{aligned}$$

where we have used estimates from the straightforward method of multiplication for 3-digit integers to get $T(3) = 27$, and obtained the term $12(n+1)$ from the 6 additions of integers with at most $4\lceil n/2 \rceil$ digits each in the recursive step.

This is a rather complicated recurrence, but its solution is within a constant factor of the solution of a recurrence described below in which the number of digits is exactly halved at every step. (The proof of this connection is not difficult, but is not illuminating. We leave it to the interested reader to verify.)

Let n be a power of 2, and define $T'(n)$ by the following recurrence.

$$\begin{aligned} T'(1) &= 1 \\ T'(n) &= 3\,T'(n/2) + n, \qquad \text{if } n > 1. \end{aligned}$$

Unraveling this recurrence with $n = 2^k$, for some $k \geq 0$ gives us

$$\begin{aligned} T'(2^k) &= 3\,T'(2^{k-1}) + 2^k \\ &= 3^2\,T'(2^{k-2}) + 3 \times 2^{k-1} + 2^k \\ &= 3^3\,T'(2^{k-3}) + 3^2 \times 2^{k-2} + 3 \times 2^{k-1} + 2^k \\ &\vdots \\ &= 3^k \times 2^0 + 3^{k-1} \times 2 + 3^{k-2} \times 2^2 + \cdots + 3^0 \times 2^k \\ &= 3^k \left[1 + \frac{2}{3} + \left(\frac{2}{3}\right)^2 + \cdots \left(\frac{2}{3}\right)^k \right] \\ &= 3^k \left[\frac{1 - \left(\frac{2}{3}\right)^k}{1 - \frac{2}{3}} \right] \\ &= 3^{k+1} - 2^{k+1}. \end{aligned}$$

Had we been given this solution to the recurrence, we could have verified by induction that $T'(2^k) = 3^{k+1} - 2^{k+1}$ for all $k \geq 0$. We leave this as an exercise. Since $k = \log_2 n$, we have $T'(n) \leq 3^{\log_2 n + 1} = 3\,n^{\log_2 3}$.

As mentioned above, we can show that the solution $T(n)$ to the original recurrence is at most a constant times $T'(n) \leq 3\,n^{\log_2 3} \approx 3\,n^{1.585}$. This dependence on n holds even when n is not a power of 2. Thus, the Karatsuba algorithm uses many fewer elementary operations than $3n^2$, the "cost" of the method we learnt in school.

This is however, not the end of the story. Yet more sophisticated multiplication algorithms have been discovered since. Andrei Toom and Stephen Cook found improvements to the Karatsuba algorithm shortly after it was published. In 1971 Arnold Schönhage and Volker Strassen described a fundamentally new method that is subtantially faster, and in practice beats the previous ones for numbers with more than around $10,000$ decimal digits. The best known algorithm is due to Martin Fürer, and was discovered only very recently, in 2007.