School of Computer Science and Engineering
University of New South Wales
COMP3121/3821/9101/9801
A. Ignjatović
Notes taken by David Greenaway and Alex North

Complexity of Basic Operations on Natural Numbers

We now demonstrate both usefulness and the limitations of the Master Theorem (and the asymptotic estimates in general) by studying the complexity of the basic arithmetic operations.

Addition

A question that we may wish to ask is 'How hard is addition?', assuming that we already have (as a primitive, single step) operation $S(x)$ which gives us the successor of x. A recursive definition of addition in terms of the successor is as follows:

$$0 + x = x$$

$$S(y) + x = S(y + x)$$

In practice, trying to perform addition using this definition would be extremely slow: adding y to x would require y operations. Addition using the 'Primary school' method is much faster: it is linear in the *number of digits* of x and y:

```
  C C C C C      carry
    X X X X X    first integer
+   X X X X X    second integer
  -----------
  X X X X X X    result
```

Multiplication

Like we did with addition, we can come up with a recursive definition of multiplication. (We assume that addition is primitive):

$$0 \cdot x = 0$$

$$(y + 1) \cdot x = y \cdot x + x$$

Once again, execution of such recursive definition of multiplication would require many more recursion steps than methods we used in primary school, and would never be practical to implement. Our basic primary school algorithm requires $O(n^2)$ operations to multiply two n digit numbers.

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ n * n intermediate operations
    X X X X       /
  X X X X        /
---------------
X X X X X X X X  <- 2n length result
```

(We assume that two `X`'s can be multiplied in $O(1)$ time. Each `X` could be a bit or a word in some arbitrary base.)

But can we do multiplication any better? Could multiplication be also doable in linear time in the number of digits? Or is multiplication intrinsically harder than addition?

Prior to the 1960's, it appears that computer scientists believed that multiplication required $O(n^2)$ operations. In 1962, Karatsuba surprised them by producing an algorithm that multiplies two numbers in only $O(n^{log_2 3}) \approx O(n^{1.585})$ steps.

Fast Integer Multiplication

The Karatsuba algorithm for fast multiplication is a simple divide-and-conquer algorithm. We take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

with A_0 being the least significant bits and A_1 being the most significant bits. AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0$$

We now have four multiplications ($A_1 \cdot B_1$, $A_1 \cdot B_0$, $B_1 \cdot A_0$, $A_0 \cdot B_0$), each of which are half the size. Thus, if we proceed by recursion, replacing evaluation of AB by evaluation of these four products, then the total number of operations $T(n)$ to calculate the product of two n bit integers is given by:

$$T(n) = 4T\left(\frac{n}{2}\right) + cn$$

The multiplication by a power of two is 'cheap', in that it involves only bit shifting, and can thus be done in linear time, as well as the necessary additions. Thus, the above is true for a (small) constant c. The term $4T\left(\frac{n}{2}\right)$ comes from the fact that we are performing 4 new multiplications of numbers of half the size $\frac{n}{2}$.

We can find the asymptotic growth rate of the solution to this recurrence using the Master Theorem: since $n^{\log_b a} = n^{\log_2 4} = n^2$ and $f(t) = cn = O(n^{1.5}) = O(n^{2-.5})$, the first case of MT applies and so $T(n) = \Theta(n^2)$. In other words, we have gained nothing by using divide and conquer.

If, however, we could somehow get rid of one of the multiplications so that we only had three multiplications, we could have an algorithm that runs in time satisfying the recurrence

$$T(n) = 3T\left(\frac{n}{2}\right) + cn$$

Since now $n^{\log_a b} = n^{\log_2 3}$, we would have

$$\begin{aligned} T(n) &= \Theta(n^{log_2 3}) \text{ (again by the first case of the Master Thereom)} \\ &= O(n^{1.585}), \end{aligned}$$

which *is* a significant improvement for large n.

The Karatsuba algorithm manages to remove one of the multiplications by relying on the observation that in order to find

$$AB = A_1B_1 2^n + (A_1B_0 + B_1A_0)2^{\frac{n}{2}} + A_0B_0$$

we only need the values of $A_1B_1, (A_1B_0 + B_1A_0)$ and A_0B_0. Thus, three multiplications suffice: one for A_1B_1, one for A_0B_0, and one for $(A_1 + A_0)(B_1 + B_0)$, because we can obtain $A_1B_0 + B_1A_0$ as

$$A_1B_0 + B_1A_0 = (A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0$$

In other words,

$$AB = A_1B_1\ 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

which involves only three multiplications! Thus the Karatsuba's algorithm runs in $n^{\log_2 3}$ many steps.

We now generalize Karatsuba's algorithm. We break up the number into several, rather than just two smaller pieces, as follows. Consider A and B as $n + 1$ blocks of k bits. We will keep n fixed and let only k grow as the size of input numbers increases:

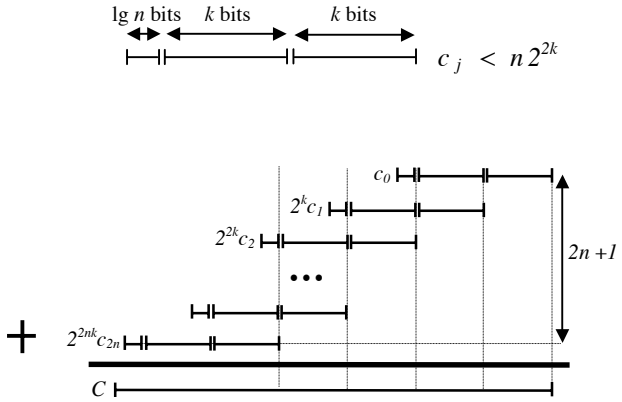
$$A = \underbrace{XXX\ldots XX}_{k \text{ bits of } A_n}\ \underbrace{XXX\ldots XX}_{k \text{ bits of } A_{n-1}}\ldots\underbrace{XXX\ldots XX}_{k \text{ bits of } A_0} = A_n 2^{kn} + A_{n-1}2^{k(n-1)} + \cdots + A_0 2^{k\cdot 0}$$

This means that we are representing A as a linear combination of powers of 2^k, with coefficients $A_i < 2^k$. Consider the naturally corresponding polynomial $A(x) = A_n x^n + A_{n-1}x^{n-1} + \cdots + A_0$. Then $A = A(2^k)$. Similarly we construct $B(x)$ such that $B = B(2^k)$. Thus, $AB = A(2^n)B(2^n) = A(x) \cdot B(x)|_{x=2^k}$.

Let us set $C(x) = A(x) \cdot B(x)$; then $C(x)$ is of degree $2n$ and thus $C(x) = \sum_{j=0}^{2n} c_j x^j$. We want to find $C(2^k)$. We have

$$C(x) = \sum_{j=0}^{2n} c_j x^j = A(x) \cdot B(x) = \sum_{j=0}^{2n} \left(\sum_{i=0}^{j} A_i B_{j-i} \right) x^j \tag{1}$$

where A_i and B_i are set to zero for $i > n$. Clearly, the value $C(2^k)$ can be found from the values of the coefficients c_j of $C(x)$. Since $A_i, B_i < 2^k$, we have that $c_j = \sum_{i=0}^{j} A_i B_{j-i} < n2^{2k}$; Thus, c_j has at most $2k + \lceil \lg n \rceil$ bits, and there are $2n+1$ numbers c_j. Consequently, $C = C(2^k) = \sum_{j=0}^{2n} c_j 2^{kj}$ looks as shown on Figure below, and, since n is kept fixed, it is clear that this sum can be evaluated in linear time in the size of our input numbers, i.e., in time $O(nk) = O(k)$.



Thus, to get our result, we have to find an efficient algorithm for finding the coefficients c_j for $j \leq 2n$, from A_i, B_i, $i \leq n$. *Prima facie,* finding the coefficients of $C(x)$ still involves $(n+1)^2$ multiplications, because all pairs of the form $A_i B_j$, $0 \leq i, j \leq n$ appear in $c_j = \sum_{i=0}^{j} A_i B_{j-i}$.

Convolution

Let $A_n \ldots A_0$ and $B_n \ldots B_0$ be an arbitrary pair of number sequences; lets us pad them with zeros to length $2n+1$, i.e., let us set $A_i = 0$ and $B_i = 0$ for $n < i \leq 2n$. Then the sequence

$$\left\{ \sum_{i=0}^{j} A_i B_{j-i} \right\}_{j=0}^{2n}$$

is called *the (linear) convolution* of the sequences A and B, denoted $A * B$:

$$A * B = \{A_n B_n, \ldots, A_2 B_0 + A_1 B_1 + A_0 B_2, A_1 B_0 + A_0 B_1, A_0 B_0\}$$

From equation (1) we see that the coefficients of the product of two polynomials are obtained as a linear convolution of the coefficients of these two polynomials. Thus, we need efficient algorithms for evaluating linear convolution of two sequences.

Coefficient vs value representation of polynomials

Every polynomial $A(x)$ of degree n is uniquely determined by its values at $n+1$ distinct input values for x:

$$A(x) \leftrightarrow \{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_n, A(x_n))\}$$

If $A(x) = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_0$, we can write in matrix form:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_n) \end{pmatrix}. \tag{2}$$

The determinant of the above matrix is the Van Der Monde determinant, and if all x_i are distinct, it is non-zero:

$$\text{Det}\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} = \begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{vmatrix} = \prod_{i \neq j}(x_i - x_j) \neq 0. \tag{3}$$

Thus, if all x_i are distinct, given any values $A(x_0), A(x_1), \dots, A(x_n)$ the coefficients $A_0, A_1, \dots, A_n$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_n) \end{pmatrix} \tag{4}$$

Similar is true for the second polynomial $B(x)$:

$$B(x) \leftrightarrow \{(x_0, B(x_0)), (x_1, B(x_1)), \dots, (x_n, B(x_n))\}$$

However, if $A(x)$ and $B(x)$ are of degree n, the product polynomial $C(x) = A(x)B(x)$ is of degree $2n$, and to uniquely determine it, we need $2n + 1$ of its values:

$$C(x) = A(x) \cdot B(x) \leftrightarrow \{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \dots, (x_{2n}, A(x_{2n})B(x_{2n}))\}$$

Thus, we will 'overdetermine' $A(x)$ and $B(x)$ by starting with $2n + 1$ values of these two polynomials:

$$A(x) \leftrightarrow \{(x_0, A(x_0)), (x_1, A(x_1)), \dots, (x_{2n}, A(x_{2n}))\}$$

$$B(x) \leftrightarrow \{(x_0, B(x_0)), (x_1, B(x_1)), \dots, (x_{2n}, B(x_{2n}))\}$$

$$C(x) = A(x) \cdot B(x) \leftrightarrow \{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \dots, (x_{2n}, A(x_{2n})B(x_{2n}))\}$$

We will then use these $2n + 1$ values of $C(x)$ to find the coefficients c_i, $0 \leq i \leq 2n$; this is called *interpolation*. Finding the values at certain set of points (knowing the coefficients of the polynomial) is called *evaluation*.

Thus, to find the coefficients of a polynomial of order $2n$ we need only find its values at $2n+1$ points, and, instead of looking at values for large x like 2^k, we can choose small values of x. This is the main idea behind this method: we evaluate $A(x) \cdot B(x)$ at the $2n+1$ smallest size integer inputs we can, namely for $x \in \{-n, -n+1 \ldots, 0, \ldots, n-1, n\}$.

Thus, we want to find the values

$$A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

Recall that in our algorithm we *fix* n (number of groups in the partition of larger numbers) and increase only k (the size - number of digits - in each group A_i). Thus, the values m^j, $-n \leq m \leq n$, $0 \leq j \leq n$ are constant, regardless of the size of the input numbers. As k increases, multiplication by a constant is *linear* in k:

```
 k bits     fixed number of bits
xxxxxxxxxx  .  xxx
------------------
 xxxxxxxxxx
xxxxxxxxxx          <= number of rows is fixed
 xxxxxxxxx
====================
xxxxxxxxxxxx
```

So these evaluations are all done in linear time, since all multiplications are by a constant from the interval $[-n^n, n^n]$.

Thus, we can find in linear time the values

$$A(-n), A(-n+1), \ldots, A(0), \ldots, A(n)$$

$$B(-n), B(-n+1), \ldots, B(0), \ldots, B(n)$$

So we can multiply them using in total $2n+1$ large number multiplications to get

$$
\begin{aligned}
C(-n) &= A(-n) \cdot B(-n) \\
C(-n+1) &= A(-n+1) \cdot B(-n+1) \\
&\vdots \\
C(0) &= A(0) \cdot B(0) \\
&\vdots \\
C(n-1) &= A(n-1) \cdot B(n-1) \\
C(n) &= A(n) \cdot B(n)
\end{aligned} \tag{5}
$$

Since $C(x) = c_{2n}x^{2n} + c_{2n-1}x^{2n-1} + \cdots + c_0$, we have

$$
\begin{aligned}
c_{2n}(-n)^{2n} + \cdots + c_0 &= C(-n) \\
c_{2n}(-(n-1))^{2n} + \cdots + c_0 &= C(-(n-1)) \\
&\vdots \\
c_{2n}(n)^{2n} + \cdots + c_0 &= C(n)
\end{aligned}
$$

which is just a system of linear equations, that can be solved for $c_0, c_1, \ldots, c_{2n}$:

$$
\begin{pmatrix}
1 & -n & (-n)^2 & \ldots & (-n)^{2n} \\
1 & -(n-1) & (-(n-1))^2 & \ldots & (-(n-1))^{2n} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
1 & n & n^2 & \ldots & n^{2n}
\end{pmatrix}
\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n} \end{pmatrix}
=
\begin{pmatrix} C(-n) \\ C(-(n-1)) \\ \vdots \\ C(n) \end{pmatrix},
$$

i.e.,

$$
\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2n} \end{pmatrix}
=
\begin{pmatrix}
1 & -n & (-n)^2 & \ldots & (-n)^{2n} \\
1 & -(n-1) & (-(n-1))^2 & \ldots & (-(n-1))^{2n} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
1 & n & n^2 & \ldots & n^{2n}
\end{pmatrix}^{-1}
\begin{pmatrix} C(-n) \\ C(-(n-1)) \\ \vdots \\ C(n) \end{pmatrix}.
$$

Thus, all c_j, $0 \leq j \leq 2n$, are just linear combinations (with constant coefficients which are the entries of the inverse matrix above) of $C(-n), C(-n+1), \ldots, C(0), \ldots, C(n-1), C(n)$, and thus can be evaluated in time $O(k)$.

Consequently, the only hard part of the recursion step are $2n+1$ large number multiplications appearing in (5), and we must evaluate the number of steps involved.

Notice that for each m such that $-n \leq m \leq n$, the value of $A(m)$ is a sum of $n+1$ values, each a product of a k-bit number A_j with a constant of the form m^j, $-n \leq m \leq n$:

$$A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

So $|A(m)| \leq (n+1) \cdot 2^k \cdot n^n$. Thus, each $A(m)$ has

$$\log_2 \left((n+1)2^k n^n\right) = k + \log_2 \left((n+1) \cdot n^n\right)$$

and $(n+1) \cdot n^n$ is constant; i.e, each $A(m)$ has $k+s$ bits, where s is a constant independent of k (where k is the number of bits in each A_j, $j \leq n$.

So we have reduced a multiplication of a pair of $k(n+1)$ digit numbers to $2n+1$ multiplications of pairs of $k+s$ digit numbers of the form $A(j) \cdot B(j)$, $-n \leq j \leq n$, and thus we get the following recurrence for the complexity of multiplication T:

$$T((n+1)k) = (2n+1)T(k+s) + \Theta(k)$$

Let $N = (n+1)k$. Then, since n is constant, $\Theta(k)$ can be replaced by $\Theta(N)$ and we get

$$T(N) = (2n+1)T\left(\frac{N}{n+1} + s\right) + \Theta(N)$$

Since s is constant, its impact can be neglected. Thus, with $a = 2n+1$ and $b = n+1$ the Master Theorem gives:

$$\begin{aligned} T(N) &= \Theta\left(N^{\log_b a}\right) \\ &= \Theta\left(N^{\log_{n+1}(2n+1)}\right) \end{aligned}$$

Note that

$$\log_{n+1}(2n+1) < \log_{n+1}(2(n+1)) = \log_{n+1} 2 + \log_{n+1}(n+1) = 1 + \log_{n+1} 2 = 1 + \frac{1}{\log_2(n+1)}$$

Thus,

$$T(N) = \Theta\left(N^{1+\frac{1}{\log_2(n+1)}}\right)$$

Since Let $1/\log_2(n+1)$ can be made arbitrarily small by increasing n, it appears that by increasing the number n of pieces into which we break our large numbers and apply recursive multiplication as described above, we can obtain an algorithm that runs as close to linear time as we wish!! Is this really so???

Well, theoretically yes, but practically NO! We now want to explain the pitfalls of trusting an asymptotic analysis carelessly.

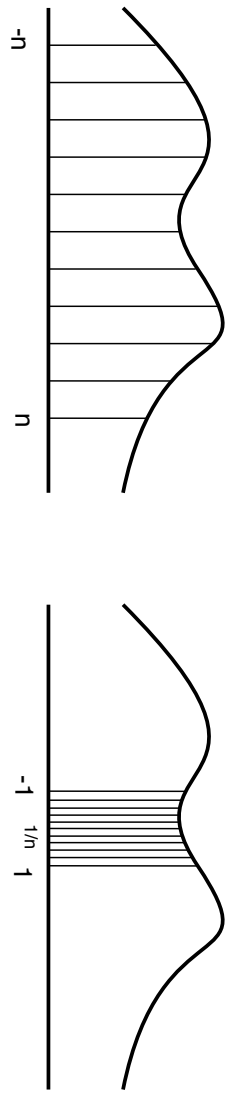
Let us denote $\varepsilon = \frac{1}{\log_2(n+1)}$. If we want $\epsilon = 1/100$, we need $1/\log_2 n < 1/100$, i.e, we need $n > 2^{100}$.

We skimmed over the multiplications by constants related to n, but as n increases the cost of multiplication by these constants grows massively! The constants used are in the range $[-n^n, n^n]$; Thus, not only does n increases fast as ε decreases, but n^n grows extremely fast, and with it the constants involved in the above asymptotic bound. This explains why the above result has no practical value: as n increases, the constants involved in the asymptotic bound are so large that the algorithm becomes efficient only for unreasonably large numbers!! The moral is, that asymptotic analysis tells little about the efficiency of our algorithm unless it is accompanied with a careful estimate of the sizes of the constants involved in the asymptotic estimates!

Alternative interpolation ranges

We might not have to use integers to interpolate. We could use reals, say in $[-1, 1]$. But then we have a problem of precision. As n increases, the values that occur in evaluation of $A[1/n]$ become extremely small (e.g., $(1/n)^n$).

Thus, we need a set of numbers all with magnitude 1, and for this we need complex numbers on the unit circle, and the FastFourier Transform, the FFT, which we introduce



next week.

As a brief history of fast integer multiplication, we mention that it seems that Karatsuba did not realize that for every $\epsilon > 0$ there is a simple modification of his algorithm which multiplies two n digit numbers in only $O(n^{1+\epsilon})$ steps.

While this result seems quite impressive ($n^{1.000001}$ is super-linear, but still pretty close to linear), as we saw, in practice, as ϵ decreases, the constants involved in the algorithm increase dramatically, thus also making the constants in its asymptotic bound huge, rendering the algorithm useless for practice. Thus, such $O(n^{1.000001})$ algorithm has only theoretical interest.

Subsequently in 1968 Strassen designed an algorithm that uses the Fast Fourier Transform (FFT) and representation of complex roots of unity with sufficiently high precision, that multiplies integers in time $O(n \lg n (\lg \lg n)^{1+\varepsilon})$ for every $\varepsilon > 0$. Shönhage and Strassen improved this result in 1971, producing a multiplication algorithm running in time $O(n \log(n) \log \log(n))$. They eliminated floating point arithmetic by using FFT over a finite ring. The algorithm asymptotic has reasonably small constants and is practically useful. Some simplifications of their algorithm yield only marginally worse asymptotic time complexity, and even seem more practical.