COMP 251: ARITHMETIC OPERATIONS

OMAR FAWZI

In this lecture, we are going to consider the complexities of basic arithmetic operations, namely integer addition and multiplication. As always when analysing the asymptotic cost of algorithms, the motivation is to understand how much harder does the problem get when the inputs get larger.

1. Integer addition

Probably the first non-trivial algorithm you learned is how to add two numbers. Given two arbitrarily large numbers, in principle, if you are given all the time you need, you would be able to compute their sum.

What is the cost of this algorithm? To answer this question, we need to define the problem formally and define what we mean by cost.

Addition

- Input: Two integers x and y represented by n decimal digits each.
- Output: The decimal representation $x + y$.

Note that $0 \leq x, y \leq 10^n - 1$, so $x + y$ has at most $n + 1$ digits. So our problem takes two inputs of size n and outputs a string of length $n + 1$. It is important to note that we are measuring the cost with respect to the size of the representation of the numbers and not the numbers themselves. We measure the cost of an algorithm by counting how many operations it does on digits (this is reasonable as the number of operations on digits is independent of the growth of the input to our actual problem, so we can consider it as constant time). In this model, it is easy to see the cost of the standard addition algorithm is linear in n.

Is it possible to do better? The size of the input is linear in n, and we have to take into account each digit for our computation, as changing one digit will change the output. Therefore, we have to do at least a linear number of elementary operations to compute the output. So the algorithm we know has an optimal asymptotic growth.

2. Integer multiplication

2.1. **Simple algorithm.** We begin by defining the problem

Multiplication

- Input: Two integers x and y represented by n decimal digits each.
- Output: The decimal representation $x \times y$. (Has at most $2n$ digits).

Date: 19th February 2009.

Our vague memories from elementary school says that multiplication is harder and longer than addition. Let us analyse the computational cost of the algorithm we used. Each digit of y gets multiplied with x, and this takes linear time in the size of x. As a result, the cost of this algorithm is quadratic in n (its running time is $\Theta(n^2)$).

And we ask the same question, is it possible to do better? The same argument as the one used for addition tells us that we cannot hope for an algorithm that runs in less than $\Theta(n)$, but there is a huge gap between $\Theta(n)$ and $\Theta(n^2)$. It turns out we can do much better.

2.2. **Karatsuba's algorithm.** In 1962, Karatsuba found an algorithm using divide and conquer whose running time is less than quadratic which was published in [2]. When we want to apply divide and conquer, we try to solve our problem using solutions of smaller instances of the problem. For this we decompose our input into the right $n/2$ digits and the left $n/2$ digits (we suppose that n is even for now), which can be written

$$\begin{aligned} x &= a + b \cdot 10^{n/2} \\ y &= c + d \cdot 10^{n/2}. \end{aligned}$$

Then,

$$x \times y = (a + b \cdot 10^{n/2})(c + d \cdot 10^{n/2}) = ac + (ad + bc) \cdot 10^{n/2} + bd \cdot 10^n.$$

We need to compute

$$\begin{aligned} z_1 &= ac \\ z_2 &= ad + bc \\ z_3 &= bd. \end{aligned}$$

Remember that a, b, c and d are $n/2$ digit numbers. A natural way to compute z_1, z_2, z_3 is to compute the products (of numbers of size $n/2$) ac, ad, bc, bd recursively and then compute z_2 at the cost of an addition. This means that we reduced a multiplication of size n into 4 multiplications of size $n/2$ plus some additions, which have linear cost. We can thus write a recurrence relation for the cost of this algorithm as follows

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

By the master theorem, the cost of the algorithm is $T(n) = \Theta(n^2)$, we get back the same quadratic running time. But there may be room for improvement, if we are able to compute the values z_1, z_2, z_3 using fewer recursive calls. Remember that addition can be considered as free, as it only takes linear time, so given a, b, c, d, we want to compute z_1, z_2, z_3 using as few multiplications as possible and as many additions as we need. If we compute

$$
\begin{aligned}
p_1 &= ac \\
p_2 &= (a+b) \times (c+d) \\
p_3 &= bd
\end{aligned}
$$

we get

$$
\begin{aligned}
z_1 &= p_1 \\
z_2 &= p_2 - p_1 - p_3 \\
z_3 &= p_3
\end{aligned}
$$

You may argue that we only gained one operation, but as we apply the algorithm recursively, this reflects in the number of the recursive calls, and as a result in the **exponent** of the running time.

The algorithm can be written as follows:

```
FastMult($x, y, n$)
  • Pad $x$ and $y$ with leading 0's so that their size is $2^k$, where $2^k$ is the smallest power
    of two larger than $n$
  • Return Mult($x, y, 2^k$)
  • Mult($\alpha, \beta, m$)
      – If $m = 1$, return $\alpha \times \beta$
      – Else
          ∗ Decompose $\alpha = a + b \cdot 10^{m/2}$ and $\beta = c + d \cdot 10^{m/2}$
          ∗ Compute $p_1 \leftarrow$ Mult($a, c, m/2$), $p_2 \leftarrow$ Mult($a+b, c+d, m/2$), $p_3 \leftarrow$
            Mult($b, d, m/2$)
          ∗ Compute $z_1 \leftarrow p_1$, $z_2 \leftarrow (p_2 - p_1 - p_2)$, $z_3 \leftarrow p_3$
          ∗ Add 0's to compute $z_2 \leftarrow z_2 \cdot 10^{m/2}$, $z_3 \leftarrow z_3 \cdot 10^m$
          ∗ Return $z_1 + z_2 + z_3$.
```

We argued the correctness of this algorithm, now we try to analyse the running time. Mult called with size n consists in 3 recursive calls to size $n/2$. Thus the running time $T(n)$ of this algorithm on inputs of size n verifies

$$T(n) = 3T(n/2) + \Theta(n). \tag{1}$$

By the master theorem, this gives $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$, which is a big improvement over the simple quadratic algorithm. Moreover, the algorithm is quite simple to implement, and the hidden constants are small, so in practice you don't have to take too large number to see Karatsuba's algorithm beat the simple algorithm.

Technical detail: We did a small mistake in our algorithm, the product $p_2 = (a+b) \times (c+d)$ may be a product of $n/2+1$-digit numbers, as $a+b$ may have $n/2+1$ digits. We disregarded this minor issue as it leads to a more complicated recurrence relation. One way of solving this problem is to forget about the $n/2+1$st digit of $a+b$ and $c+d$ and add them separately. More precisely, if we write $a+b = l_1 \cdot 10^{n/2} + r_1$ and $c+d = l_2 \cdot 10^{n/2} + r_2$. Then $(a+b) \times (c+d) = l_1 l_2 \cdot 10^n + (l_1 r_2 + l_2 r_1) \cdot 10^{n/2} + r_1 r_2$. Because l_1 and l_2 are digits, the first three terms can be computed in linear time, and the last one using a recursive call. For this modified algorithm, the recurrence relation 1 is correct, and we get the wanted running time.

2.3. **Even faster multiplication with Fast Fourier Transform (FFT).**

There is still a gap between our lower bound $\Omega(n)$ and this algorithm whose running time is $\Theta(n^{1.585})$. There is an algorithm for integer multiplication due to Schönhage and Strassen in [3] that runs in $O(n \log n \log \log n)$, but it is beyond the scope of this course. However, the ideas used are interesting, namely the use of FFT. In this section, we present the Fast Fourier Transform and how to apply FFT to polynomial multiplication.

Computer scientists usually say the main use of FFT is for polynomial multiplication, but it is at least equally useful to quickly compute Fourier transforms, as its name suggests. The ideas of the FFT algorithm date back to Gauss in 1805, who used it to interpolate trajectories of some asteroids, but it became popular after by Cooley and Tukey published an article [1] in 1965 describing the algorithm clearly. The reported story of this paper is that Tukey first came up with the idea to apply it to the detection of nuclear tests in the Soviet Union, and Cooley implemented the idea for a completely different problem: the analysis of 3d crystallographic data.

The history of the FFT already shows how diverse applications it may have. It is now heavily used in many domains, like digital signal processing, audio compression, polynomial multiplication... etc.

2.3.1. *The Discrete Fourier Transform (DFT).*

Let us define the DFT.

DFT

- Input: vector (signal) of real (or complex) numbers $x[0], x[1], \ldots, x[n-1]$.
- Output: vector of complex numbers

$$\hat{x}[j] = \sum_k x[k] e^{-\frac{i2\pi jk}{n}}, 0 \le j \le n-1$$

As you can see, the DFT uses complex numbers, so we list basic some facts about complex number.

Quick review of complex numbers:

- i is defined as $i^2 = -1$.
- A complex number $z = a + ib$ where a and b are reals.
- $e^{ix} = \cos x + i \sin x$, these points lie on the unit circle (see figure 1).
- The complex numbers $e^{\frac{i2\pi j}{n}}$ for $0 \le j \le n-1$ are called the n-th roots of unity.

2.3.2. *Computing the DFT efficiently.*

The input vector is a sequence of n real numbers. So, here, our unit will be a real number (in the previous sections, the unit was a digit). Consequently, we suppose that we can apply operations on real and complex numbers at unit cost, so we measure the cost by the number of complex operations used by our algorithm, this measure is sometimes called the *arithmetic complexity*.

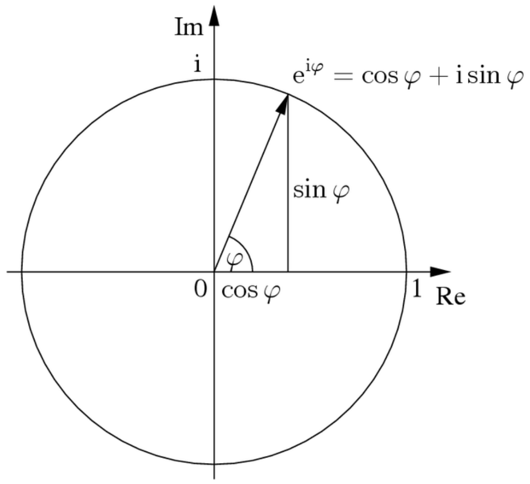

FIGURE 1. The complex unit circle (figure taken from Wikipedia article on Euler's Formula)

The simplest way of computing the DFT would be to compute the elements $\hat{x}[j]$ of the vector one by one. This gives an algorithm whose cost is $\Theta(n^2)$.

The main idea is to observe that the DFT of x can be computed from the DFTs of 2 vectors with half the size of x (in other words, find a divide and conquer approach). This can be seen by separating the elements of x of odd even and odd index, (we'll suppose that n is a power of 2, $n = 2^k$)

$$\begin{aligned}\hat{x}[j] =& x[0] + x[2]e^{-\frac{i2\pi j}{n/2}} + x[4]e^{-\frac{i2\pi j\cdot 2}{n/2}} + \cdots + x[n-2]e^{-\frac{i2\pi j\cdot(n-2)/2}{n/2}} \\ &+ x[1]e^{-\frac{i2\pi j}{n}} + x[3]e^{-\frac{i2\pi j}{n}}e^{-\frac{i2\pi j}{n/2}} + \cdots + x[n-1]e^{-\frac{i2\pi j}{n}}e^{-\frac{i2\pi j\cdot(n-2)/2}{n/2}}\end{aligned}$$

Let us name the vector of even indices y, and odd indices z, such that $y[j] = x[2j]$ and $z[j] = x[2j+1]$ for $0 \le j \le n/2 - 1$, then we see that

$$\hat{x}[j] = \hat{y}[j] + e^{-\frac{i2\pi j}{n}}\hat{z}[j], 0 \le j \le n/2 - 1.$$

and for $j \ge n/2$ we use the identity $e^{-\frac{i2\pi(j+n/2)}{n/2}} = e^{-\frac{i2\pi j}{n/2}}e^{-i2\pi} = e^{-\frac{i2\pi j}{n/2}}$. As a result,

$$\hat{x}[j] = \hat{y}[j-n/2] + e^{-\frac{i2\pi j}{n}}\hat{z}[j-n/2], n/2 \le j \le n-1.$$

Given the DFT of y and z, we can thus compute the DFT of x in linear time. Using this relationship, we can build an easy recursive algorithm that computes the DFT. This algorithm is the simplest form of the Fast Fourier transform. The number of arithmetic operations required by this algorithm verifies:

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

This is exactly the same recurrence relation as merge sort, we conclude that $T(n) = \Theta(n \log n)$.

2.3.3. *Link to multiplication.* Now let us get to the link between the Fourier transform and multiplication, more precisely polynomial multiplication.

Recall polynomial multiplication, if $P = p_0 + p_1 X + \cdots + p_{n-1} X^{n-1}$ and $Q = q_0 + q_1 X + \cdots + q_{n-1} X^{n-1}$, then by letting $p_j = q_j = 0$ for $j \geq n$,

$$PQ = p_0 q_0 + (p_0 q_1 + p_1 q_0) X + \cdots + \sum_{0 \leq k \leq j} p_k q_{j-k} X^j + \cdots + q_{n-1} p_{n-1} X^{2n-2}.$$

We can define the polynomial multiplication problem as follows:

Polynomial multiplication

- Input: Polynomials P and Q of degree at most $n-1$, represented by their coefficients $(p_0, p_1, \ldots, p_{n-1})$ and $(q_0, q_1, \ldots, q_{n-1})$.
- Output: PQ represented by the list of its coefficients $(c_0, c_1, \ldots, c_{2n-2})$.

Note that if we let the coefficients p_j and q_j be digits (i.e. having value ≤ 9) and replace X by 10, then the representations of P and Q by their coefficients are exactly the decimal representation of the corresponding number $P(10)$ and $Q(10)$. Moreover, in this case, the coefficients of PQ almost make up the decimal representation the product $P(10) \times Q(10)$. The only difference is in the carry. In fact the coefficients $\sum_{0 \leq k \leq j} p_k q_{j-k}$ of PQ may be greater than 9. But we can see that integer multiplication and polynomial multiplication are related, and that polynomial multiplication is likely to be easier, as we don't have to care about carries.

Now we try to use FFT to obtain a fast algorithm for polynomial multiplication. One trick is to change representation, we want a representation of polynomials for which it is easy to multiply two representations. Remember that a polynomial may be determined in other ways. For example, a polynomial P of degree at most $n-1$ is completely determined by the values it takes on n (or more) distinct numbers $a_0, \ldots, a_{n-1}$ (if you don't remember this fact, it comes from the fact that a polynomial of degree $n-1$ cannot have more than $n-1$ zeros). So P can be represent as well with the list $(P(a_0), P(a_1), \ldots, P(a_{n-1}))$. Now observe that if we fix the values $a_0, \ldots, a_m$, and represent our polynomials of degree less than $n-1$ in this way, multiplication is very easy. In fact, the representation of PQ is simply

$$PQ(a_j) = P(a_j) \cdot Q(a_j)$$

which only requires a linear number of operations in m. Note that we will have to choose m to be at least $2n-1$, so that the product PQ is completely determined. But this is still linear in n.

So our strategy to multiply polynomials P and Q given in standard representation (with coefficients) will be to evaluate the polynomials P and Q in $2n$ well chosen points of the complex plane, then we immediately have the values that PQ takes on these points, and then we get back to the coefficient representation of PQ by doing the inverse transformation.

The points we will choose are the $2n$-th roots of unity, i.e. the complex numbers $e^{-\frac{i2\pi j}{2n}}$, for $0 \leq j \leq 2n-1$. Observe that the evaluation vectors $P(e^{-\frac{i2\pi j}{2n}})$ we want to compute look very similar to the discrete Fourier transforms of $(p_0, p_1, \ldots, p_{n-1})$ and $(q_0, q_1, \ldots, q_{n-1})$, in fact these evaluations are the Fourier transforms of the vectors $(p_0, p_1, \ldots, p_{n-1}, 0, \ldots, 0)$ and $(q_0, q_1, \ldots, q_{n-1}, 0, \ldots, 0)$ of size $2n$ (just write the definition of the DFT).

So the evaluations can be computed with $O(n \log n)$ arithmetic operations. Now it only remains to do the inverse transformation, that is to compute the coefficients of PQ given its values on the $2n$-th roots of unity (this is called *interpolation* in general). This looks harder, but it turns out that for the points we chose (the roots of unity), interpolation is almost the same as evaluation, so, roughly, all we have to do is to compute the Fourier transform of the vector $(PQ(1), PQ(e^{-\frac{i2\pi j}{2n}}), \ldots, PQ(e^{-\frac{i2\pi j}{2n}}))$. In other words, the direct Fourier transform and the inverse Fourier transform are almost the same operation.

In fact, let us see what we get when we compute the Fourier transform of the Fourier tranform of x:

$$\begin{aligned}
\hat{\hat{x}}[l] &= \sum_{j=0}^{2n-1} \sum_{k=0}^{2n-1} x[k] e^{-\frac{i2\pi jk}{2n}} e^{-\frac{i2\pi jl}{2n}} \\
&= \sum_{k=0}^{2n-1} x[k] \sum_{j=0}^{2n-1} (e^{-\frac{i2\pi(k+l)}{2n}})^j \\
&= \sum_{k=0, k+l \neq 2n}^{2n-1} x[k] \frac{e^{-i2\pi(k+l)} - 1}{e^{-\frac{i2\pi(k+l)}{2n}} - 1} + x[2n-l] \cdot 2n \\
&= 2nx[n-l].
\end{aligned}$$

So by computing the FFT of the vector $(PQ(1), PQ(e^{-\frac{i2\pi}{2n}}), \ldots, PQ(e^{-\frac{i2\pi(2n-1)}{2n}}))$, rearranging coefficients and dividing by $2n$, we get the coefficients of the polynomial PQ.

Finally, this gives an algorithm that computes the product of two polynomials of degree at most $n-1$ using $O(n \log n)$ arithmetic operations.

References

[1] Cooley, J. and Tukey, J. An algorithm for the machine calculation of complex Fourier series. Math. Comput. 19, 297-301, 1965.

[2] Karatsuba, A. and Ofman, Yu. Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk SSSR 145, 293-294, 1962. Translation in Physics-Doklady 7, 595-596, 1963.

[3] Schönhage, A. and Strassen, V. Schnelle Multiplikation grosser Zahlen. Computing, vol. 7, 281-292, 1971.