Introduction to mathematical cryptography
PCMI 2022 Undergraduate Summer School

Christelle Vincent

July 18, 2022

Contents

1 Computational complexity **1**
1.1 How to measure complexity 1
1.2 Easy and hard problems 5

1 Computational complexity

As mentioned in the first lecture, the fundamental ideal behind cryptography is that there are operations that are **easy** to do but **hard** to undo. In this section we make this notion more precise.

1.1 How to measure complexity

In mathematics, an **algorithm** is a finite sequence of instructions or computations that, when performed, yield a result. For example, you might have learned to multiply integers in school by applying the so-called "schoolbook algorithm" for multiplication. Notice that here the algorithm is not just "multiplying two integers" but the specific process by which the answer to the multiplication problem is obtained. This might seem surprising, but there are other algorithms to multiply integers that have completely different steps to reach the same answer![1]

Given a specific algorithm, the **computational complexity** of that algorithm is how much resources it takes to accomplish the computation. The word "resources" is purposefully vague here: In this course we will almost exclusively talk about either time complexity or arithmetic complexity, which counts either how much time it takes for the algorithm to

[1] If you are interested, you can look up the Karatsuba algorithm or the family of Toom-Cook algorithms, which are a generalization of the Karatsuba algorithm.

complete, or how many arithmetic operations must be performed to complete the algorithm. Another quantity that is often of interest is the amount of memory or storage necessary to perform a computation, but that will not come up for us except possibly in passing.

In the case of our example, the schoolbook algorithm for multiplication (or "schoolbook multiplication" for short), it's perhaps most natural to consider the arithmetic complexity of the algorithm, by which here we mean the total number of additions and multiplications necessary to perform the computation. For example, to compute 15×6, we would:

1. Multiply $5 \times 6 = 30$, then
2. multiply $1 \times 6 = 6$, then
3. add $6 + 3 = 9$,

using 3 operations to obtain the answer 90.

Pretty quickly, it becomes clear that even if we always apply the schoolbook multiplication algorithm perfectly, the number of steps necessary to perform the computation depends on the specific numbers that we are multiplying. Numbers with more digits take a lot more steps to multiply, but even two pairs of numbers with factors of the same size might not take exactly the same number of steps because of carries which introduce extra additions (here, by a "carry" I mean the addition in the problem 15×6, where the 3 tens from the units multiplication get added to the 6 tens from the tens multiplication). Since the number of arithmetic steps we need to perform depends on the numbers being multiplied, what could we possibly mean by the complexity of the whole algorithm?

After more experimentation with various numbers, we might notice that the main factor that influences the number of steps in a multiplication problem is the **size** of the numbers being multiplied. By this we mean that while adding the carries does introduce some extra operations here and there, "most" of the operations we perform are multiplications, so integers with the same numbers of digits require roughly the same number of operations to multiply.

This will turn out to be the case for the majority of the algorithms we talk about, so we will, from now on, always count the number of arithmetic steps of an algorithm as a function of the size of the input integers.

Definition 1.1. Let n be a positive integer. Then its **size** k is the number of digits in its decimal expansion. This is given by the expression

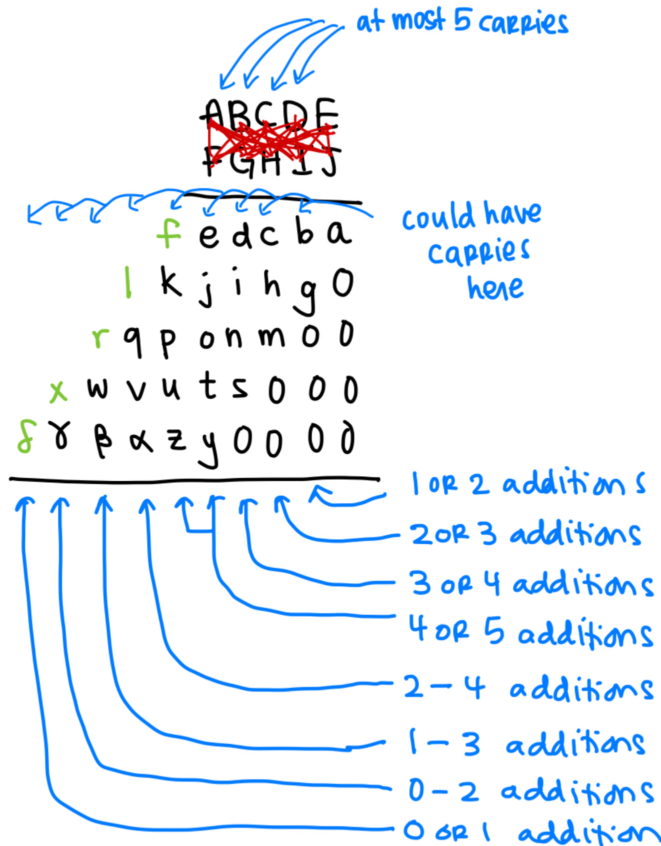
$$k = \lfloor \log_{10} n \rfloor + 1,$$

where $\lfloor \cdot \rfloor$ is the floor function. Often computing books use instead the number of bits in the binary expansion of the integer, which is given by the formula

$$\lfloor \log_2 n \rfloor + 1.$$

We will see that for our purposes we can use either notion of size interchangeably.

To see exactly how the number of steps in the schoolbook multiplication algorithm depends on the size of the integers being multiplied, let's consider multiplication of two five-digit integers. The image below shows what we mean:



Let's count the operations, noting the ones that would be done for any five-digit multiplication, and the ones that only occur sometimes:

- There are always twenty-five multiplications, as each digit must multiply each digit.
- There are also always at least sixteen additions in the column additions at the bottom.
- Carries during multiplications can add up to 5 additions for each of the four positions, so up to twenty additions.
- The column additions at the bottom could have up to thirteen additional additions.

In total there are therefore 41 operations that must always be performed, and up to 33 additional additions depending on carries. Therefore if $f(5)$ is the number of steps it takes to multiply two five-digit integers using schoolbook multiplication, we have

$$41 \leq f(5) \leq 74.$$

Compare this to multiplying two six-digit integers, when there are 61 operations that must always be performed, and up to 46 additional additions depending on carries. In general, if $f(k)$ is the number of steps that it takes to multiply two integers each with k digits using schoolbook multiplication, then we have

$$2k^2 - 2k + 1 \leq f(k) \leq 3k^2 - 1,$$

where

$$2k^2 - 2k + 1 = k^2 + \frac{k(k-1)}{2} + \frac{(k-1)(k-2)}{2}$$

and

$$3k^2 - 1 = k^2 + \frac{k(k-1)}{2} + \frac{(k-1)(k-2)}{2} + k(k-1) + k + 2(k-1).$$

Therefore one can say that multiplying two k-digit integers using schoolbook multiplication takes at least k^2 steps, but no more than $3k^2$ steps.

This is already pretty neat, but we do even say a bit more. This is because when k is very large, the difference between k^2 and $3k^2$ remains the same (one number is 3 times as big as the other). That's a "cost" that's built into our algorithm and which doesn't depend on the size of the numbers we are multiplying, and therefore we don't always want to keep track of it. It's a lot easier to remember that schoolbook multiplication takes "about" k^2 steps, since that's the number that will give you a sense of how long a multiplication problem will take.

To formalize what we mean by "about" k^2 steps, we need some notation. First we need a notion for a function that eventually becomes smaller than a multiple of another function:

Definition 1.2. Let f and g be two functions taking as input positive integers, and outputting positive integers. We will write this as $f, g\colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \ll g$$

if there are positive constants c and C such that

$$f(k) \leq cg(k) \qquad \text{for all } k \geq C.$$

The expression "$f \ll g$" is read "f is less than less than g."

We will also need a notion for a function that eventually becomes bigger than a multiple of another function:

Definition 1.3. Let $f, g\colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \gg g$$

if there are positive constants c and C such that

$$f(k) \geq cg(k) \qquad \text{for all } k \geq C.$$

The expression "$f \gg g$" is read "f is greater than greater than g."

These two notions together allow us to define when two functions are eventually "about the same magnitude:"

Definition 1.4. Let $f, g\colon \mathbb{N} \to \mathbb{N}$. We write that

$$f \sim g$$

if we have that

$$f \ll g \quad \text{and} \quad f \gg g.$$

The expression "$f \sim g$" is read "f is of the order of g."

There is often an easy way to determine if $f \sim g$:

Proposition 1.5. Let $f, g\colon \mathbb{N} \to \mathbb{N}$. Then if the limit

$$\lim_{k\to\infty} \frac{f(k)}{g(k)}$$

exists and is nonzero, $f \sim g$.

Now with this notation, we can say that if f is the number of steps it takes to multiply two k-digit integers, then $f \sim k^2$. Accordingly, we usually say that schoolbook multiplication can be accomplished in "quadratic time" since the number of steps f is of the order of a quadratic polynomial.

1.2 Easy and hard problems

Now that we have a way to express about how many steps it takes to perform a computation, we can talk about "easy" and "hard" problems. First, again here we must make precise that we cannot really talk about the complexity of solving a problem without having an algorithm in mind for solving that problem. However, we commonly will say that a problem is "easy" or "hard" depending on the number of steps it takes to solve the problem using the most efficient known algorithm.

In addition, here all of our problems are solved by algorithms, we assume that we can accomplish all the steps, so an "easy" problem is simply one that we can solve quickly, and a "hard" problem is one that takes an unreasonable amount of time to solve.

More precisely:

Definition 1.6. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **polynomially** if there are positive constants a and b such that

$$k^a \ll f(k) \ll k^b.$$

If the number of steps in an algorithm, when given as a function of the size of the inputs to the algorithm, grows polynomially, then we say that this algorithm is **fast**.
If a problem can be solved by a known fast algorithm, then we say that this problem is **easy**.

At the other end of the spectrum we have:

Definition 1.7. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **exponentially** if there are positive constants a and b such that

$$e^{ak} \ll f(k) \ll e^{bk}.$$

If the number of steps in an algorithm, when given as a function of the size of the inputs to the algorithm, grows exponentially, then we say that this algorithm is **slow**.
If the most-efficient known algorithm to solve a problem is slow, then we say that this problem is **hard**.

We will see that there is also an intermediate "speed" at which certain problems can be solved:

Definition 1.8. Let $f\colon \mathbb{N} \to \mathbb{N}$. We say that f grows **subexponentially** if for every positive constants a (no matter how big) and b (no matter how small) we have

$$k^a \ll f(k) \ll e^{bk}.$$

In other words, f is "larger" than any polynomial, but "smaller" than any exponential function.
If the most-efficient known algorithm to solve a problem has a number of steps that grows subexponentially, we usually still think of the problem as hard, but we must keep in mind that it is not exponentially hard.