Theory of computation

Péter Gács
Boston University

September 17, 2008

Abstract

These notes for a short lecture try to give an idea about the problems that a theoretical computer scientist is interested in. The intended audience is technically literate, typically scientists in other areas, for example biology, or sufficiently curious high-school students.

1 Computers

A computer, in the broadest sense, is an information-processing device. What can computers do with information? I am actually asking what can be done with information at all, since I do not believe that humans have some extra capabilities that computers don't. However, you do not have to agree with me on this issue to find what follows interesting.

1.1 What can computers do with information?

Let us agree on what are the main ingredients of a computer:

- Internal memory. Finite, say storing numbers in registers $R_1, R_2, \dots, R_{10}$.
- External memory (slower). Potentially infinite, say on a disk. Part of it is used for storing the program. The machine is always scanning some position called the *current position P*.
- Processor. Performs the program, a sequence of simple instructions, like:
  - Add to register R_1 the number found at position $P+R_2$ on the disk. Then, go to the next instruction.
  - If the number in register R_3 is 0 then go to instruction 12 of the program.

There are many variations on this theme, but if you have computers C_1 and C_2 then C_1 can always simulate C_2: on the same input, C_1 will give the same output as C_2. If C_1 has less internal memory than C_2 then it will just use more of its external memory. The simulating computation may be somewhat slower than the simulated one, but only to a reasonable extent. Since C_1 simulates C_2 step-for-step, it uses the same "algorithm" to solve the problem as C_2.

1.2 Algorithms

What theoretical computer scientists are actually interested in is not computers but algorithms. An algorithm is an abstract procedure to solve a problem, that can then be implemented on particular computers in slightly different ways.

So, mathematicians call the problems solvable by computers *algorithmically solvable*.

2 Unsolvable problems

Are there any questions that we can formulate for a computer and it still cannot solve?

2.1 The halting problem

We allow our computers unbounded memory and unbounded time to operate. Given a fixed computer C, consider the following task: decide (maybe, with the help of another computer, C') for each possible program p for C, whether when we start computer C on program p, it will ever finish the computation?

This would be extremely useful, since we could be alerted about a very buggy program (that would run forever), without actually running it (and just killing it after some timeout).

Unfortunately, there is a mathematical theorem saying that this is an algorithmically unsolvable problem.

As you see, when talking about algorithmic solvability of a problem, what a mathematician really has in mind is not one problem, but a whole *infinite family of problems*. Each possible value of the input to the algorithm defines a different member of the family. In the above example, each program p would be such an input to the algorithm solving the halting problem. An algorithmic solution must be a general procedure that works for every possible input.

Remark 2.1. The theory of computatability predates computers. The mere existence of algorithmically unsolvable problems follows already from some very general set-theoretical considerations (the set of possible programs is countable). ⌟

2.2 Solving integer equations

Are there algorithmically unsolvable problems about things other than computers? Yes. Consider an equation of the form

$$x^3 = 3y^6 - 2x^4 - 11.$$

We are interested in finding out whether this equation has any solution in integers. Is there any algorithm to solve *all such problems*? The answer is a famous theorem, saying no!

3 Measuring information content

I will give you a flavor of the kind of reasoning leading to such results. I will actually prove the algorithmic unsolvability of some problem.

3.1 Compressibility

Let us try to measure the amount of information in some text. For simplicity, assume that the text has been written using only two possible characters, 0 and 1, so it is some string

$$x = x_1 x_2 \dots x_n.$$

If $x = 0101\dots01$ then this string can be described by just saying: "take $n/2$ repetitions of 01". If n is large then saying this is simpler than writing out the sequence. We can say that the sequence can be "compressed", or "encoded" into a much shorter string.

This compression is not a formal mathematical notion until we agree how the shorter description can be "interpreted", or "decoded". To have some fixed and fairly general standard, let us agree on the following. We fix some computer T which we will use by giving it some 0-1 string p as input. If T ends its work by outputting some 0-1 string x then we write $T(p) = x$, and say that p is a *description* of x for the computer T. We define

$$K_T(x) = \min_{T(p)=x} |p|.$$

In words, $K_T(x)$ is the length of the shortest input for T which gives x as an output. It is the length of the shortest description of x on T, we will also call it the *description complexity* of x on T.

It is useful to think of the description p as a "program" for the machine T.

3.2 Invariance

Description complexity seems to depend very much on the machine T, but this is not so. It turns out that there is an *optimal machine U*, namely one for which the complexity is almost minimal: for every other machine T there is a constant c such that

$$K_U(x) < K_T(x) + c.$$

Optimal machines are not very special at all: all the machines you are familiar with are optimal. So, the description complexity of a string x is essentially an inherent (and interesting) property of x. It tells us how much information is "really" there in x, and how much is just fluff. From now on, I fix U and omit it from the notation:

$$K(x) = K_U(x).$$

3.3 Arbitrarily complex objects do exist

What can one do with such a definition?

It seems unwise to attempt actual mathematical argument in a popular lecture but my aim is to demonstrate the power of simple reasoning, so I will do some proofs.

Let us show that there are arbitrarily complex strings. For example, there are some strings x with $K(x) > 100$. Indeed, how many strings are there whose complexity is, say, 55? At most 2^{55}, since this is the number of possible descriptions of length 55. Therefore the total number of strings with descriptions of length $\leqslant 100$ is at most $1 + 2 + \cdots + 2^{100}$, a finite number. All other strings are more complex.

Note the peculiar (to some, perverse!) nature of this proof: I did not show you any string x with $K(x) > 100$. I just proved that there is one. You are a potential mathematician if you enjoy the dizziness coming from contemplating such *non-constructive* proofs.

3.4 Uncomputability

Description complexity has some beautiful properties, but I will not talk about them now. I just want to show you that however interesting this function is, it is not something you can use in practice: it is uncomputable.

The proof goes along the lines of an old paradox. There are some numbers that can be defined with a few words: say, "the first number that begins with 100 9's", etc. Clearly, there are only finitely many numbers definable with a sentence shorter than 100 characters. Therefore, there is a first number that cannot be defined by a sentence shorter than 100. But—I have just defined it!

This is just a paradox, good to enliven a coctail party, but it does not prove anything, since it uses an undefined notion of "define". But, when we specify what we mean by "define", (namely, here we say that p defines x if $U(p) = x$) then we can draw real mathematical profit from the paradox, as shown below.

We will prove that the function $K(x)$ is not computable by assuming that it is, and arriving at a contradiction. So, we assume that there is an algorithm that on input x, computes the number $K(x)$. Then there is also an algorithm Q that on input k, outputs the first string $x(k)$ such that $K(x) > k$. (We have seen that there are strings of arbitrarily large complexity.)

On our machine U, let q be the length of a program for the above algorithm Q. For some number k, we can write now some program $r(k)$ for U that outputs $x(k)$. How long must this program $r(k)$ be? We need q bits for the program of Q, then $\log_2 k$ bits for the number k, and finally some more bits (a constant number p) to tell the machine U what to do with this information: this is

$$p + q + \log_2 k.$$

Since p, q are constant, if k is sufficently large then this is less than k. We arrived at a contradiction: we output $x(k)$ with a program shorter than k, but $x(k)$ was defined as the first string whose complexity is larger than k.

4 Time complexity

There are many interesting algorithmically unsolvable problems. At the same time, the fact that a problem is algorithmically solvable does not guarantee that it will also be practically solvable. Among other things, if the algorithm runs too long then it is not practical.

4.1 Multiplication

How long does a given algorithm run? In some cases, this is easy to estimate. As an example, look at the problem of multiplying two long numbers:

$$54551977800004632211 7367 \cdot 9899101317848263092.$$

You learned a multiplication algorithm at school, and that could be applied here. First multiply the whole first number by 2, then by 9, then by 0 then by 3 etc., and add up the (shifted) results.

How long does this take? If the first number had m digits and the second number had n digits, then it takes approximately m operations to carry out the first multiplication, another m operations for the second multiplication and m more operations to carry out the addition. With each new digit of the second multiplier, we need about $2m$ new operations, so altogether we need about $2mn$ elementary operations for the whole.

Once mathematicians looked at the multiplication problem seriously, they asked the question: is this the *inherent* complexity of multiplication? Do we really need n^2 elementary operations in order to multiply two numbers of length n? Or, maybe there is some better way?

Remark 4.1. You may ask: who wants to multiply such enormous numbers? There are two answers to this question. The more important answer is that every simple example must be thoroughly examined in order to gain some understanding about the nature of complexity in general.

The second answer is that in fact, multiplication of large numbers is very important nowadays, since arithmetic of very large numbers is used in the best-known cryptographyc algorithms. ⌟

A non-mathematician may say: of course, you need n^2 steps, since of course, you need to multiply every digit of the first number with every digit of the second number! Mathematicians are just pedants wasting their time (and our money) by trying to also find a *proof* that this is indeed so. Well—it turns out that when this question was asked precisely, a surprising answer came: there is an algorithm to multiply large numbers much faster! How fast? For example, faster then $c \times n^{1.1}$ steps, where c is some constant. (That method is practical only for large numbers, since the constant c is large.) It may amuse some of you to learn that the currently best known multiplication algorithm uses Fourier transforms.

4.2 Polynomial versus exponential

Taking in a bigger picture, there is a useful distinction between solutions that can in general be regarded as using a reasonable amount of resources and those that don't.

Factorization

We learned at school, how to multiply or divide two numbers. If the numbers have n digits then both operations take about n^2 operations (maybe $2n^2$ or $3n^2$ but we are after bigger fish than saving a coefficient of 2). We have also learned about another algorithm: how to *factorize* a number. Now, forget about finding all factors, just try to find at least one factor. Suppose the number is x. What we learned is: let us try to divide x by 2, 3, $\dots$, until we find a number that divides x. It is easy to see that it is sufficient to search up to $\sqrt{x}$, but this is still $\sqrt{x}$ divisions. Now, in order to compare this to the number of operations needed for multiplication and division, let us express it in terms of the number of digits of x. If x has n digits then

$$\sqrt{x} \approx 10^{n/2} > 3^n.$$

So, the cost of factorization (by the known algorithm) differs from the cost of multiplication/division dramatically. The cost of multiplication is n^2, which grows only *polynomially* in n, the cost of factorization is 3^n, growing *exponentially*. To multiply numbers that have $(n+1)$ digits instead of n, we have to do $(2n+1)$ additional operations. To factorize numbers that have $(n+1)$ digits instead of n, we have to perform 3 times more operations! Today's computers will multiply numbers of length 1000 cheerfully. But to factorize such numbers with the above algorithm, it would not help even if we turned the whole universe into the fastest imaginable computer and ran it till it collapses.

As you see, we measure the cost of an algorithm as a function of the *length of the input*. The gulf between algorithms that take polynomial time and those that take exponential time is generally enormous, and is the dividing line that computer scientists are most concerned with. In lots and lots of important cases, the question whether we found a polynomial algorithm for solving a problem or only an exponential one (or even worse!) is a crucial test of whether we understood something deeper about the problem or have only a so-called *brute-force* method. The factorizing algorithm I have shown is clearly brute-force: we just (essentially) "try every possibility".

Mathematicians have worked on finding better methods for factorization for centuries; there are definitely better methods than the one I have shown, and still: all known methods take exponential time. However, there is no guarantee that a polynomial algorithm will not be found in the future.

Prime tests, randomization

For example, there is a related problem: finding out whether a number is prime. If we cannot find any factors of a number x then x is a prime. From this, it seems that if we cannot factorize faster then we cannot find out faster also whether a number is prime. This is not so: there is an ingenious algorithm that finds out whether a given number of n digits is prime, in time that takes a polynomial number (say, n^3) of steps only. This is because prime numbers have some special mathematical properties that it is possible to exploit, and to do something much better than brute force.

A peculiar feature of the prime-test algorithm is that it is *randomized*: it uses *random numbers*, and hence the answer it gives is not absolutely certain. Randomized algorithms are a fascinating topic, but we cannot get to them in this talk. Enough to say that in practice, it is satisfactory if some algorithm works with very high probability.

5 Proving that certain tasks are difficult

Is factoring really hard? Or maybe there is a fast solution algorithm, it just has not been found yet? Today's mathematics is not able to answer such questions yet.

5.1 A provably hard problem

It is hard to come up with any really interesting problems for which we have a proof that they are hard. Here is an example problem that provably takes *exponential time* to solve. Take the Japanese game "Go", played on an $n \times n$ board instead of a 19×19 one. Put up some board configuration, and ask the question: in this configuration, which side will win (if he plays in the best possible way)?

5.2 Poor man's substitute for hardness proof: NP-completeness

There is a large group of problems including factorization, called NP problems, about which we can also say more. In this group of problems there are some that are known to be the *hardest*: these are called NP-*complete*. We do not know how hard these are but it is generally assumed that they take exponential time.

Example 5.1. Factorization is (probably) not an NP-complete problem. Here is an example of such a problem, called the *knapsack problem*. Suppose we have some items with volumes $v_1, v_2, \dots, v_n$ (these are positive integers) and a knapsack of volume w. Can we pick some of these items, leaving at home the rest, and fill with them the knapsack *exactly*? If somebody gave a polynomial algorithm for this innocent-looking problem, then in one strike, his algorithm would solve a lot of the most interesting algorithmic problems in the world (including factorization), and would change completely the general view about what is easy and what is hard. There is no proof that this will never happen. ⌟

6 Profiting from complexity: cryptography

Nowadays a lot of sensitive communication occurs over the internet. There is a need to encrypt messages, to protect them from a potentially malicious third party. Cryptography has been an art, a bag of tricks, for millenia, but until recently, its level of sophistication has been quite low.[1] Complexity theory changed this, it turned cryptography into a highly evolved science.

6.1 Encrypted communication

What happens in the simplest cryptographic setup? Alice wants to send a (long) text x to Bob. They share a (short) secret key k. Alice applies some *encryption transformation* to the text x that depends on the key k, and turns it into a text

$$y = E(x, k).$$

Bob receives the encoded message y. He also knows the key k and this permits him to recover x, using a *decryption transformation*

$$x = D(y, k).$$

Example 6.1. k might be the number telling us by how many steps to rotate the alphabet. The encryption $E(x, k)$ rotates each letter of text x by k steps forward in the alphabet (passing from z to a again). So, if $k = 2$ and $x =$ ABLAK, then $y =$ CDNCM. The decryption $D(y, k)$ rotates each letter of y backward by k. ⌟

There is a third player, Eve, the eavesdropper, who intercepts the message y and wants to find out x. The system is good only if this is hard to do for Eve. In the example, if Eve is a little persistent, she will break the code. There are only 26 possible values of k, she can try all. Let us attempt to improve on the system: let the key be a *sequence of numbers* of length 100:

$$k = (k_0, \dots, k_{99}).$$

For example, $k = (12, 25, 8, 23, \dots, 5)$. If x_i is the ith letter of x and $i = 100q+r$ where $r < 100$ then we rotate x_i by the number k_r. So, for example, we rotate x_0 and x_{100} by 12, x_1 and x_{101} by 25, etc. Now our key k is so large that Eve cannot apply the brute-force method of trying all possibilities. However, she can do more sophisticated things, like looking at the statistics of the sequence $y_0, y_{100}, y_{200}, \dots,$ to find k_0. We can try to confuse things even more, and hope that is enough but how can we be sure it is?

6.2 The nature of cryptographic guarantees

It would be great to have some *theoretical guarantee* that without knowing the key k, Eve *must* spend an enormous number of computation steps to recover x from $y = E(x, k)$. Nobody has such guarantees nowadays, and we will have a really firm foundation for cryptography only when such guarantees will be available. But there are *conditional* guarantees. Some of the currently popular systems of cryptography are designed such that they are safe if the problem of factorization is difficult. Thus, it is proven that if Eve could break the code in polynomial time then she could factorize large numbers in polynomial time: she would surpass some of the best minds of the last two millenia.

Remark 6.2. When a news story appears saying that the "RSA cryptosystem was broken", this generally means that some particular large numbers were factored using exhaustive search and many computers. ⌟

[1] However, the role of cryptography in World War II is a fascinating story.

6.3 New cryptographic feats

Encryption of communication is only the simplest task that can be solved by cryptographic means. More generally, you can imagine any interaction among several parties involving information exchange, in which the interests of the different parties do not coincide. Organizing the interaction in such a way that the desired result comes out but nobody learns anything that she is not supposed to, is done with the help of *cryptographic protocols*.

Example 6.3. The American Airlines company wants to solve a huge scheduling problem. Several mathematical software companies compete for this business. AA will decide whom to give the contract on the basis of a smaller sample problem that it posts. The Genius software company wants to prove AA that it can solve the sample problem, but it does not want to actually show the solution. Moreover: it does not want AA to learn *anything* beyond the fact that Genius has a solution: it suspects that somebody at AA might sell such information to Genius's competitors.

There is a cryptographic protocol, called Zero-Knowledge Proof, allowing Genius company to actually do this: AA will be convinced beyond all reasonable doubt that Genius has the solution, but AA will not learn absolutely anything else. ⌟