Lecture 1 - Introduction to Complexity.

Boaz Barak

February 7, 2006

What is a computational problem? A computational problem is basically a mapping/relation from inputs to desired outputs. Examples:

- Addition: inputs x, y output $x + y$
- Multiplication: inputs x, y output $x \cdot y$
- Table arrangement: **input:** a list of guests along with a list of "forbidden pairs" who should not be seated next to each other **output:** arrangement of seats along a circular table without any forbidden neighbors, or `fail` if this is impossible.
- Hamiltonian cycle: **input:** a graph $G(V, E)$ **output:** an n-cycle $E' \subseteq E$ if there exists such a cycle, or `fail` if there is none.

Computational problems arise all the time, and not just in the context of silicon-based computers. Our brain solves such problems any time we move, grab something or look at our surrounding. Nature solved and solves such problems many times in the course of evolution. Essentially any scientific discovery or invention involved also solving computational problems (e.g., find the equation that best fits the data).

What is an algorithm? An algorithm is a sequence of basic instructions to solve a computational problem. At the moment, we'll use the "I know it when I see it" definition for "basic instructions" - you can think of the most simple operation a person or computer can perform (e.g., writing down a digit, adding two digits, etc..).

Examples:

1. Grade-school algorithm for computing addition.
2. Multiplication by repeated addition: compute $x \cdot y$ by adding x to itself y times.
3. Grade-school algorithm for multiplication.

We see that there can be several algorithms for the same problem. However, we feel that the grade-school algorithm is somehow "better" than repeated addition. This is because it is more efficient: assume both x and y have n digits and hence are between 10^{n-1} and 10^n. Repeated addition uses y additions of x: at least 10^{n-1} additions (each of which takes n basic operations). The grade-school algorithm uses only n additions of n-digit numbers (and multiplying by power of ten, which is trivial to perform). Again, since each addition takes about n basic operation, the total number of operations is roughly n^2. Surprisingly, there are several better algorithms, taking as few as $n \log n$ operations (although whether it's possible to solve this in $O(n)$ operations is a major open problem).

Note that once we think of even moderately large numbers, the difference between the inefficient repeated addition algorithm and the grade-school algorithm becomes the difference between an *immensely useful* and *almost useless* algorithm.

The best known algorithm for the Hamiltonian cycle/table arrangement problem with n vertices/guests takes roughly $n!$ operations, which means that even if you had a light-speed supercomputer, you'd need to plan dinners with more than 50 guests or so a few billion years in advance.

Goal of computational complexity The goal of computational complexity is to understand the *inherent difficulty* of computational problems: what is the efficiency of the *best* algorithm to solve each problem.

Thus, given a computational problem, say the problem of computing $f(x)$ from x we want to do two things:

1. Find the best algorithm to compute $f(x)$.
2. Prove that there does not exist a more efficient algorithm.

As demonstrated by age-old problems that found new algorithms in the last few years (e.g., primality testing), finding efficient algorithms can require considerable ingenuity and originality. However, at least it's pretty clear what the task to solve is. This is not the case for the second task: how do you prove that something is impossible? how can you rule out the existence of a yet undiscovered better algorithm?

Impossibility results (lower bounds) The idea of trying to *prove* that some task is impossible has a long history in mathematics. Such questions often proved to be among the hardest and most enduring in mathematics, which is perhaps not surprising, since proving that something can *not* be done does seem intuitively difficult. However, even though at a first look an impossibility result does not seem to be very useful, such questions and their solutions also turned out to be incredibly insightful and useful, sometimes inspiring whole fields of study. Some examples for such questions are:

Irrationality of $\sqrt{2}$ the proof, although simple, that no matter what, the square root of 2 (or in other words, the length of the diagonal line in a unit square) can not be expressed as a ratio of two integers, came as a shock to the ancient greeks. The story (myth?) is that the student of Pythagoras that discovered this was sentenced to death by him.

Formulas solving 5^{th} degree equations, geometric problems of antiquity We learned in school the formula for solving a quadratic equations. It turns out there are more complicated formulas for solving equations of degree three and four, but nobody knew a formula for degree five. At some point mathematicians began to suspect that there is no such formula, and in fact such a result was announced by Ruffini in 1799 (although his proof contained a gap). Abel proved this rigorously in 1824, and Galois further studied this in the 1820's and 30's. These studies eventually led to the study of *group theory* — an immensely useful branch of mathematics with applications to many other sciences such as physics, chemistry and computer science.

Along the way, this study provided other famous impossibility results. The ancient greeks were immensely interested in geometric constructions using a compass and an unmarked ruler. They had many results and successes, but the solutions to some construction problems evaded them. These are the famous "geometric problems of antiquity", which included **(1)** the doubling of the cube (i.e., construct a cube whose volume is double that of a given one), **(2)** angle trisection (i.e., trisect an arbitrary angle) and **(3)** squaring a circle (constructing a square whose area equals that of a given circle). After more than 2000 years of studies and false proof, all these problems were eventually proven impossible to construct in the 19^{th} century, and the result for the first two problems (Wantzel 1837) follows from Galois theory . The result for the third one (squaring the circle) follows from another famous impossibility result (Lindemman 1882): that the number π cannot be expressed as the root of any polynomial.

Independence of the Euclidean 5^{th} postulate : the *Elements* by Euclid is the best known textbook of all time, and is claimed to be second only to the bible in the number of editions printed. In this book Euclid develops geometry starting from 5 postulates or axioms. The first four essentially state that every two points define exactly one infinitely long line passing through both of them, and one circle centered on one and touching the other, and define a right angle as one whose sum is the "flat" angle. The fifth postulate says that for each line and point outside the line, there's exactly one parallel line passing through this point.

Euclid was not terribly happy about the fifth postulate, but could not prove it from the previous four. He attempted to delay using it as much as possible, and did not use it in the proofs of his first 28 theorems. Ever since his time, people have been trying to prove the fifth postulate from the previous four, and it had been called "the one sentence in the history of science that has given rise to more publication than any other.". In 1767, this question was called "the scandal of elementary geometry" while Gauss, in 1813 called it "a shameful part of mathematics".

There were many false proofs by many great mathematicians throughout the history of this question. Many of them assumed the fifth postulate is false, and tried to derive a contradiction, although the contradiction always turned out to be to some implicit assumption of the prover, and not to the other four postulates.

The important mental breakthrough happened in the 1820's by Bolyai and Lobachevsky (and also some unpublished works of Gauss) — instead of trying to derive a contradiction from the negation of the fifth postulate, they explored the possibility that it actually *can not* be proved from the other four and thus derived the wonderful new *non-euclidian geometry*. This was finally formally proven by Beltrami in 1868. In his general theory of relativity, Einstein showed that non-euclidian geometry is actually the right way to describe our physical world.

Hilbert's Project and Godel's theorem. David Hilbert was one of the most prominent mathematicians of the late 19^{th}/early 20^{th} century with contributions to algebra, geometry, number theory, analysis and logic. In his last 30 years as a researcher, he dedicated himself to the so called "Hilbert Project" of placing mathematics on solid foundations. He wanted to place all the theories of mathematics on the basis of a few axioms which can be proven to be *consistent* (that is, you can not use these axioms to prove both a statement and its negation). This was already done (by him) for geometry, and he wanted to extend this to number theory and analysis. In a sense, he wanted to show that at least in principle, mathematics can be mechanized: there can be a mechanical procedure (we would say computer program) that can check the truth of any mathematical statement, by trying all possible ways to derive it from the axioms.

This project was crushed by a surprising impossibility result of Kurt Goedel (1931, Goedel moved to Princeton in 1940 and stayed here until his death in 1978): he proved that any consistent axiomatic system describing number theory will be *incomplete* in that there will be a statement such that both the statement and its negation are not provable in the system. He also proved that no axiomatic system can prove its own consistency.

Questions of complexity theory Some of the questions we'll deal with in complexity theory are the following:

1. How fast can we multiply two numbers?
2. Is the brute force algorithm essentially the fastest one for the Hamiltonian cycle problem? Is there a very efficient algorithm (e.g., number of steps linear in the size of the graph)?
3. Hamiltonian cycle problem is a *search problem* in the sense that once you find the solution (e.g., a valid seating arrangement) it is easy to verify it. Many such computational problems arise in various contexts. Is there one of these problems where the brute force algorithm is the fastest one? Or perhaps all of these problems admit a very efficient algorithm?
4. What about algorithms for *approximate solutions*? For example, is it possible to have an efficient algorithm such that whenever there exists a conflict free seating arrangement, the algorithm might not find this arrangement, but will find an arrangement with very few (e.g., at most a constant number) of conflicts?
5. What about the power of various computational resources. For example, if we allow an algorithm to toss coins and make some mistakes (i.e., only compute the right value with probability 2/3), can we get significantly faster algorithms for these problems?

Alas, at this point, complexity theory offers more questions than answers. However, we will manage to prove many connections between these questions, for example showing:

1. Questions 2 and 3 are related: if there is an efficient algorithm for the Hamiltonian cycle problem, then in fact there is an efficient algorithm to all the search problems whose solution can be *verified* efficiently. Thus, if there is even one such problem that is hard, then the Hamiltonian cycle problem is also hard.
2. Question 4 is related to these also: if the Hamiltonian cycle problem can be approximated (i.e., in the case a perfect arrangement is possible then there's an algorithm to find an arrangement with only a constant number of conflicts) then it can actually be solved completely.
3. Question 5 is also related to these questions: if the Hamiltonian cycle problem can not be solved significantly faster than brute force search then any algorithm that tosses coins and may make mistakes can be converted to an algorithm that does not use any randomness and makes no mistakes, but has similar efficiency. Furthermore, if the Hamiltonian cycle problem is easy and has an efficient algorithm then this is still the case. In fact, the only way that randomness adds significant power to algorithms is if the Hamiltonian cycle problem had "intermediate" complexity — faster than brute force, but not truly efficient.

We'll see these connections, as well as many other results, during the course.

The Computational Model While the "I know it when I see it" definition of an algorithm was used for thousands of years, it is not sufficient for advanced study of algorithms, and certainly not for obtaining and even phrasing negative results. Thus, we need a mathematical formalization of what is an efficient algorithm, even if we can often let ourselves forget these details and go back to the "I know it when I see it" definition. We'll now (briefly) describe the computational model we use. For more details see chapter 1 of the textbook.

Turing machines The Turing machine, invented by Alan Turing (a former Princeton grad student) is a mathematical model of computation. The machine has a memory tape which can be thought of as an infinite line of cells (although the machine will only use a finite part of it during finitely many computing steps), where each cell contains a symbol from a finite alphabet, say $\{0,1\}$ (we also add a special blank symbol). The machine has a read/write head, which at each step can read/write one cell of the tape and move one step left or right, and a finite number of possible states. A machine is specified by the (finite) function it uses to determine what to do when it is in a particular state and the head reads a particular symbol. If $\pi : \{0,1\}^n \to \{0,1\}^m$ is a function and M is a Turing machine, we say that M *computes* $\pi(\cdot)$ if whenever M's tape is initialized to x and M's head is placed on the first bit of x with M initialized to its initial state q_{start}, there is some number $T \in \mathbb{N}$ such that within T steps M halts with the tape now containing $\pi(x)$.

Let $\pi : \{0,1\}^* \to \{0,1\}^*$ and $T : \mathbb{N} \to \mathbb{N}$ be two functions. We say that $\pi \in \mathbf{DTIME}(T)$ if there exists a Turing machine M and a constant C such that for every $x \in \{0,1\}^*$, $M(x)$ outputs $\pi(x)$ within $C \cdot T(|x|)$ steps.

We note that one can define Turing machines with several tapes, with a special purpose output tape, with a tape that's infinite only in one direction and many other variants. All these variants do not change the class $\mathbf{DTIME}(T)$. (The class $\mathbf{P}$ which we'll define shortly is even more robust to the choice of a computational model, and is for example remains the same if we use Turing machine with a two-dimensional or three-dimensional tape, or even if we a machine with a memory array (so called RAM machine).)

Universal Turing machine One of the properties observed by Turing that make the Turing machine interesting, is that we can have a single machine that is a general purpose computer. That is, we have a machine U that can get as input a "program" (i.e., a description of a Turing machine M) and a string x, and outputs $M(x)$. Furthermore, if M took T steps to halt on input x, $U(M,x)$ will output $M(X)$ in $O(T \log T)$ steps. Thus, the machine U is sufficient to simulate all possible computations with quite a small overhead.

The class P. We define the class $\mathbf{P}$ to be the union of $\mathbf{DTIME}(n^d)$ for every $d > 0$. That is, $\pi \in \mathbf{P}$ if there are constants c, d and a Turing machine M such that $M(x)$ outputs $\pi(x)$ within $c|x|^d$ steps. For the moment, we are going to use $\mathbf{P}$ as our candidate for the class of "easy" or "efficiently solvable" problems. This is not a completely uncontroversial choice. Some arguments against treating $\mathbf{P}$ as synonymous with efficient computation include:

P is too restrictive At least syntactically, it seems that $\mathbf{P}$ does not allow for computers that toss coins. There are also suggestions of using Quantum effects to speed up computation, which again, at least syntactically, $\mathbf{P}$ does not allow (however at the moment it is not yet certain that these suggestions are physically realizable). Since the problems we encounter in the world are all finite, there's also the question of whether a finite, non-uniform model of computation is not the right model to study (we'll see a bit more about this in the next class). There's also the issue that **P** is defined with respect to *worst case* performance, rather than *average case*. Indeed, we have other classes such as **BPP**, **BQP**, $\mathbf{P}_{/\mathbf{poly}}$ and **AvgP** (all of which contain **P**, but except for $\mathbf{P}_{/\mathbf{poly}}$, it is not known whether or not they are equal to **P**) that aim to capture such concerns. When we finally prove that Hamiltonian cycle is not in **P** the obvious next step would be to try to prove it's also not in all the other classes.

P is too permissive Another critique is that **P** contains problems that are patently hard such as problems solvable in time n^{100} but not in time n^{99}. However, it still seems that knowing that a problem is in **P** is very meaningful: for example this means that when we add a few bits to the input length, the time complexity of the problem does not double. Furthermore, most of the known algorithms for natural problems seem to be either super-polynomial (typically exponential or subexponential) or small degree polynomials. In many cases, showing that a problem in **P** seems to be the crucial first step in proving a, say, n^3-time algorithm for it. Furthermore, the definition of **P** is convenient since it allows for composition (since a polynomial applied to a polynomial is still a polynomial), and robustness of the model (i.e., **P** is the same whether defined for Turing machines, RAM machines, or parallel computers).

While we should always keep in mind these critiques, **P** will serve us very well, at least as a crude approximation for efficient computation.

Note: In most sources (including the textbook), the class **P** is defined with respect to computing functions with *binary* (one bit) output. Such functions can be equated with subsets of $\{0,1\}^*$ and in this context are often called *languages* or *decision problems*. This does not make much difference in most contexts, and we'll sometimes implicitly restrict ourselves to such functions as well. To separate general functions from decision problems, some texts call the class we defined above **FP**. (I decided to avoid this definition to minimize the number of distinct complexity classes and notations in this course.)

Search problems Recall that a machine M computes a function $\pi : \{0,1\}^* \to \{0,1\}^*$ if whenever the input tape is initialized with x, M will output $\pi(x)$ within a finite number of steps.

However, this formalism does not capture search problems such as the Hamiltonian cycle problem. This is because the natural way to define this problem as a function would be that for every graph G, we'll define $\pi(G)$ to be the Hamiltonian cycle in G. However, many graphs can have zero or more than one cycles, and hence it's not clear how to define this function.

We thus generalize the definition of computation to the case of *relations* that may not necessarily be functions. That is, we think of π as a relation which is a subset of $\{0,1\}^* \times \{0,1\}^*$. For every x we denote by $\pi(x)$ the set $\{y \mid \langle x,y\rangle \in \pi\}$. Note that $\pi(x)$ may be empty or have more than one element. (If π is a function then $\pi(x)$ always contains a single element.) We call $\pi(x)$ the set of *witnesses* for x. We say that a Turing machine M *solves* π in $T(n)$ steps if for every x with $\pi(x) \neq \emptyset$, $M(x)$ outputs $y \in \pi(x)$ within $T(|x|)$ steps, whereas if $\pi(x) = \emptyset$ the machine outputs a special "fail" output, which we denote by $\perp$.

For example, for every graph G we can define the relation $HAM(G)$ to be the set of Hamiltonian cycles in G. Thus a machine that solves HAM solves the Hamiltonian cycle problem.

We define the class **P** to include all the relations that are solvable in polynomial time.[1] Recall

[1] As mentioned below, there is actually an additional technical condition we'll add later on.

that one of the questions we asked last lecture is whether the Hamiltonian cycle problem has an efficient algorithm. We can formalize this question mathematically as "Is $HAM \in \mathbf{P}$?".

The class NP As we said, the Hamiltonian cycle problem has the property that a solution to the problem can be easily verified. In general, we say that a relation π is *polynomial-time verifiable* there is an algorithm (i.e., Turing machine) that given *both* x and y, outputs within $\text{poly}(|x|)$ steps 1 iff $\langle x, y\rangle \in \pi$. Note that for example taking the relation HAM above, we know that it is polynomial-time verifiable, but we do not know whether it is polynomial-time solvable.

We define **NP** be the set of all polynomial-time verifiable relations. We now ask the question, of whether all polynomial-time verifiable relations are in fact polynomial-time solvable. That is, is "$\mathbf{NP} \subseteq \mathbf{P}$".

Note: The way we defined **P**, there may very well be relations in **P** that are not in **NP**. This is because there may be a contrived relation where given x it's always easy to find one member of $\pi(x)$ but not necessarily to verify whether $\langle x, y\rangle \in \pi$ for all y. The way to get rid of this technicality is to say that a relation is in **P** if it is both polynomial-time solveable and polynomial-time verifiable. Note that for functions this does not make any difference. Now the question becomes "Is $\mathbf{NP} = \mathbf{P}$?".

Decision vs. Search In most sources (including the text), both **P** and **NP** are defined with respect to *decision problems* and not to *search problems*. We'll now show why these two questions are equivalent.

Let π be a polynomial-time verifiable relation. We define the decision problem $L(\pi)$ to be the set of x such that $\pi(x) \neq \emptyset$. We define $\mathbf{NP'}$ to be the set of all decision problems corresponding to polynomial-time verifiable relations. We define $\mathbf{P'}$ to be the set of decision problems in **P**.

Theorem 1. $\mathbf{P} = \mathbf{NP}$ *if and only if* $\mathbf{P'} = \mathbf{NP'}$.

Proof. If we can solve the search problem in polynomial-time, then we can surely solve the decision problem. Thus, if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{P'} = \mathbf{NP'}$. The other direction is less trivial. Suppose that we have, say, a polynomial-time algorithm to solve the decision problem $L(HAM)$, can we use it to solve the search problem HAM itself?

We'll now show this for HAM. The general result will follow from that because HAM is **NP**-complete.

Algorithm for HAM given algorithm for $L(HAM)$:

- **Input:** graph $G = (V, E)$.
- **Resource:** Algorithm $DHAM$ to solve $L(HAM)$.
- For every edge $e \in E$, consider the graph $G' = (V, E\setminus\{e\})$. If $DHAM(G') = 1$, then let $G \leftarrow G'$ and repeat until for every $e \in E$, $DHAM(V, E \setminus \{e\}) = 0$.
- Output E.

Analysis, TBC. □

From now on, we will mostly use the *decision-problem* definition of **NP** and **P**, but from time to time move implicitly between the search-problem and the decision-problem definitions.