Computational Complexity

This is to give some background to the question, "How fast can Linear Programming problems be solved?", and its relation to the central question of computational complexity theory, "Is $P = NP$?". This will be a somewhat informal discussion, leaving out some of the messy details.

The theory mainly deals with *decision problems*. A decision problem is a question with a yes-or-no answer, depending on some variables. An example of a decision problem is:

Given positive integers x and y, is x divisible by y?

This decision problem can be solved rather efficiently by a algorithm: long division. We're interested in how many steps such an algorithm might take. Of course the number of steps needed is likely to depend on the inputs. In particular, larger and more complicated inputs tend to require more steps. The size of the input might be measured by the number of bytes used to write it. An algorithm is said to be "polynomial-time" if there are constants c and d such that on every acceptable input of size n the algorithm stops after at most cn^d steps. Long division is an example of a polynomial-time algorithm (with $d = 2$). From the theoretical point of view, a polynomial-time algorithm is considered "fast" and anything that's not polynomial-time is "slow". P is the class of decision problems for which polynomial-time algorithms exist. So basically problems in P are considered tractable and those not in P are intractable. Depending on the values of the constants, a non-polynomial-time algorithm might be competitive with a polynomial-time algorithm for some small n's, but if you try larger and larger values of n the polynomial-time algorithm will eventually win.

For example, contrast a polynomial-time algorithm A that takes up to $c_1 n^3$ steps with a non-polynomial-time algorithm B that takes $c_2 2^n$ steps on a problem of size n. Suppose you've been solving problems of a certain size n on your computer, and it takes up to an hour. Now you trade in the computer for a newer model that is twice as fast, so it can do twice as many steps in an hour. How does this affect the size of problems you can handle in an hour? For the polynomial-time algorithm A, you could increase n by a substantial amount, about 25% ($1.25^3 < 2$). But for the non-polynomial-time algorithm B, you could only increase n by one byte.

From this theoretical point of view the constants c and d are not important. In practice they are: we wouldn't get very far with an algorithm that took $10^{1000} n^{1000}$ steps. But in practice, most problems that are known

to be in the class P turn out to have algorithms with small enough values of c and d that quite substantial problems can be solved in a reasonable length of time. That's not the case for many problems that are not in P.

It's also important to note that we're dealing with "worst-case" scenarios. You might have an algorithm that works very quickly for nearly all of the possible inputs of size n, but a very small fraction of those inputs take a very long time. Such an algorithm could be very useful in practice, despite not being polynomial-time. In fact, the Simplex Method falls in this category. The Klee-Minty examples show that the Simplex Method is not a polynomial-time algorithm, but usually it seems to work quite well.

The other important class of problems is NP. These are decision problems where a "yes" answer can always be verified by a polynomial-time algorithm, given some extra information. For example, consider the problem

Given positive integer x, is x a composite number (i.e. is it divisible by some positive integer other than x and 1)?

For any positive integer y with $1 < y < x$, we have a polynomial-time algorithm (long division) to check whether x is divisible by y. If x is composite, then given the correct y we can quickly verify the fact that x is composite. So this problem is in NP. On the other hand, finding the factor y seems to be a hard problem: no polynomial-time algorithm for that is known. Notice that we don't require a verification for a "no" answer.

Every problem in P is in NP: you just throw away any extra information, and run the polynomial-time algorithm. The big question is, is $P = NP$, i.e. do the problems in NP all have polynomial-time algorithms? The question has been open since about 1971 when it was formulated by Stephen Cook of University of Toronto. Most (but not all) experts think the answer is no, but we don't have a proof. If you can answer this question (with a proof), you'll become extremely famous, in addition to collecting a $1 million prize.

There are certain problems that are, in a sense, the hardest problems in NP. These are called NP-complete. A problem A is NP-complete if every problem B in NP can be transformed into an example of problem A, in such a way that if you had a polynomial-time algorithm for problem A you could use it to solve problem B in polynomial time. So if even one NP-complete problem is in P, it would mean that $P = NP$. Thousands of problems have been shown to be NP-complete since 1971. One example is the *Subset Sum Problem*:

Given a finite set $S = \{s_1, \ldots, s_n\}$ of integers, is there a nonempty subset of S whose sum is 0?

If you had a polynomial-time algorithm for this problem, you could use it to solve every NP problem in polynomial time.

Why is this important? For one thing, the cryptographic methods which are used for most communications security these days are based on the idea that certain problems are difficult to solve. If you had an efficient algorithm for an NP-complete problem, you could break most of the world's codes.

Now Linear Programming is an optimization problem rather than a decision problem, but the two are closely related. For each optimization problem

Maximize $f(x_1, \ldots, x_n)$ subject to a set C of constraints
there is a related decision problem

Given y, does there exist $(x_1, ..., x_n)$ such that $f(x_1, \ldots, x_n) >= y$ and the constraints C are satisfied?

So if I say that an optimization problem is in P or NP or NP-complete, what it means is that the decision problem is in that class.

Now Linear Programming is in NP. The extra information you could use to verify a "yes" answer is a set of basic variables for an optimal tableau (or a tableau that shows the problem is unbounded). This was already known when the $P = NP$ question was formulated.

On the other hand, Integer Linear Programming, where the variable values are required to be integers, is NP-complete. For example, the Subset Sum Problem can be formulated as an Integer Linear Programming problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} s_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} s_j x_j \leq 0 \\ \text{all} \quad & x_j \in \{0, 1\} \end{aligned}$$

So if there was a polynomial-time algorithm for Integer Linear Programming, you could solve Subset Sum in polynomial time, and you'd have $P = NP$. But Integer Linear Programming is not Linear Programming.

This is how matters stood for some years: Linear Programming was known to be in NP, but was not known to be in P, and on the other hand it was not known to be NP-complete. The Simplex Method was not a polynomial-time algorithm, but nothing better was known.

Then in 1979 came a breakthrough: in Moscow, Leonid Khachiyan discovered a polynomial-time algorithm for Linear Programming, called the Ellipsoid Algorithm. It created quite a stir at the time, helped to some degree by newspaper accounts that, as often happens, missed the point completely: they seemed to say that he had solved the Traveling Salesman Problem, which is NP-complete. When I read that, I immediately knew that it couldn't be right. This was the height of the Cold War, and the government of the Soviet Union would never have allowed a polynomial-time algorithm for NP-complete problems to be published: they'd keep it secret and use it to break all the West's codes. Of course it wasn't that at all, it was just for Linear Programming, which nobody had ever claimed was NP-complete.

It turned out that, although it made Linear Programming tractable in principle, in practice Khachiyan's algorithm was not useful. It was much worse than the Simplex Method on typical problems. But this did leave open the possibility that a more practical polynomial-time algorithm might be found. And that's what happened in 1984, when Narendra Karmarkar at Bell Labs found a new polynomial-time algorithm, called Karmarkar's Algorithm. This one was good both in theory and in practice, and could compete with the Simplex Method on typical problems of moderate size (100 variables or so). Since 1984 there has been a lot of work developing "interior point methods" for linear programming and other convex optimization problems.