

Lecture 30: Time Complexity and P versus NP

We describe complexity classes of algorithms and computational problems and explain the question of whether $P = NP$.

Classes of algorithms

We use the notation $T_A(n)$ to denote the maximum number of steps used by algorithm A on inputs of size n . We'll assume there is some well defined model (eg, an abstract random access machine) used to measure the number of steps. Depending on how fast $T_A(n)$ grows as a function of n , we may describe the algorithm A as constant time, or logarithmic time, or linear time, or quadratic time, and so on.

A is constant time if $T_A(n) = O(1)$. This doesn't mean that A always runs in the same number of steps, but that it never takes more than some fixed constant number of steps. A is logarithmic time if $T_A(n) = O(\log n)$. An example of this is binary search of a sorted table. Recall that the O notation makes the base of the logarithm irrelevant. A is linear time if $T_A(n) = O(n)$, quadratic time if $T_A(n) = O(n^2)$, cubic time if $T_A(n) = O(n^3)$. Efficient sorting algorithms run in time $O(n \log n)$, a class without a special name.

An algorithm A is polynomial time if there exists some constant k such that $T_A(n) = O(n^k)$. Note that this includes linear, quadratic, cubic, etc. algorithms. There is some disagreement about the best definition of exponential time, but one definition is that A is exponential time if there exists some constant $c > 1$ such that $T_A(n) = O(c^n)$.

To revisit primality testing, what is the correct classification of the complexity of the "naive" algorithm of testing every possible divisor up to the square root of the number? At first, it might seem that the correct complexity is proportional to the square root of n , that is, sublinear. However, this is a confusion induced by the fact that we have used n in two different ways – as the input to the primality testing problem, and as a measure of the size of the the input.

So, let us rename the input to the primality testing problem to be i , and keep n as the measure of the size of the input. Thus, n is the number of decimal digits in i . The worst case running time of the naive algorithm occurs when i is prime, and \sqrt{i} trial divisions are performed. The trial divisions will dominate the other operations in the algorithm, so the running time will be $O(\sqrt{i})$. What is this in terms of n ? The largest number with n digits is $10^n - 1$, so i is approximately 10^n , and the square root of 10^n is $(\sqrt{10})^n$, which is greater than $(3.162)^n$. Thus, the running time of the naive primality test is exponential. It is important to keep in mind the distinction between the value of input number and its "size" (or number of digits) when discussing algorithms that operate on large numbers.

Classes of problems and P

In addition to algorithms, we also want to classify problems, for example, sorting, parsing context free languages, testing primality, or factoring. For this, we say that a problem is in polynomial time if there exists a polynomial time algorithm that solves the problem. Similarly for logarithmic time, linear time, quadratic time, and so on. That is, we measure the time complexity of a problem by the "fastest" algorithm for the problem. (Technically speaking, some problems don't have a "fastest" algorithm, but the intuition is helpful anyway.)

Recall that a decision problem is one with a yes or no answer. The definition of the class P is the set of decision problems that have polynomial time algorithms. For example, there is a straightforward linear

algorithm to decide if an input list of integers is in nondecreasing order, so this problem is in P. There is a polynomial time algorithm to determine whether a given string is generated by a given context free grammar, so this problem, too, is in P. The recent discovery of a polynomial time algorithm for testing primality shows that primality testing is also in P.

The class P is quite stable under changes in the underlying computer model. This is primarily because polynomials are closed under composition, and there are polynomial-time simulations of the major models of computation by one another. For example, a problem that has a polynomial time algorithm on a log-cost random access machine necessarily also has a polynomial time algorithm on a one-tape Turing machine, and vice versa. The difference may be an $O(n^3)$ algorithm on the RAM and an $O(n^6)$ algorithm on the Turing machine, but both running times are polynomial.

Quantum computing provides the main current challenge to our idea that P is the same class for all “reasonable” models of computation. On a quantum computer (and *very* small quantum computers have been built), factoring an integer can be done in polynomial time using Peter Shor’s algorithm. We do not know of any polynomial time algorithm for factoring using a conventional sequential Von Neumann machine. This suggests that the problems solvable in polynomial time on quantum and conventional computers may not be the same. (As noted in an earlier lecture, the apparent difficulty of factoring is the basis of the RSA public key cryptosystem, so faster factoring algorithms potentially have nontrivial practical consequences.)

And what is NP?

The class NP is a set of decision problems that includes P as a subset, possibly a proper subset. The question of whether the containment is proper is precisely the question of whether $P = NP$. We introduce NP by example.

Consider the Travelling Salesman Problem, described as follows. The input is a list of cities, and for each pair of cities, a number representing the cost of a direct flight from one city to the other. One city is distinguished as the start city. The output is a minimum cost tour, that is, an ordering of all the cities, starting with the start city and including each city exactly once, with the property that the sum of the costs of the flights from each city to the next, and from the last city back to the start city, is minimized. The size n of the problem is the number of cities.

What is a brute force method to solve this problem? We could systematically enumerate all $n!$ permutations of all the cities, add up the costs associated with each possible solution, and keep track of the minimum possible cost. This shows that the problem is algorithmically solvable. The time for this algorithm is proportional to $n!$ times the time to compute the value of each possible solution. Assuming we have stored the flight costs in an array indexed by pairs of cities, we can compute the sum of the costs in time $O(n)$. Also, we can compute the next permutation in sequence in time $O(n)$, so $O(n \cdot n!)$ seems like a fair estimate of the time of this algorithm.

But how big is $n!$ in terms of polynomials and exponentials? A very useful approximation is Stirling’s (look it up if you are curious), but we can use something rather less accurate. $n!$ is a product of n numbers, each less than or equal to n , so

$$n! \leq n^n = 2^{n \log_2 n},$$

for all integers $n \geq 1$. Thus, the running time of our brute force algorithm is $O(2^{(n+1) \log_2 n})$. This is worse than exponential in the definition we gave above, since the exponent contains a nonlinear function of n .

Brute force is all very well, but isn’t there something better? Isn’t there a polynomial time algorithm for this problem? This is just the question of whether $P = NP$ in disguise. That is, we can show that there is a polynomial time algorithm for the Travelling Salesman Problem if and only if $P = NP$. This is a central, famous, and long-standing open problem in the area of computer science. It is the only problem

from computer science that made it onto the Clay Institute's list of seven "Millennial Problems" – intended to stimulate mathematical work on these problems during the 21st century as Hilbert's 1900 list did in the 20th century.

NP is a set of decision problems. The Travelling Salesman Problem as stated above is not a decision problem. However, there is a standard trick to turn it into a decision problem. For the decision version of the problem, in addition to the other inputs, there is an integer bound B , and the question is whether or not there exists a tour of cost at most B . This is clearly a decision problem, and if we could solve it in polynomial time, then we could use it as a subroutine to find a minimum cost tour in polynomial time. (Binary search for the minimum cost, and then selectively increase the cost of travel between a pair of cities to see if a minimum cost tour must use that pair.)

A decision problem is in NP if there are polynomial size, polynomial time checkable "proofs" for yes answers. Suppose I claim for a given Travelling Salesman input, that there indeed is a tour of cost less than 3000. You don't have to take my word for it – you can say "show me!" I then provide a tour, and you can check for yourself (in polynomial time) that the cost of the tour is less than 3000. For these problems, yes and no are asymmetric – there is, in general, no easily checked proof that I can give you to show you that there is NO tour of cost less than 3000.

The existence of polynomial size, polynomial time checkable proofs of yes answers shows that the decision version of the Travelling Salesman Problem is in NP.

Another problem in NP

The CNF Satisfiability Problem is another problem in NP. The input is a Boolean formula in conjunctive normal form, for example:

$$(x_1 + x'_2 + x_4) \cdot (x'_1 + x'_3 + x'_4) \cdot (x'_1 + x_2 + x_3).$$

A Boolean formula is satisfiable if there exists an assignment of truth values to its variables that makes it true. The output for the CNF Satisfiability Problem is whether or not the input formula is satisfiable.

For the given formula, the answer is yes. There is a polynomial size, polynomial time checkable "proof" that it is satisfiable, namely, an assignment of truth values to the variables that makes the formula true, for example:

$$x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0.$$

Clearly, given such an assignment, there is a polynomial time algorithm to check whether it satisfies the formula. Thus, the CNF Satisfiability Problem is also in NP.

Like the Travelling Salesman decision problem, the CNF Satisfiability Problem is complete in NP. There is not time in this course to give the definition of NP-completeness, but one of its corollaries is that if there is a polynomial time algorithm for any one of the NP-complete problems, then there is a polynomial time algorithm for every problem in NP, and $P = NP$. (Note that every problem in P is trivially in NP, because we can check both yes and no answers for ourselves in polynomial time, with no "proofs" necessary.) There are hundreds (perhaps thousands) of NP-complete problems known. The question of whether any of them has a polynomial time algorithm is open.