14 Computational Complexity and Polynomial Time

A program is only said to compute, or decide, a problem if it eventually stops on every input. For computability, it does not matter how long the program takes to produce an answer.

But, from a practical standpoint, time matters. The data switches that form the backbone of the internet process a data packet roughly every 10-20 nanoseconds. They can only afford to run extremely fast algorithms. For most everyday purposes, an algorithm that gets its answer in a few seconds is fast enough, and for some problems, a program that takes a few minutes or hours is fast enough. But a program that takes years is usually not acceptable.

With this section we start to look at what can be computed efficiently. To do that, we need to choose a reasonable definition of an "efficient" algorithm, and to study which problems can be solved efficiently under that definition.

A definition of efficiency cannot be based on any fixed amount of time. The larger the input is, the longer we expect a program to take, so a reasonable notion of efficiency should be concerned with the time that a program takes as a function of the length of the input. Also, a faster computer will produce an answer faster even for the same algorithm, and we need a way to get processor speed out the way.

14.1 The Class P

A program performs a sequence of instructions, and the time that it uses is just a count of the number of instructions that it performs before it stops. In this view, time has no units; it is a pure number. We assume that it takes at least one instruction to look at a symbol in the input and at least one instruction to write one symbol in the output.

Definition 14.1. *Time(p,x)* is the number of instructions that program p takes when it is run on input x. If $p(x)\uparrow$, then $\text{Time}(p,x) = \infty$.

Definition 14.2. Let $f : \mathcal{N} \to \mathcal{N}$. A program p runs in *time $O(f(n))$* if there exists constants a and c so that, for all $n > a$ and all strings x of length n, $\text{Time}(p, x) \le c \cdot f(n)$.

Definition 14.3. Program p runs in *polynomial time* if there exists a positive integer k so that p runs in time $O(n^k)$. When p runs in polynomial time, we say that p is a *polynomial-time algorithm*.

Definition 14.4. *P* is the class of all *decision problems* that have polynomial-time algorithms.

Notice that P is a set of problems, not a set of programs. It makes no sense to say that a program is in P.

14.2 Examples of Problems that Are In P

14.2.1 Example: is x a Palindrome?

A *palindrome* is a string such as "*aabaa*" that is the same forwards and backwards. The *Palindrome Problem* is the following decision problem.

> **Input.** String x.
>
> **Question.** Is x a palindrome?

The Palindrome Problem is in P. You should be able to figure out an algorithm that solves the Palindrome Problem in time $O(n)$.

14.2.2 Example: Does DFA M Accept x?

We have seen that it is computable to determine if a given DFA M accepts a given string x. It should also be clear that an algorithm can do that in time that is proportional to the product of the length of the description of M and the length of x. (You should be able to do much better than that, but it is fast enough for our purposes.) If the input has length n, that is surely $O(n^2)$ time.

14.2.3 Example: Is $x \cdot y = z$?

How might you solve the following decision problem?

> **Input.** Three positive integers x, y and z.
>
> **Question.** Is $x \cdot y = z$?

The obvious thing to do is to multiply x and y and check whether the answer is z. It is important to notice that the time needed to do that is not a constant, since x, y and z can be very large. When a number occurs in the input, its length is the number of digits needed to write it down. For example, 490 has length 3. The algorithm that you learned in elementary school multiplies an i-digit number by a j-digit number in time $O(ij)$. The length of the input is the total number of symbols that it contains. Clearly, if x has length i and y has length j and the total length of the input if n, then $i < n$ and $j < n$, and it is possible to check an integer product in time $O(n^2)$. That is polynomial time.

14.2.4 Example: Is x a Prime Number?

The *Primality Problem* is the following decision problem.

> **Input.** A positive integer x.
>
> **Question.** Is x prime?

Here is an algorithm that solves the Primality Problem.

```
"{prime(x):
   i = 2
   while i < x
      if n mod i == 0
         return 0
      i = i + 1
   return 1
}"
```

How much time does that algorithm take? It goes around the loop $x - 2$ times. If x is n digits long, then x is in the rough vicinity of 10^n. (Assuming no leading 0s, $10^{n-1} \le x < 10^n$.) The division algorithm that you learned in elementary school divides an m-digit number by an n-digit number in time $O(mn)$. Puting that all together, we find that our algorithm for testing whether an integer is prime takes time $O(n^2 10^n)$. But function $f(n) = 10^n$ grows faster than any polynomial. That is, for every k,

$$\lim_{n\to\infty} \frac{10^n}{n^k} = \infty.$$

Our primality-testing algorithm is not a polynomial-time algorithm.

What does that tell us about whether the Primality Problem is in P? Absolutely nothing! You can write a bad algorithm for any computable problem. The issue is not whether there is a bad algorithm to solve the Primality Problem, but whether there is a polynomial-time algorithm for it.

As it turns out, the primality problem is in P. It was long conjectured to be in P, and was shown to be in P in 2003.

14.3 The Validity Problem for Propositional Logic

Chapter 1 defines the notion of a valid propositional formula. The *Validity Problem for Propositional Logic* (or simply the Validity Problem) is the following decision problem.

> **Input.** A propositional formula ϕ.
>
> **Question.** Is ϕ valid?

You already know an algorithm that solves that problem: truth tables. Suppose that ϕ has v variables. Then a truth table for ϕ has 2^v rows, and determining validity takes time at least 2^v.

Determining whether an algorithm runs in polynomial time requires determining the algorithm's running time in terms of the length n of the input. The number of variables v is surely shorter than the total length of ϕ, but how close to n can v be? As long as we allow long variable names (such as x_1, x_2, $\dots$), it is easy to write an interesting propositional formula of length n with at least $\sqrt{n}$ variables. A little calculus shows that

$$\lim_{n\to\infty} \frac{2^{\sqrt{n}}}{n^k} = \infty$$

for every k, so the truth table algorithm does not run in polynomial time.

What does that have to say about whether the Validity Problem is in P? Absolutely nothing! Nobody knows a polynomial-time algorithm for the Validity Problem, but that lack of knowledge also is not convincing evidence that the Validity Problem is not in P.

Beginning with the next section, we begin to address the question of whether the Validity Problem is in P.

prev **next**