

Chapter 7

Polynomial-Time Computability

Feasible computability

Up to now, we have approached computability from the point of view that there should be no constraints on the time required for a particular computation, or on the amount of memory space that might be required. The result is that some total computable functions will take a very long time to compute. If a function f grows very rapidly, then for large x it will take a long time simply to generate the output $f(x)$. But there are also bounded functions that require a large amount of time.

Of course, the complexity of computing $f(x)$ depends on the program used. Suppose we define $M(e, x)$ to be the number of steps the program with Gödel number e uses on input x :

$$\begin{aligned} M(e, x) &= \mu t T(e, x, t) \\ &= \mu t (\llbracket x \rrbracket(x) \downarrow \text{ in } \leq t \text{ steps}) \end{aligned}$$

Then the computable partial function M has the following two properties:

- $M(e, x) \downarrow \iff \llbracket e \rrbracket(x) \downarrow$.
- The ternary relation $\{(e, x, t) \mid M(e, x) \downarrow \text{ and } M(e, x) \leq t\}$ is a computable relation.

In fact, that ternary relation is nothing but the relation $T(e, x, t)$. It can be thought of more simply as $\{(e, x, t) \mid M(e, x) \leq t\}$ if we adopt the convention that $M(e, x) = \infty > t$ whenever $M(e, x) \uparrow$. (That is, computations that never halt take infinite time.)

Digression: Michael Rabin and Manuel Blum have developed a theory of “axiomatic complexity” based on having a computable partial function M meeting the two conditions listed above. Such a function M is called a *measure of complexity*. Not only is time (i.e., the number of steps) a measure of complexity, but so is space. That is, if we define $M'(e, x)$ to be the sum of the largest number of symbols that get put into the registers (this being a measure of the “space” program e uses on input x) whenever $\llbracket e \rrbracket(x) \downarrow$, then M' also meets the conditions to be a measure of complexity.

The following theorem asserts that we can find subsets of \mathbb{N} whose decision problems are arbitrarily difficult. For example, suppose we take some function that grows rapidly, such as $h(x) = 2^{2^x}$. The theorem provides a subset A of \mathbb{N} that on the one hand is computable (so there *are* programs that decide membership in A), and on the other hand has the feature that *any* program

that decides membership in A will need, on input x , at least 2^{2^x} steps, with the exception of finitely many inputs. (Some programs might have a built-in table for the first thousand members of A , so that the program is very fast for small values of x . But from some point on, the program will be slow.)

Rabin's theorem (1960): For any total computable function h , we can find a total computable function $f : \mathbb{N} \rightarrow \{0, 1\}$ such that for any index i of f ,

$$M(i, x) > h(x) \text{ except for finitely many values of } x.$$

Proof outline: We will construct f in such a way that whenever some number j is an index for a fast function, then f differs from that function.

In calculating $f(x)$ for a particular x , we might “eliminate” some number j . (Eliminating j will assure that it cannot be an index of f .) So suppose we already know what numbers, if any, have been eliminated in computing $f(0), \dots, f(x-1)$.

To compute $f(x)$, we first find the least $j \leq x$ (if any) such that

- j is fast at x , that is, $M(j, x) \leq h(x)$
- j has not already been eliminated in computing $f(0), \dots, f(x-1)$.

If there is no such j , then let $f(x) = 0$. But if we find such a j , then we eliminate it and we let $f(x) = 1 \div \llbracket j \rrbracket(x)$. (This assures that j cannot be an index for f , because $\llbracket j \rrbracket$ and f differ at x .)

That completes the construction of f . Does it work? At least $f : \mathbb{N} \rightarrow \{0, 1\}$ and f is total and computable. Now let i be any index of f . Then i was never eliminated. Why not? Take any x large enough that $x \geq i$ and anything below i that ever was going to be eliminated has been eliminated before we come to $f(x)$. Then in computing $f(x)$ we would have eliminated i if $M(i, x) \leq h(x)$ had held. So it did not hold; that is, $M(i, x) > h(x)$. \dashv

The conclusion of Rabin's theorem, that some computable functions are really hard to compute, conforms to our informal feeling about such matters. What is more unexpected is that the feeling can be formulated as a precise theorem.

Is there a more restricted concept of “feasibly computable function” where the amount of time required does not grow beyond all reason, where the amount of time required is an amount that might actually be practical, at least when the input is not absurdly large? To this very vague question, an exact answer has been proposed.

Call a function f *polynomial-time computable* (or for short, P-time computable) if there is a program e for f and a polynomial p such that for every x , the program e computes $f(x)$ in no more than $p(|x|)$ steps, where $|x|$ is the length of x .

This definition requires some explanation and support. If f is a function over Σ^* , the set of words over a finite alphabet Σ , then of course $|x|$ is just the number of symbols in the word x . If f is a function over \mathbb{N} , then $|x|$ is

the length of the *numeral* for x . (Here we come again to the fact that effective procedures work with numerals, not numbers.) So if we use base-2 numerals for \mathbb{N} (either binary or dyadic notation), then $|x|$ is about $\log_2 x$.

Moreover, we were vague about how the number of steps in a computation was to be determined. Here the situation is very encouraging: The class of P-time computable functions is the same, under the different reasonable ways of counting steps. For definiteness, we adopt the following conventions: The computing is to be done by register machines operating on words over the alphabet $\{0, 1\}$, and the numerals are to be binary numerals. Thus, for example, in computing $f(2055)$ we are allowed at most $p(12)$ steps, because the binary numeral for 2055 has twelve bits.

The “industry standard” conventions are to use Turing machines and binary numerals. Here we have opted for register machines in order to be able to make use of the material already developed for register machines.

Earlier, we came across the encouraging fact that many different ways of formalizing the concept of effective calculability yielded exactly the same class of functions. For example, the class of functions computable by Turing machines operating on words over $\{0, 1\}$ coincides with the class of functions computable by register machines operating on words over $\{0, 1\}$. As remarkable as that fact is, even more is true. The number of steps required in one case is bounded by a polynomial in the number of steps required by the other. For example, there exists a polynomial p (of moderate degree) such that a computation by a Turing machine that requires n steps can be simulated by a register machine that requires no more than $p(n)$ steps. Consequently the concept of a P-time computable function is robust: We get the same class of functions, regardless of which choice we make. To be sure, the degrees of the polynomials will vary somewhat, but the class of P-time functions is unchanged. Moreover, this equivalence extends to other “reasonable” formalizations of computability.

Encouraged by this result, and inspired in particular by 1971 work of Stephen Cook, people since the 1970’s have come to regard the class of P-time functions as the correct formalization of the idea of functions for which computations are feasible, without totally impractical running times.

By analogy to Church’s thesis, the statement that P-time computability corresponds to feasibly practical computability has come to be known as *Cook’s thesis* or the *Cook–Karp thesis*. (The concept of P-time computability appeared as early as 1964 in work of Alan Cobham. Jack Edmunds in 1965 pointed out the good features of P-time algorithms. Richard Karp in 1972 extended Cook’s work.)

So what are the P-time computable functions? As a lower bound, we can show that all of the polynomial functions are P-time computable, as are some functions that grow faster than any polynomial.

As an upper bound, we can say that the P-time computable functions form a subclass of the primitive recursive functions:

Theorem: Any P-time computable function is primitive recursive.

Proof idea: We saw earlier that any function computable in primitive re-

cursive time is primitive recursive. The argument there can be adapted to the present situation, where we are using register machines over a two-letter alphabet. \dashv

But this is a very high upper bound; the converse to this result does not hold. We will see that many primitive recursive functions are *not* P-time computable. There is a limit to the growth rate of P-time computable functions, imposed by the fact that printing an output symbol takes a step. That is, we have the following constraint:

Growth limitation property: If f is computable in time bounded by the polynomial p , then $|f(x)| \leq |x| + p(|x|)$.

Proof: Writing a symbol requires a step. \dashv

This prevents exponential functions from being P-time computable; there is not enough time to write down the result.

Corollary:

We can conclude that the class of P-time computable functions is more than the class of polynomials, but less than the class of primitive recursive functions.

Often P-time computability is presented in terms of acceptance of languages (i.e., sets of words). Where Σ is the finite alphabet in question, consider a language $L \subseteq \Sigma^*$. We say that $L \in P$ if there is a program and a polynomial p such that whenever a word w is in L , then the program halts on input w (i.e., it “accepts” w) in no more than $p(|w|)$ steps, and whenever a word w is not in L , then the program never halts on input w (i.e., the program does not accept w). This is equivalent to saying that the characteristic function of L is P-time computable, because we can add to the program an alarm clock that rings after time $p(|w|)$. For example, it is now known that the set of prime numbers (as a set of words written in the usual base-10 notation) belongs to P.

Of course, if the characteristic function of L is P-time computable, then so is the characteristic function of its complement, \bar{L} . That is, $P = \text{co-P}$, where co-P is the collection of complements of languages in P.

Informally, L is in P if L is not only a decidable set of words, but moreover there is a “fast” decision procedure for P—one that we can actually implement in a practical way. For example, finite graphs can be coded by words over a suitable finite alphabet. The set of 2-colorable graphs (i.e., the set of graphs that can be properly colored with two colors) is in P, because coloring a graph with two colors does not involve any backtracking; either the coloring succeeds or we find a cycle of odd length. The set of graphs with an Euler cycle is in P, because it is fast to check that the graph is connected and that every vertex has even degree.

What about 3-colorable graphs, or graphs with Hamiltonian cycles? Here there are no known fast decision procedures. But there are weaker facts: Given a proper coloring with three colors, it is fast to verify that it is indeed a proper

coloring. Given a Hamiltonian cycle, it is fast to verify that it is indeed Hamiltonian. Both 3-colorable graphs and Hamiltonian graphs are examples of languages that belong to a class known as NP.

One way to define NP is to use *non-deterministic* Turing machines. (The symbols “NP” stand for “non-deterministic polynomial time.”) Back in Chapter 1, the definition of a Turing machine demanded that a machine’s table of quintuples be unambiguous, that is, that no two different quintuples have the same first two components. By simply omitting that demand, we obtain the concept of a non-deterministic Turing machine. A computation of such a machine, at each step, is allowed to execute *any* quintuple that begins with its present state and the symbol being scanned. Then we say that $L \in \text{NP}$ if there is a non-deterministic Turing machine M and a polynomial p such that whenever a word w is in L , then *some* computation of M starting from input w halts in no more than $p(|w|)$ steps, and whenever a word w is not in L , then *no* computation of M starting from input w *ever* halts. (An accepting computation can be thought of as having made a number of lucky guesses.)

There is an equivalent, and somewhat more workable, characterization along the lines of Σ_1 definability: $L \in \text{NP}$ if and only if there is binary relation $R \in \text{P}$ and a polynomial p such that for every word w ,

$$w \in L \iff \exists y[|y| \leq p(|w|) \text{ and } R(w, y)].$$

Another example of a language in NP is SAT, the set of satisfiable formulas of sentential logic. The truth-table method for determining whether a formula with n sentence symbols is satisfiable involves forming all 2^n lines of the formula’s truth table, and looking to see if there is a line making the formula true. But this is not a feasible algorithm, because 2^{80} microseconds greatly exceeds the age of the universe. But if we (non-deterministically) guess the correct line of the table, then we can quickly verify that the formula is true under that line.

There is a clear analogy between computable and c.e. sets on the one hand, and P and NP on the other hand. The computable sets are decidable; the sets in P are decidable by fast algorithms. And c.e. sets are one existential quantifier away from being computable; sets in NP are one existential quantifier away from being in P. Moreover, there are c.e. sets that are complete with respect to \leq_m ; there are NP sets with a similar property. Say that L_1 is P-time *reducible* to L_2 if there is a P-time computable (total) function f that many-one reduces L_1 to L_2 . The following result was proved independently by Cook (1971) and Leonid Levin (1973):

Cook–Levin theorem: SAT is in NP, and every NP language is P-time reducible to SAT.

In other words, SAT is NP-complete. Karp has shown that many other NP languages (3-colorable graphs, Hamiltonian graphs, and others) are NP-complete.

P versus NP

How far does the analogy between “NP” and “c.e.” go? We know that there are non-computable c.e. sets, and a set is computable if and only if both it and its complement are c.e. While it is clear that $P \subseteq NP \cap \text{co-NP}$ (that is, every language in P is also in NP, as is its complement), it is not known whether $P = NP$, or if NP is closed under complement.

The diagonalization that produces a non-computable c.e. set K was “relativized” in Chapter 5 to show that for any fixed oracle B , there is a set B' that is c.e. relative to B but not computable relative to B . Might some diagonal argument produce a set in NP that was not in P? Would that argument then relativize? The definitions of P and NP extend easily to P^B and NP^B , where the computations can query the oracle B (in one step).

In a 1975 paper, Theodore Baker, John Gill, and Robert Solovay showed that there are oracles B and C such that on the one hand $P^B = NP^B$ and on the other hand $P^C \neq NP^C$. This result suggests that the “P versus NP” question is difficult, because whatever argument might settle the question cannot relativize in a straightforward way. It has also been shown that if we choose the oracle B at *random* (with respect to the natural probability measure on $\mathcal{P}\mathbb{N}$), then $P^B \neq NP^B$ with probability 1.

The P versus NP question remains the outstanding problem in theoretical computer science. In recognition of this fact, the Clay Mathematics Institute has offered a million-dollar prize for its solution.

	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
$f(n) = n^3$	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
$f(n) = n^5$	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.	13.0 min.
$f(n) = 2^n$	1.0 sec.	17.9 min.	12.7 days	35.7 years	366 centuries
$f(n) = 3^n$	58 min.	6.5 years	3,855 centuries	2×10^8 centuries	the age of the universe

The length of $f(n)$ microseconds, for several choices of $f(n)$
