

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.

In this chapter, we want to adopt a different attitude, and to pay some attention to the running time that a program might require. For the most part, we will not attempt to give complete rigorous proofs. Instead, we will concentrate on introducing the concepts and describing the ideas behind the results, and on giving pointers to topics for further study.

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 :

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

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

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

In fact, that ternary relation is nothing but the primitive recursive relation $T(e, x, t)$. It can be thought of more simply as $\{\langle e, x, t \rangle \mid M(e, x) \leq t\}$ if we adopt the convention that whenever $M(e, x) \uparrow$, then $M(e, x) = \infty$ and $\infty > t$ for any t . (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

⁰Chapters 5, 6, and 7 are largely independent, and can be read in any order.

“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.

Definition: Call a function f *polynomial-time computable* (or for short, P-time computable) if there exists 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 some 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. (In particular, it should be emphasized that we are *not* allowed $p(2055)$ steps. If our program is trying to find the smallest prime factor of a 900-bit number, then we are allowed $p(900)$ steps, and not $p(2^{900})$ steps.)

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.)

The following table gives a small illustration. Suppose the input string consists of n bits. If we can execute a million steps a second, then the time required to execute $q(n)$ steps is, of course, $q(n)$ microseconds. The table converts $q(n)$ microseconds to more comprehensible units, for five choice of n and four choices of q .

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

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.

But first, we will look at upper bounds. For a start, 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 (in Exercise 6, page 320) that any function computable in primitive recursive 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.

Suppose we have a program that computes $f(x)$ in no more than $p(|x|)$ steps. Now $|x|$ is a lot smaller than x (with the exception of $x = 0$) and we can assume that the polynomial p has positive coefficients, and hence is monotonic. So the number of steps is bounded by $p(x)$, a primitive recursive function of x .

(Actually, we can get sharper bounds. For some constant c , we can get $p(|x|) \leq x + c$. But for our present purposes, the weaker result suffices.) \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 $|f(x)| \leq p(|x|)$.

Proof: Initially, register 0 contains the empty word. Adding a symbol to register 0 requires at least one step. And we need to add $|f(x)|$ symbols. \dashv

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

Lemma: Where $|x|$ is the length of the binary numeral for x , the following hold for all $x \neq 0$.

- (a) $2^{|x|-1} \leq x \leq 2^{|x|} - 1$.
- (b) $|x| = \lfloor \log_2 x \rfloor + 1$.

Proof: (a) We must have x somewhere between the smallest $|x|$ -bit number and the largest:

$$1 \underbrace{00 \cdots 0}_{|x|-1} \leq x \leq 1 \underbrace{11 \cdots 1}_{|x|-1}$$

And this is the inequality stated by (a).

(b) We can write part (a) as $2^{|x|-1} \leq x < 2^{|x|}$. To this inequality, apply the (monotonic) \log_2 function:

$$|x| - 1 \leq \log_2 x < |x|$$

Now round down: $|x| - 1 = \lfloor \log_2 x \rfloor$. \dashv

Corollary: The exponential function $x \mapsto 2^x$ is *not* P-time computable.

Proof: How long would it take to write out 2^x ? We have $|2^x| = x + 1$, and by the lemma this exceeds $2^{|x|-1}$. The function $t \mapsto 2^{t-1}$ grows faster than any polynomial. We conclude that the number of steps needed to write out 2^x cannot be bounded by a polynomial in $|x|$. (For example, if x is a 60-bit number, then 2^x has more than 2^{59} bits. At a million bits per second, it will take hundreds of centuries to write out 2^x .) \dashv

To balance this negative result, let's look at some functions that *can* be computed in P-time.

Example: The squaring function $x \mapsto x^2$ is P-time computable. First, think about how a *human* goes about squaring an n -bit number x . The standard procedure we all learned in the third grade involves building up an $n \times n$ array and then adding up the $2n$ columns. Building up the array is not difficult (multiplying by 0 or by 1 is easy), and the time it takes us will be proportional to n^2 . Then adding up the $2n$ columns, each of height no more than n , will again take us some amount of time proportional to n^2 . Altogether, the number of steps will be bounded by a quadratic in n .

A register machine is not a human, so there is more work to be done. To verify that squaring is P-time computable, we need to program the foregoing human procedure, and then obtain a bound on how long the program takes. The program can start by making a second copy of x (in $k|x|$ steps) and then

calling the multiplication program outlined on page 327. The total number of steps will be bounded by a quadratic in $|x|$.

One way *not* to do squaring is by repeated addition. That is, a program that computes x^2 by adding x to itself x times will require exponential time (because x is about $2^{|x|}$) and $2^{|x|}$ grows faster than any polynomial in $|x|$.

Proposition: The composition $g \circ f$ of P-time computable functions is again P-time computable.

Proof: Assume that program \mathcal{M} computes f in time bounded by the polynomial p , and that program \mathcal{N} computes g in time bounded by the polynomial q . Then the program \mathcal{M} followed by some minor housekeeping followed by \mathcal{N} will compute $g \circ f$. How long does it take? The number of steps on input x is bounded by

$$p(|x|) + \text{small amount} + q(|f(x)|).$$

By the growth limitation property, $|f(x)| \leq p(|x|)$. We may assume that the polynomial q has only positive coefficients, and hence is monotonic. So our time bound becomes

$$p(|x|) + \text{small amount} + q(p(|x|))$$

which is a polynomial in $|x|$. \dashv

For a function $f(x, y)$ of two variables to be considered P-time computable, there must be a polynomial p such that some program produces $f(x, y)$ in no more than $p(|x| + |y|)$ steps. That is, we can look at $|x| + |y|$ as the total length of the input. We can handle functions of more variables in a similar way.

The preceding proposition, regarding composition $g \circ f$, can be extended to cover compositions $g(f_1(\vec{x}), \dots, f_k(\vec{x}))$ of functions of several variables.

Example: The addition function $\langle x, y \rangle \mapsto x + y$ is P-time computable. As outlined on page 327, there is a program that will give $x + y$ in a number of steps bounded by $k \max(|x|, |y|)$ for a constant k . And $\max(|x|, |y|) \leq |x| + |y|$.

Example: The multiplication function $\langle x, y \rangle \mapsto xy$ is P-time computable. This is a lot like the squaring function.

Proposition: Any polynomial function $p(x)$ is P-time computable.

Proof outline: We build up the polynomial piece by piece, using the foregoing examples. Raising x to a power (that is, the function $x \mapsto x^k$) can be done by composition of multiplications. A monomial $x \mapsto cx^k$ involves one more multiplication, this time by a constant. Finally, we use a composition of additions. \dashv

Example: Let

$$f(x) = 2^{|x|^2} = 100 \cdots 0_{\text{two}}$$

where the string of 0's is $|x|^2$ long. This function can be computed in polynomial time. To append a string of 0's of length $|x|$ to the output, we use a loop where

each time through the loop we append one 0 and we erase one symbol from a copy of x . Now we put that loop inside another similar loop. Together, the loops append a string of 0's of length $|x|^2$, and they do this in quadratic (in $|x|$) time.

How fast does f grow? With help from a recent lemma, we obtain

$$f(x) = \left(2^{|x|}\right)^{|x|} \geq x^{|x|} \geq x^{\log_2 x}$$

so f grows faster than x^k for any fixed k . Thus we have here a P-time computable function that grows faster than any polynomial p , in the sense that $f(x)/p(x) \rightarrow \infty$ as x increases.

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). We have a finite alphabet $\Sigma = \{0, 1\}$. Over this alphabet, the set Σ^* of all words is the set of binary strings. (A binary string is the same thing as a base-2 numeral, except for the annoying problem of leading zeros. Ignoring that annoyance, we can think of Σ^* as being the same as the set of numerals for \mathbb{N} .) By a *language* L , we mean a set of words (i.e., a subset of Σ^*). We say that $L \in \text{P}$ if there is a program and a polynomial p such that the following hold:

- Whenever a word w is in L , then the program halts on input w (that is, it “accepts” w), and does so in no more than $p(|w|)$ steps.
- Whenever a word w is not in L , then the program never halts on input w (that is, the program does not accept w).

This definition is equivalent to one formulated in terms of P-time computable functions:

Theorem: A language L is in P if and only if its characteristic function C_L is P-time computable.

Proof idea: In one direction, this is easy: If C_L is P-time computable, then we can make an acceptance procedure that, given an input word w , computes $C_L(w)$ and then either halts or goes into an infinite loop.

It is the other direction that is interesting. Assume we have an acceptance procedure (which, in effect, is computing the semi-characteristic function c_L) that runs in time bounded by a polynomial p . We can add to the program an alarm clock that rings after time $p(|w|)$.

That is, given the input word w , we first compute $p(|w|)$. This does not take long; it can be done in a number of steps that is a polynomial in $|w|$. We store this number in a “timer” register.

Then we proceed as follows: With our right hand, we run the acceptance procedure on the word w ; with our left hand we decrement the timer. Or more precisely (since register machines don't have hands), we interleave two programs.

In odd-numbered steps we do the acceptance procedure; in even-numbered steps we decrement the timer.

If and when the the acceptance procedure halts (the timer will not have run out), then we give output Yes. If and when the timer runs out (the acceptance procedure will not have halted), we give output No. \dashv

As a corollary, we can conclude that whenever $L \in P$, then L , viewed as a set of numbers, is primitive recursive.

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

Example: It is now known that the set of prime numbers, as a set of words written in the usual base-2 notation (or base-10, for that matter), belongs to P. Because the set of primes is in P, it follows that the set of composite numbers is in P as well.

Non-example: Consider the function

$$f(x) = \text{the least prime divisor of } x$$

with the convention that $f(0) = 0$ and $f(1) = 1$. It is easy to see that f is primitive recursive. Despite the fact that the set of primes belongs to P, it is currently an open question whether or not f is P-time computable.

The concept of belonging to P extends in a natural way to binary relations R on the set of words (i.e., $R \subseteq \Sigma^* \times \Sigma^*$).

Informally, L is in P if L is not only a decidable set of words, but moreover there is a “fast” decision procedure for L —one that we can actually implement in a practical way. For example, finite graphs can be coded by binary strings. 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 the set of 3-colorable graphs and the set of Hamiltonian graphs are examples of languages that are “verifiable” in P-time. That is, we might not know fast decision procedures, but we do know how, given the correct evidence, to verify quickly that the evidence does indeed show that the graph has the claimed property. Such languages 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 \mathcal{M} , at each step, is allowed to execute *any* quintuple that begins with its present state and the symbol being scanned. So when we start \mathcal{M} on some input, there can be many possible computations, depending on which of the allowed quintuples it chooses to execute.

Then we say that $L \in \text{NP}$ if there is a non-deterministic Turing machine M and a polynomial p such that the following conditions hold:

- Whenever a word w is in L , then *some* computation of \mathcal{M} starting from input w halts, and does so in no more than $p(|w|)$ steps.
- Whenever a word w is not in L , then *no* computation of \mathcal{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:

Definition: For a language L , $L \in \text{NP}$ if there is binary relation $R \in \text{P}$ and a polynomial p such that for every word w ,

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

Examples: The set of three-colorable graphs (as a set of binary strings) is in NP. A graph w is three-colorable iff there exists some 3-coloring y such that w is properly colored by y (that is, adjacent vertices are always different colors).

Similarly, the set of Hamiltonian graphs is in NP. Here, the evidence y is a Hamiltonian cycle in the graph w .

Another example of a language in NP is SAT, the set of satisfiable formulas of sentential logic. The truth-table method taught in logic courses 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 r.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 r.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 r.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 (written $L_1 \leq_P L_2$) 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.

Digression: We have defined P-time reducibility in a way that is analogous to many-one reducibility. But there is another option: We could define a concept analogous to Turing reducibility. That is, we could specify that L_1 be polynomial-time decidable, but allow an oracle for L_2 .

P versus NP

How far does the analogy between “NP” and “r.e.” go? We know that there are non-computable r.e. sets, and a set is computable if and only if both it and its complement are r.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 r.e. set K was “relativized” in Chapter 6 to show that for any fixed oracle B , there is a set B' that is r.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.

Other complexity classes

And what might lie beyond NP? Since there is some analogy between NP and Σ_1 , what might be analogous to Σ_n ? And what might be computable in “exponential time,” where we allow the computing time to be bounded by $2^{p(|x|)}$ for a polynomial p ?

As indicated earlier, two reasonable measurements of complexity are *time* (the number of steps the computation executes before halting) and *space* (where we take the largest number of symbols a register ever contains in the course of the computation, and add these numbers up for all the registers).

These two measurements are related. If a computation halts in time t on input x , then the space used is bounded by $t+|x|$, because writing a symbol takes a step. What about the other direction? Suppose that a particular calculation from a program halts, having used space s . We want a bound on the time it used.

Consider the snapshots

[location counter, memory number]

that arise in the course of the computation. One thing we can be sure of is that no snapshot occurs twice. This is because if a computation ever hits a snapshot for a second time, then the computation will run forever, returning to this snapshot infinitely often. So for a calculation that halts, the time is bounded by the number of possible snapshots. And what is the number of possible snapshots? For a program with Gödel number e , there are at most $1 + \text{lh } e$ values for the location counter. And if the program addresses k different registers, there could be (for the alphabet $\{0, 1\}$) at most 2^{ks} different values for the memory number. Putting the pieces together, we see that for constants c and k (depending on the program), the running time is bounded by $c2^{ks}$.

We have been looking at the complexity class P. This class might just as well be called PTIME, because it uses time as the complexity measure. A language L (that is, a subset of $\{0, 1\}^*$) is in the class if its characteristic function at x is computable in *time* bounded by a polynomial $q(|x|)$ in $|x|$.

Analogously, define PSPACE as follows: A language L belongs to PSPACE if there is a program and a polynomial p such that the program computes the characteristic function of L at each word x using *space* at most $p(|x|)$. Then $P \subseteq \text{PSPACE}$ because a computation that uses time $q(|x|)$ can use at most space $|x| + q(|x|)$.

Next, we want to define EXPTIME and EXPSPACE. A language L belongs to EXPTIME if there is a program and a polynomial p such that the program computes the characteristic function of L at each word x in at most $2^{p(|x|)}$ steps. And a language L belongs to EXPSPACE if there is a program and a polynomial p such that the program computes the characteristic function of L at each word x using *space* at most $2^{p(|x|)}$.

We claim that $\text{PSPACE} \subseteq \text{EXPTIME}$. Suppose a computation uses space $p(|x|)$. Then for some constants c and k that depend on the program (but not on x), the running time is bounded by $c2^{kp(|x|)} = 2^{\ln c + kp(|x|)}$. We observe that the exponent here is a polynomial in $|x|$.

Moreover, $\text{EXPTIME} \subseteq \text{EXPSPACE}$. Suppose a computation uses time $2^{p(|x|)}$. Then the space is bounded by $|x| + 2^{p(|x|)}$. This is in turn bounded by $2^{q(|x|)}$ for a suitable polynomial q ; see Exercise 4.

Thus we are left with the inclusions

$$P \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE \subseteq PR$$

where PR is the class of primitive recursive languages. There are numerous questions that can be raised regarding these classes. For some of the questions, answers are known. And other questions remain open.

The subject of computational complexity is young and growing. See the list of references for some avenues well worth exploring.

Exercises

1. Show that the concept of P-time reducibility is reflexive and transitive. That is, show that we always have $L \leq_P L$, and that whenever $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$.
2. Assume that $L_1 \leq_P L_2$ and $L_2 \in P$. Show that $L_1 \in P$.
3. Assume that $L_1 \leq_P L_2$ and $L_2 \in NP$. Show that $L_1 \in NP$.
4. Assume that p is a polynomial. Show that for some polynomial q we have $z + 2^{p(z)} \leq 2^{q(z)}$ for all z .