

Lecture 17

Complexity and NP-completeness

Supplemental reading in CLRS: Chapter 34

As an engineer or computer scientist, it is important not only to be able to solve problems, but also to know which problems one can expect to solve efficiently. In this lecture we will explore the *complexity* of various problems, which is a measure of how efficiently they can be solved.

17.1 Examples

To begin, we'll review three problems that we already know how to solve efficiently. For each of the three problems, we will propose a variation which might not be so easy to solve.

- **Flow.** Given a flow network G with integer capacities, we can efficiently find an integer flow that is optimal using the Edmonds–Karp algorithm.
- *Multi-Commodity Flow.* In §13.3, we considered a variation of network flow in which there are k commodities which need to simultaneously flow through our network, where we need to send at least d_i units of commodity i from source s_i to sink t_i .
- **Minimum Cut.** Given an undirected weighted graph G , we can efficiently find a cut $(S, V \setminus S)$ of minimum weight.

Exercise 17.1. *Design an efficient algorithm to solve the minimum cut problem. (Hint in this footnote.¹)*

- *Maximum Cut.* What if we want to find a cut of maximum weight?
- **Minimum Spanning Tree.** Given an undirected weighted graph $G = (V, E)$, we know how to efficiently find a spanning tree of minimum weight. A spanning tree of G is a subgraph $G' = (V', E') \subseteq G$ such that
 - G' is connected and contains no cycles
 - $V' = V$.

¹ Hint: Turn G into a flow network and find a maximum flow. In Theorem 13.7, we saw how to turn a maximum flow into a minimum cut.

- *Steiner Tree*. What if, instead of requiring $V' = V$, we require $V' = S$ for some given subset $S \subseteq V$?

Each of these three variations is *NP-hard*. The typical attitude towards NP-hard problems is

Don't expect an efficient algorithm for this problem.

However, I prefer the following more optimistic interpretation:

If you find an efficient algorithm for this problem, you will get \$1 million.²

Unlike the pessimistic interpretation, the above statement is 100% factual. The so-called **P vs. NP** problem is one of seven important open research questions for which Clay Mathematics Institute is offering a \$1 million prize.³

17.2 Complexity

So far in the course, we have been ignoring the low-level details of the mathematical framework underlying our analyses. We have relied on the intuitive notion that our ideal computer is “like a real-world computer, but with infinite memory”; we have not worried about explicitly defining what a “step” is. The fact of the matter is that there are many reasonable *models of computation* which make these notions explicit. Historically the first one was the **Turing machine**, invented by Alan Turing, considered by many to be the founding father of computer science. The model we have been using throughout the course is similar to a Turing machine; the main difference is that a “step” on our ideal computer closely resembles a processor cycle on a modern computer, whereas the “steps” of a Turing machine involve sequentially reading and writing to the so-called *tape* which represents its memory. For this lecture, you won't have to go to the trouble of working with a particular model of computation in full detail; but it is worth noting that such details are important in theoretical computer science and should not be regarded as a triviality.

In what follows, we will define the complexity classes P and NP. Before doing so, we will need a couple more definitions:

Definition. A **decision problem** is a computation problem to which the answer is either “yes” or “no.” In mathematical language, we can think of a decision problem as a function whose domain is the set of possible input strings⁴ and whose range is $\{0, 1\}$ (with 0 meaning “no” and 1 meaning “yes”).

Definition. A **complexity class** is simply a set of decision problems.

Most of the problems we have considered so far in the course are not decision problems but rather **search problems**—they ask not just whether a solution exists, but also what the solution is. Given a search problem, we can derive decision problems which ask yes-or-no questions about the solution; for example, we might ask:

² I should warn you though, most computer scientists believe that it is not possible to find one. (In other words, most computer scientists believe that $P \neq NP$.)

³ One of the questions has already been solved, so currently there are six prizes remaining.

⁴ Strings over what alphabet? A typical choice is $\{0, 1\}$ (i.e., binary); another possible choice is the ASCII alphabet. The main reason the choice of alphabet matters is that it determines what “an input of size n ” is. The number 255 has size 8 in binary, size 1 in ASCII, and size 255 in unary. An algorithm whose running time is linear with respect to a unary input would be exponential with respect to a binary input.

Problem 17.1. Given a graph G and an integer k , is there a spanning tree of size less than k ?

For most real-world applications, search problems are much more important than decision problems. So why do we restrict our attention to decision problems when defining complexity classes? Here are a few reasons:

- The answer to a decision problem is simple.
- The answer to a decision problem is unique. (A search problem might have multiple correct answers, e.g., a given graph might have multiple minimum spanning trees.)
- A decision problem which asks about the answer to a search problem is at most as difficult as the search problem itself. For example, if we can find a minimum spanning tree efficiently, then we can certainly also solve Problem 17.1 efficiently.

17.2.1 P and NP

The existence of many different models of computation is part of the reason for the following definition:

Definition. “**Efficient**” means “polynomial-time.” An algorithm is **polynomial-time** if there exists a constant r such that the running time on an input of size n is $O(n^r)$. The set of all decision problems which have polynomial-time solutions is called **P**.

Polynomial time is the shortest class of running times that is invariant across the vast majority of reasonable, mainstream models of computation. To see that shorter running times need not be invariant, consider the following program:

```
1 Read the first bit of memory
2 Read the  $n$ th bit of memory
```

In our model of computation, which has random access to memory, this would take constant time. However, in a model of computation with only serial access to memory (such as a Turing machine), this would take linear time. It is true, though, that any polynomial-time program in our model is also polynomial-time on a Turing machine, and vice versa.

Search problems have the property that, once a solution is found, it can be verified quickly. This verifiability is the motivation for the complexity class NP.

Definition. A decision problem P is in **NP** if there exists a polynomial-time algorithm $A(x, y)$ such that, for every input x to the problem P ,

$$P(x) = 1 \iff \text{there exists some } y \text{ such that } A(x, y) = 1.$$

The string y is called a **witness** or **certificate**; the algorithm A is called a **verifier** or a **nondeterministic algorithm**.⁵

For example, in Problem 17.1, the witness y could be the spanning tree itself—we can certainly verify in polynomial time that a given object y is a spanning tree of size less than k .

⁵ The abbreviation NP stands for “nondeterministic polynomial-time.” The reason for this name is as follows. Imagine receiving x (the input to the problem P) but leaving the choice of y unspecified. The result is a set of possible running times of A , one for each choice of y . The problem P is in NP if and only if at least one of these possible running times is bounded by a polynomial $p(|x|)$ in the size of x . (The choice of y can depend on x , but p cannot depend on x .)

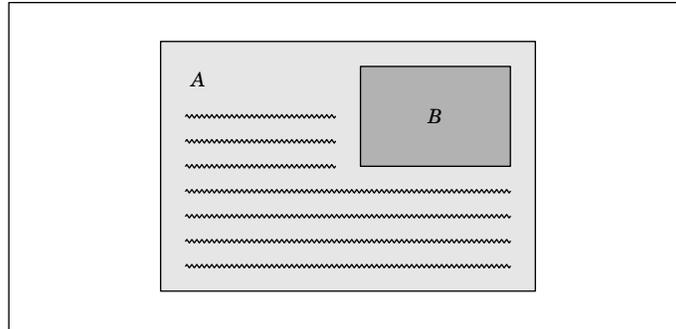


Figure 17.1. A Cook reduction of A to B is a program that would run in polynomial time on an *oracle machine*—that is, a Turing machine equipped with an oracle for B . If some day we find an efficient algorithm for B , then we can create an efficient algorithm for A by replacing the oracle with a subroutine.

Proposition 17.2. $P \subseteq NP$.

Proof. Given a decision problem P , view P as a function whose domain is the set of strings and whose range is $\{0, 1\}$. If P can be computed in polynomial time, then we can just take $A(x, y) = P(x)$. In this case, the verifier just re-solves the entire problem. \square

The converse to the above proposition is a famous open problem:

Problem 17.3 (P vs. NP). Is it true that $P = NP$?

The vast majority of computer scientists believe that $P \neq NP$, and so the P vs. NP problem is sometimes called the $P \neq NP$ problem. If it were true that $P = NP$, then lots of problems that seem hard would actually be easy: one such example is the algorithm search problem described in §17.3.

17.2.2 Polynomial-time reductions

It is possible to know that one problem is “at least as hard” as another without knowing exactly how hard each problem is. If problem A can be **polynomial-time reduced** to problem B , then it stands to reason B is at least as hard as A . There are two different notions of polynomial-time reduction, which we now lay out.

Definition. We say that the decision problem A is **Karp-reducible** to the decision problem B if there exists a polynomial-time computable function f such that, for every input x ,

$$A(x) = 1 \iff B(f(x)) = 1.$$

Definition. Problem A is **Cook-reducible** to problem B if there exists an algorithm which, given an oracle⁶ for B , solves A in polynomial time. (See Figure 17.1.)

Note that a Karp reduction is also a Cook reduction, but not vice versa. Historically, Karp reductions and Cook reductions correspond to different traditions.

Definition. A problem is **NP-hard** if every problem in NP can be Cook-reduced to it.

⁶ An **oracle** for B is a magical black-box which solves any instance of problem B in one step. A Cook reduction is a program which is allowed to do any of the normal things a computer program does, and is also allowed to query the oracle.

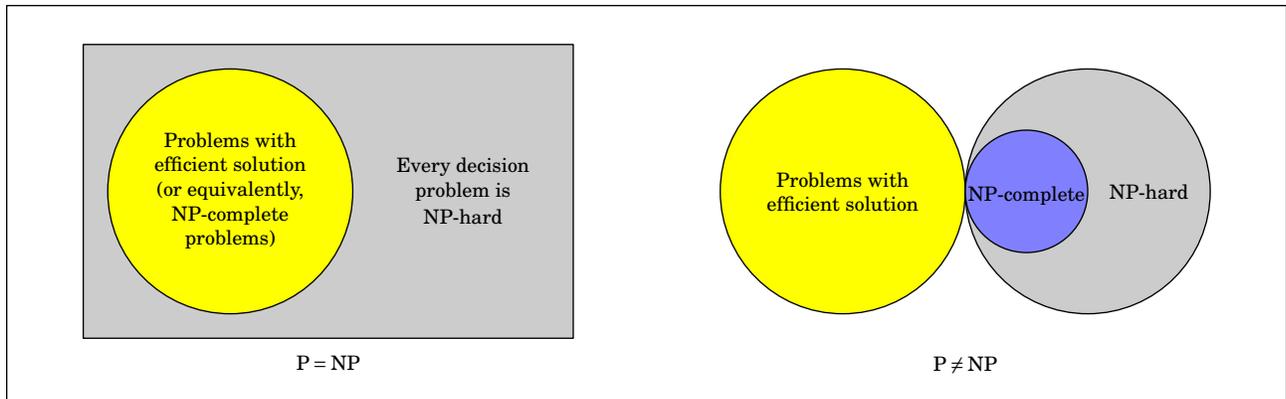


Figure 17.2. Left: If $P = NP$, then every decision problem is NP-hard (why?). Right: If $P \neq NP$.

Definition. A problem is **NP-complete** if it is both NP-hard and in NP.

Using the notion of NP-completeness, we can make an analogy between NP-hardness and big- O notation:

$O(f(n))$ <i>on the order of at most $f(n)$</i>	\rightsquigarrow	in NP <i>at most as hard as an NP-complete problem</i>
$\Theta(f(n))$ <i>tightly on the order of $f(n)$</i>	\rightsquigarrow	NP-complete <i>exactly as hard as any other NP-complete problem</i>
$\Omega(f(n))$ <i>on the order of at least $f(n)$</i>	\rightsquigarrow	NP-hard <i>at least as hard as an NP-complete problem</i>

Showing that a given problem is in NP is relatively straightforward (or at least, it is clear what the proof should look like): one must give a polynomial-time verifier. By contrast, it is much less clear how one might show that a given problem is NP-hard. One strategy is to reduce another NP-hard problem to it. But this strategy only works if one already knows certain problems to be NP-hard; it could not have been used as the first ever proof that a problem was NP-hard. That first proof was accomplished by Cook in 1971:

Theorem 17.4 (Cook’s Theorem). *3SAT is NP-complete.*

Problem 17.5 (3SAT). Given a set of atomic statements x_1, \dots, x_n , a *literal* is either an atom x_i or its negation $\neg x_i$. A *clause* is the disjunction (“or”) of a finite set of literals. The 3SAT problem asks, given a propositional formula $\varphi(x_1, \dots, x_n)$ which is the “and” of finitely many clauses of length 3, does there exist an assignment of either TRUE or FALSE to each x_i which makes $\varphi(x_1, \dots, x_n)$ evaluate to TRUE?

For example, one instance of 3SAT asks whether there exists an assignment of x_1, \dots, x_4 which makes the proposition

$$\underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\text{clause}} \wedge \underbrace{(\neg x_1 \vee x_3 \vee x_4)}_{\text{clause}} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{\text{clause}} \wedge \underbrace{(x_1 \vee x_3 \vee \neg x_4)}_{\text{clause}}.$$

evaluate to TRUE. The answer to this instance happens to be “yes,” as shown by the assignment

$$x_2 \mapsto \text{TRUE}, \quad x_1, x_3, x_4 \mapsto \text{FALSE}.$$

17.3 Example: Algorithm Search

In this section, we solve an algorithmic problem whose output is itself an algorithm:

Problem 17.6. Given an algorithmic problem P and a function $T(n)$, find an algorithm which runs in time at most $T(n)$, if such an algorithm exists. Output not just a description of the algorithm, but also a proof of correctness and running time analysis.

Proposition 17.7. *There exists a “meta-algorithm” which solves Problem 17.6 (but runs forever if no algorithm exists). If $P = NP$, then the running time of this meta-algorithm is polynomial in the size of the shortest possible output.*

The formats of both the input and the output of this algorithm deserve some explanation. By “an algorithmic problem,” we mean a mathematical description of the relationship between the input and the output. For example, we could express the notion of “a flow with magnitude at least k ” in symbols as

$$\left\{ \begin{array}{l} \forall u, v \in V \quad 0 \leq f(u, v) \leq c(u, v) \\ \forall u \in V \setminus \{s, t\} \quad \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u) \\ \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \geq k \end{array} \right\}.$$

In the case of a decision problem, this would be more along the lines of

$$\begin{cases} 1 & \text{if there exists } f \text{ satisfying the above} \\ 0 & \text{otherwise.} \end{cases}$$

Regarding the output of our meta-algorithm, we need a format for “a description of the algorithm.” One possibility is a text file containing the source code of an implementation in a particular programming language. Next, we need a format for “a proof of correctness and running time analysis.” For this we appeal to a *machine-checkable proof language* of the sort used by theorem-checking software. One such software suite is Coq.⁷ If you’ve never seen Coq before, I suggest you check it out!

The key to the proof of Proposition 17.7 is to consider the following problem:

Problem 17.8. Given an algorithmic problem P , a function $T(n)$ and an integer k , does there exist a solution to Problem 17.6 in which the output has length at most k ?

Problem 17.8 is in NP, so if $P = NP$ then there exists a polynomial-time algorithm for it.

Proof of Proposition 17.7. Consider the following search problem:

Problem 17.9. Given an algorithmic problem P , a function $T(n)$, an integer k , and a prefix string s , does there exist a solution to Problem 17.8 in which the output starts with s ?

⁷ <http://coq.inria.fr/>

Problem 17.9 is in NP, so if $P = NP$ then there exists a polynomial-time algorithm for it. Thus, if we let $|P|$ denote the length of the description of P , $|T|$ the length of the definition of the function T , and $|s|$ the length of s , then there exist constants a, b, c, d such that Problem 17.9 has a solution $A(P, T, k, s)$ which runs in time

$$O(|P|^a |T|^b k^c |s|^d).$$

We can use this algorithm to solve Problem 17.8 by probing longer and longer prefixes s . For example, supposing we use the 26-letter alphabet A, \dots, Z for inputs, we would proceed as follows:

- Run $A(P, T, k, A)$. If the answer is 0, then run $A(P, T, k, B)$. Proceed in this way until you find a prefix which returns 1. If all 26 letters return 0, then the answer to Problem 17.8 is “no.”
- Otherwise, let’s say for the sake of a concrete example that $A(P, T, k, F) = 1$. Then, run

$$A(P, T, k, FA), \quad A(P, T, k, FB), \quad \text{etc.},$$

until you find a two-letter prefix that returns 1.

- Proceeding in this way, the prefix s will eventually become the answer to Problem 17.8.

The above procedure solves Problem 17.8. Because the length of s ranges from 1 to at most k , the running time is

$$\begin{aligned} & O\left(26|P|^a |T|^b k^c 1^d + 26|P|^a |T|^b k^c 2^d + \dots + 26|P|^a |T|^b k^c k^d\right) \\ &= O\left(|P|^a |T|^b k^c \left(1^d + 2^d + \dots + k^d\right)\right) \\ &= O\left(|P|^a |T|^b k^c k^{d+1}\right) \\ &= O\left(|P|^a |T|^b k^{c+d+1}\right). \end{aligned}$$

Thus, we have done more than just show that Problem 17.8 is in P. We have shown that, for some constants α, β, γ , there exists a solution to Problem 17.8 which runs in time $O(|P|^\alpha |T|^\beta k^\gamma)$ and also *returns the full algorithm-and-proofs*, not just 0 or 1.

To conclude the proof, our meta-algorithm is to run the above procedure for $k = 1, 2, 3, \dots$. If the shortest possible algorithm-and-proofs has length ℓ , then the running time of the meta-algorithm is

$$O\left(|P|^\alpha |T|^\beta 1^\gamma + |P|^\alpha |T|^\beta 2^\gamma + \dots + |P|^\alpha |T|^\beta \ell^\gamma\right) = O\left(|P|^\alpha |T|^\beta \ell^{\gamma+1}\right). \quad \square$$

MIT OpenCourseWare
<http://ocw.mit.edu>

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.