Complexity classes P and NP

Millennium Prize Problems

P versus NP

The Hodge conjecture

The Poincaré conjecture

The Riemann hypothesis

Yang–Mills existence and mass gap

Navier-Stokes existence and smoothness

The Birch and Swinnerton-Dyer conjecture

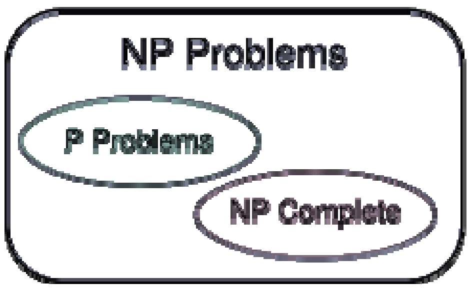

Diagram of complexity classes provided that **P** ≠ **NP**. The existence of problems outside both **P** and **NP**-complete in this case was established by Ladner.[1]

The relationship between the **complexity classes** **P and NP** is an unsolved question in theoretical computer science. It is generally agreed to be the most important such unsolved problem. It is also generally agreed to be one of the most important unsolved problems in mathematics; the Clay Mathematics Institute has offered a $1 million US prize for the first correct proof.

In essence, the **P** = **NP** question asks: if positive solutions to a YES/NO problem can be *verified* quickly (where "quickly" means "in polynomial time"), can the answers also be *computed* quickly?

Consider, for instance, the subset-sum problem, an example of a problem which is easy to verify, but whose answer is *believed* (but not proven) to be difficult to compute. Given a set of integers, does some nonempty subset of them sum to 0? For instance, does a subset of the set {−2, −3, 15, 14, 7, −10} add up to 0? The answer is YES, though it may take a while to find a subset that does, depending on its size. On the other hand, if someone claims that the answer is "YES, because {−2, −3, −10, 15} add up to zero", then we can quickly check that with a few additions. Verifying that the subset adds up to zero is much faster than finding the subset in the first place. The information needed to verify a positive answer is also called a *certificate*. So we conclude that given the right certificates, positive answers to our problem can be verified quickly (in polynomial time) and that's why this problem is in **NP**.

An answer to the **P**=**NP** question would determine whether problems like SUBSET-SUM are as easy to compute as to verify. If it turned out **P** does not equal **NP**, it would mean that some **NP** problems would be substantially harder to compute than to verify. The answer would apply to all such problems, not just the specific example of SUBSET-SUM.

The restriction to YES/NO problems doesn't really make a difference; even if we allow more complicated answers, the resulting problem (whether **FP** = **FNP**) is equivalent.

Context of the problem

The relation between the **complexity classes** **P and NP** is studied in computational complexity theory, the part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps it takes to solve a problem) and space (how much memory it takes to solve a problem).

In such analysis, a model of the computer for which time must be analyzed is required. Typically, such models assume that the computer is deterministic (given the computer's present state and any inputs, there is only one possible action that the computer might take) and *sequential* (it performs actions one after the other). These assumptions reflect the behavior of all practical computers yet devised, even including machines featuring parallel computing.

In this theory, the class **P** consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; the class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine. Arguably, the biggest open question in theoretical computer science concerns the relationship between those two classes:

> Is **P** equal to **NP**?

In a 2002 poll of 100 researchers, 61 believed the answer is no, 9 believed the answer is yes, 22 were unsure, and 8 believed the question may be independent of the currently accepted axioms, and so impossible to prove or disprove.

Formal definitions

More precisely, a *decision problem* is a problem that takes as input some string and requires as output either YES or NO. If there is an algorithm (say a Turing machine, or a Lisp or Pascal program with unbounded memory) which is able to produce the correct answer for any input string of length *n* in at most steps, where *k* and *c* are some constants independent of the input string, then we say that the problem can be solved in *polynomial time* and we place it in the class **P**. Intuitively, we think of the problems in **P** as those that can be solved reasonably quickly.

Now suppose there is an algorithm A(*w*,*C*) which takes two arguments, a string *w* which is an input string to our decision problem, and a string *C* which is a "proposed certificate", and such that A produces a YES/NO answer in at most steps (where *n* is the length of *w* and neither *c* nor *k* depend on *w*). Suppose furthermore that: *w* is a YES instance of the decision problem if and only if there exists *C* such that A(*w*,*C*) returns YES. Then we say that the problem can be solved in *non-deterministic polynomial time* and we place it in the class **NP**. We think of the algorithm A as a verifier of proposed certificates which runs reasonably fast. (Note that the abbreviation **NP** stands for "**N**on-deterministic **P**olynomial" and *not* for "**N**on-**P**olynomial".)

NP-complete

To attack the **P** = **NP** question, the concept of **NP**-completeness is very useful. Informally, the **NP**-complete problems are the "toughest" problems in **NP** in the sense that they are the ones most likely not to be in **P**. **NP**-hard problems are those to which *any* problem in **NP** can be reduced in polynomial time. **NP**-complete problems are those **NP**-hard problems which are in **NP**. For instance, the decision problem version of the traveling salesman problem is **NP**-complete. So *any* instance of *any* problem in **NP** can be transformed mechanically into an instance of the traveling salesman problem, in polynomial time. So, if the traveling salesman problem turned out to be in **P**, then **P** = **NP**. The traveling salesman problem is one of many such **NP**-complete problems. If any **NP**-complete problem is in **P**, then it would follow that **P** = **NP**. Unfortunately, many important problems have been shown to be **NP**-complete and not a single fast algorithm for any of them is known.

It is relatively obvious that the class NP-complete is non-empty because a trivial NP and NP-hard decision problem called DUH can be formulated: given a description of a Turing machine M guaranteed to halt in polynomial time, DUH is the question of whether there exists a polynomial-size input that M will accept.[1] However, since DUH is contrived and of primarily theoretical interest, it came as a breakthrough to discover that numerous existing, highly practical problems were also NP-complete.

The first natural problem proven to be NP-complete was the boolean satisfiability problem. This result was proven by Stephen Cook in 1971, and came to be known as Cook's theorem. Cook's proof that satisfiability is NP-complete is very complicated. However, after this problem was proved to be NP-complete, proof by reduction has provided a simpler way to show that many other problems are in this class. Thus, a vast class of seemingly unrelated problems are all reducible to one another, and are in a sense the "same problem" -- a profound and unexpected result.

Still harder problems

Although it is unknown whether **P** = **NP**, problems outside of **P** are known. A number of succinct problems, that is, problems which operate not on normal input but on a computational description of the input, are known to be **EXPTIME-complete**. Because it can be shown that P EXPTIME, these problems are outside **P**, and so require more than polynomial time. In fact, by the time hierarchy theorem, they cannot be solved in significantly less than exponential time.

The problem of deciding the truth of a statement in Presburger arithmetic is even harder. Fischer and Rabin proved in 1974 that every algorithm which decides the truth of Presburger statements has a runtime of at least for some constant *c*. Here, *n* is the length of the Presburger statement. Hence, the problem is known to need more than exponential run time. Even more difficult are the undecidable problems, such as the halting problem. They cannot be solved in general given any amount of time.

Is P really practical?

All of the above discussion has assumed that **P** means "easy" and "not in **P**" means "hard". While this is a common and reasonably accurate assumption in complexity theory, it is not always true in practice, for several reasons:

- It ignores constant factors. A problem that takes time $10^{1000}n$ is in **P** (it is linear time), but is completely impractical. A problem that takes time $(1+10^{-10000})^n$ is not in **P** (it is exponential time), but is very practical for values of n up into the thousands.

- It ignores the size of the exponents. A problem with time n^{1000} is in **P**, yet impractical. Problems have been proven to exist in **P** that require arbitrarily large exponents (see time hierarchy theorem). A problem with time $2^{n/1000}$ is not in **P**, yet is practical for *n* up into the thousands.
- It only considers worst-case times. There might be a problem that arises in the real world such that most of the time, it can be solved in time *n*, but on very rare occasions you'll see an instance of the problem that takes time 2^n. This problem might have an average time that is polynomial, but the worst case is exponential, so the problem wouldn't be in **P**.
- It only considers deterministic solutions. Imagine a problem that you can solve quickly if you accept a tiny error probability, but a guaranteed correct answer is much harder to get. The problem would not belong to **P** even though in practice it can be solved quickly. This is in fact a common approach to attack problems in **NP** not known to be in **P** (see **RP**, **BPP**). Even if **P**=**BPP**, as many researchers believe, it is often considerably easier to find probabilistic algorithms.
- New computing models such as quantum computers may be able to quickly solve some problems not known to be in **P**; however, none of the problems they are known to be able to solve are **NP**-hard. However, the *definition* of **P** and **NP** are in terms of classical computing models like Turing machines. Therefore, even if a quantum computer algorithm were discovered to efficiently solve an **NP**-hard problem, we would only have a way of physically solving difficult problems quickly, not a proof that the mathematical classes **P** and **NP** are equal.

Why do many computer scientists think P ≠ NP?

Most computer scientists believe that **P**≠**NP**. A key reason for this belief is that after decades of studying these problems, no one has been able to find a polynomial-time algorithm for any **NP-hard** problem. Moreover, these algorithms were sought long before the concept of **NP-complete**ness was even known (Karp's 21 NP-complete problems, among the first found, were all well-known existing problems). Furthermore, the result **P** = **NP** would imply many other startling results that are currently believed to be false, such as **NP** = **co-NP** and **P** = **PH**.

It is also intuitively argued that the existence of problems that are hard to solve but for which the solutions are easy to verify matches real-world experience.

On the other hand, some researchers believe that we are overconfident in **P** ≠ **NP** and should explore proofs of **P** = **NP** as well. For example, in 2002 these statements were made: [2]

—Moshe Vardi, Rice University

Be
in
g
att

—Anil Nerode, Cornell University

Consequences of proof

One of the reasons why the problem attracts so much attention are the consequences of the answer.

A proof of **P** = **NP** could have stunning practical consequences, if the proof leads to efficient methods for solving some of the important problems in NP. Various NP-complete problems are fundamental in many fields. There are enormous positive consequences that would follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are NP-complete, such as some types of integer programming, and the travelling salesman problem, to name two of the most famous examples. Efficient solutions to these problems would have enormous implications for logistics. But it is by no means the only field that would be profoundly changed. To take one of very many examples, important problems in Protein structure prediction are NP-complete [3]; if these problems were solvable efficiently it could spur considerable advances in biology.

But such changes may pale in significance compared to the revolution an efficient method for solving NP-complete problems would cause in mathematics itself. According to Stephen Cook [4] (PDF),

> ...it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of a reasonable length, since formal proofs can easily be recognized in polynomial time. Example problems may well include all of the CMI prize problems.

Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated - for instance, Fermat's Last Theorem took over three centuries to prove. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle and transform mathematics into a search for useful things to prove.

A proof that showed that **P** ≠ **NP**, while lacking the practical computational benefits of a proof that **P** = **NP**, would represent a massive advance in computational complexity theory and provide guidance for future research. It would allow one to show in a formal way that many common problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems. Due to widespread belief in **P** ≠ **NP**, much of this focusing of research has already taken place.

Results about difficulty of proof

A million-dollar prize and a huge amount of dedicated research with no substantial results are enough to show the problem is difficult. There have also been some formal results demonstrating why the problem might be difficult to solve.

One of the most frequently-cited is a result involving oracles. Imagine you have a magical machine called an *oracle* that can solve only one problem, such as determining if a given number is prime, but can solve it in constant time. Our new question is now, if we're allowed to use this oracle as much as we want, are there problems we can verify in polynomial time that we can't solve in polynomial time? It turns out that, depending on the problem that the oracle solves, with certain oracles one has **P** = **NP**, while for other oracles one has **P** ≠ **NP**. The practical consequence of this is that any proof which can be modified to account for the existence of these oracles cannot solve the problem. Unfortunately, most known methods and nearly all classical methods can be modified in such a way (we say they are *relativizing*).

Furthermore, a 1993 result by Alexander Razborov and Steven Rudich showed that, given a certain credible assumption, proofs that are "natural" in a certain sense cannot solve the **P** = **NP** problem (see natural proof). This demonstrated that some of the most seemingly-promising methods of the time were also unlikely to succeed. As more theorems of this kind are proved, a potential proof of the theorem has more and more traps to avoid.

This is actually another reason why **NP-complete** problems are useful: if a polynomial-time algorithm can be demonstrated for an **NP-complete** problem, this would solve the **P** = **NP** problem in a way which is not excluded by the above results.

Polynomial-time algorithms

No one knows whether polynomial-time algorithms exist for **NP-complete** languages. But if such algorithms do exist, we already know some of them! For example, the following algorithm correctly accepts an **NP-complete** language, but no one knows how long it takes in general. This is a polynomial-time algorithm if and only if **P** = **NP**.

```
// Algorithm that accepts the NP-complete language SUBSET-SUM.
//
// This is a polynomial-time algorithm if and only if P=NP.
//
// "Polynomial-time" means it returns "YES" in polynomial time when
```

```
// the answer should be "YES", and runs forever when it's "NO".
//
// Input:  S = a finite set of integers
// Output: "YES" if any subset of S adds up to 0.
//         Otherwise, it runs forever with no output.
// Note:  "Program number P" is the program you get by
//         writing the integer P in binary, then
//         considering that string of bits to be a
//         program.  Every possible program can be
//         generated this way, though most do nothing
//                                   because    of    syntax    errors.

FOR N = 1...infinity
    FOR P = 1...N
        Run program number P for N steps with input S
        IF the program outputs a list of distinct integers
            AND the integers are all in S
            AND the integers sum to 0

        THEN
            OUTPUT "YES" and HALT
```

If **P** = **NP**, then this is a polynomial-time algorithm accepting an **NP-Complete** language. "Accepting" means it gives "YES" answers in polynomial time, but is allowed to run forever when the answer is "NO".

Perhaps we want to "solve" the SUBSET-SUM problem, rather than just "accept" the SUBSET-SUM language. That means we want it to always halt and return a "YES" or "NO" answer. Does any algorithm exist that can provably do this in polynomial time? No one knows. But if such algorithms do exist, then we already know some of them! Just replace the IF statement in the above algorithm with this:

```
        IF the program outputs a complete math proof
            AND each step of the proof is legal
            AND the conclusion is that S does (or does not) have a
subset summing to 0
        THEN
            OUTPUT "YES" (or "NO" if that were proved) and HALT
```

Logical characterizations

The **P**=**NP** problem can be restated in terms of the expressibility of certain classes of logical statements. All languages in **P** can be expressed in first-order logic with the addition of a least fixed point operator (effectively, this allows the definition of recursive functions). Similarly, **NP** is the set of languages expressible in existential second-order logic — that is, second-order logic restricted to exclude universal quantification over relations, functions, and subsets. The languages in the polynomial hierarchy, **PH**, correspond to all of second-order logic. Thus, the question "is **P** a proper subset of **NP**" can be reformulated as "is existential second-order logic able to describe languages that first-order logic with least fixed point cannot?"

Humor and cultural references

The Princeton University computer science building has the question "**P**=**NP**?" encoded in binary in its brickwork on the top floor of the west side. If it is proven that **P**=**NP**, the bricks can easily be changed to encode "**P**=**NP**!". If **P** does not equal **NP**, it can be changed to "**P**<**NP**!". [5]

Hubert Chen, PhD, of Cornell University offers this tongue-in-cheek proof that **P** does not equal **NP**: "*Proof by contradiction. Assume **P** = **NP**. Let y be a proof that **P** = **NP**. The proof y can be verified in polynomial time by a competent computer scientist, the existence of which we assert. However, since **P** = **NP**, the proof y can be generated in polynomial time by such computer scientists. Since this generation has not yet occurred (despite attempts by such computer scientists to produce a proof), we have a contradiction.*"[6]

In the science fiction story *Antibodies* by Charles Stross (which appears in his collection "Toast"), the discovery that **P** = **NP** quickly leads to the emergence of Artificial Intelligence bent on enslaving humanity.

The **P** = **NP** problem has also been featured in television:

- In a scene of *The Simpsons* entitled "Homer^3" (part of the Treehouse of Horror VI episode), Homer enters the third dimension where "**P** = **NP**" appears as a hovering equation in this bizarre parallel universe.

- In an episode of *Futurama*, Fry and Amy spend a moment in a supply closet, in which there are two separate binders labelled "**P**" and "**NP**".

- In the second episode of the CBS show *NUMB3RS*, Charlie, a mathematician, works with his brother, an FBI agent, to predict which bank a group of seemingly non-violent robbers will hit next. When FBI's attempt to arrest the criminals ends in bloodshed, Charlie tries to deal with his emotional reaction by attempting to solve **P** = **NP**. (The show used the popular computer game Minesweeper to help explain what he was working on.)

References

- A. S. Fraenkel and D. Lichtenstein, Computing a perfect strategy for n*n chess requires time exponential in n, Proc. 8th Int. Coll. *Automata, Languages, and Programming*, Springer LNCS 115 (1981) 278-293 and *J. Comb. Th. A* 31 (1981) 199-214.
- E. Berlekamp and D. Wolfe, Mathematical Go: Chilling Gets the Last Point, A. K. Peters, 1994. D. Wolfe, Go endgames are hard, MSRI Combinatorial Game Theory Research Worksh., 2000.
- Neil Immerman. Languages Which Capture Complexity Classes. *15th ACM STOC Symposium*, pp.347-354. 1983.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). "Chapter 34: NP-Completeness", *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, pp.966–1021. ISBN 0-262-03293-7.
- Christos Papadimitriou (1993). "Chapter 14: On P vs. NP", *Computational Complexity*, 1st edition, Addison Wesley, pp.329–356. ISBN 0-201-53082-1.

1. ^ R. E. Ladner "On the structure of polynomial time reducibility," J.ACM, 22, pp. 151–171, 1975. Corollary 1.1. ACM site.