What's “P=NP?”, and why is it such a famous question? [closed]

The question of whether P=NP is perhaps the most famous in all of Computer Science. What does it mean? And why is it so interesting?

Oh, and for extra credit, please post a proof of the statement's truth or falsehood. :)

computer-science theory complexity-theory np-complete p-np

edited Feb 18 '10 at 11:45 tanascius 28.4k 12 73 106

asked Sep 21 '08 at 16:07 raldi 6,215 19 53 77

2 As nicely laid out by Scott Aaronson, MIT "If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss..." excerpt from en.wikipedia.org/wiki/Complexity_classes_P_and_NP . – gts Jul 31 '12 at 22:10

See also In basic terms, what is the definition of P, NP, NP-Complete, and NP-Hard? on Computer Science. – Kaveh Jun 9 '14 at 21:17

7 Answers

P stands for polynomial time. NP stands for non-deterministic polynomial time.

Polynomial time means that the complexity of the algorithm is O(n^k), where n is the size of your data (e. g. number of elements in a list to be sorted), and k is a constant. Complexity is time measured in the number of operations it would take, as a function of the number of data items. And an operation is whatever makes sense as a basic operation for a particular task. For sorting the basic operation is a comparison. For matrix multiplication the basic operation is multiplication of two numbers.

Now the question is, what does deterministic vs. non-deterministic mean. There is an abstract computational model, an imaginary computer called a Turing machine (TM). This machine has a finite number of states, and an infinite tape, which has discrete cells into which a finite set of symbols can be written and read. At any given time, the TM is in one of its states, and it is looking at a particular cell on the tape. Depending on what it reads from that cell, it can write a new symbol into that cell, move the tape one cell forward or backward, and go into a different state. This is called a state transition. Amazingly enough, by carefully constructing states and transitions, you can design a TM, which is equivalent to any computer program that can be written. This is why it is used as a theoretical model for proving things about what computers can and cannot do.

There are two kinds of TM's that concern us here: deterministic and non-deterministic. A deterministic TM only has one transition from each state for each symbol that it is reading off the tape. A non-deterministic TM may have several such transition, i. e. it is able to check several possibilities simultaneously. This is sort of like spawning multiple threads. The difference is that a non-deterministic TM can spawn as many such "threads" as it wants, while on a real computers only a specific number of threads can be executed at a time (equal to the number of CPUs). In reality, computers are basically deterministic TMs with finite tapes. On the other hand, a non-deterministic TM cannot be physically realized, except maybe with a quantum computer.

It has been proven that any problem that can be solved by a non-deterministic TM can be solved by a deterministic TM. However, it is not clear how much time it will take. The statement P=NP means that if a problem takes polynomial time on a non-deterministic TM, then one can build a deterministic TM which would solve the same problem also in polynomial time. So far nobody have been able to show that it can be done, but nobody has been able to prove that it cannot be done, either.

NP-complete problem means an NP problem X, such that any NP problem Y can be reduced to X by a polynomial reduction. That implies that if anyone ever comes up with a polynomial-time solution to an NP-complete problem, that will also give a polynomial-time solution to any NP problem. Thus that would prove that P=NP. Conversely, if anyone were to prove that P!=NP, then we would be certain that there is no way to solve an NP problem in polynomial time on a conventional computer.

An example of an NP-complete problem is the problem of finding a truth assignment that would make a boolean expression containing n variables true.
For the moment in practice any problem that takes polynomial time on the non-deterministic TM can only be done in exponential time on a deterministic TM or on a conventional computer.
For example, the only way to solve the truth assignment problem is to try 2^n possibilities.

edited Feb 19 '11 at 13:42 answered Sep 24 '08 at 15:20
Dima
25.2k 10 45 90

3 It isn't true that the only way to solve SAT is enumeration of cases. See en.wikipedia.org/wiki/… for info on the DPLL algorithm, which is actually very efficient in many common cases. – Doug McClean Sep 24 '08 at 16:40

1 Okay, you made this way more complicated by bringing in non-deterministic Turing machines. That's totally a peripheral issue. – Derek Park Sep 24 '08 at 16:46

28 Derek, I beg to disagree. I really don't see how you explain P and NP without Turing machines. I was once in an algorithms class, which tried that. If I didn't know about TM's, I'd be totally lost. – Dima Sep 24 '08 at 18:20

2 It's true *in practice* that solving NP-complete problems takes greater than polynomial time on a real computer, but that's not what it means, it's just the current state of the art, as a consequence of the fact that P=NP is unknown. If anyone found a polynomial algorithm to solve any NP-complete problem, that would prove P=NP, and we know that hasn't happened because it would be in the news! Conversely if it was proved that P!=NP, then we could confidently say that no NP-complete problem is solvable in polynomial time. – Steve Jessop Feb 19 '11 at 10:45

7 I know this is quite old, but I just wanna say that the answer is epic and it's the first that clicked for me ! Good job – Dimitar Dimitrov Aug 17 '13 at 19:51

1. A yes-or-no problem is in **P** (**P**oynomial time) if the answer can be computed in polynomial time.
2. A yes-or-no problem is in **NP** (**N**on-deterministic **P**oynomial time) if a yes answer can be *verified* in polynomial time.

Intuitively, we can see that if a problem is in **P**, then it is in **NP**. Given a potential answer for a problem in **P**, we can verify the answer by simply recalculating the answer.

Less obvious, and much more difficult to answer, is whether all problems in **NP** are in **P**. Does the fact that we can verify an answer in polynomial time mean that we can compute that answer in polynomial time?

There are a large number of important problems that are known to be **NP**-complete (basically, if any these problems are proven to be in **P**, then *all* **NP** problems are proven to be in **P**). If **P** = **NP**, then all of these problems will be proven to have an efficient (polynomial time) solution.

Most scientists believe that **P**!=**NP**. However, no proof has yet been established for either **P** = **NP** or **P**!=**NP**. If anyone provides a proof for either conjecture, they will win $1MM.

edited Apr 9 '13 at 21:13 answered Sep 24 '08 at 17:03
Jeremy S. Derek Park
2,318 6 29 52 **29.4k** 7 41 68

To give the simplest answer I can think of:

Suppose we have a problem that takes a certain number of inputs, and has various potential solutions, which may or may not solve the problem for given inputs. A logic puzzle in a puzzle magazine would be a good example: the inputs are the conditions ("George doesn't live in the blue or green house"), and the potential solution is a list of statements ("George lives in the yellow house, grows peas, and owns the dog"). A famous example is the Traveling Salesman problem: given a list of cities, and the times to get from any city to any other, and a time limit, a potential solution would be a list of cities in the order the salesman visits them, and it would work if the sum of the travel times was less than the time limit.

Such a problem is in NP if we can efficiently check a potential solution to see if it works. For example, given a list of cities for the salesman to visit in order, we can add up the times for each trip between cities, and easily see if it's under the time limit. A problem is in P if we can efficiently find a solution if one exists.

(Efficiently, here, has a precise mathematical meaning. Practically, it means that large problems aren't unreasonably difficult to solve. When searching for a possible solution, an inefficient way would be to list all possible potential solutions, or something close to that, while an efficient way would require searching a much more limited set.)

Therefore, the P=NP problem can be expressed this way: If you can verify a solution for a problem of the sort described above efficiently, can you find a solution (or prove there is none) efficiently? The obvious answer is "Why should you be able to?", and that's pretty much where the matter stands today. Nobody has been able to prove it one way or another, and that bothers a lot of mathematicians and computer scientists. That's why anybody who can prove the solution is up for a million dollars from the Claypool Foundation.

We generally assume that P does not equal NP, that there is no general way to find solutions. If it turned out that P=NP, a lot of things would change. For example, cryptography would become impossible, and with it any sort of privacy or verifiability on the Internet. After all, we can efficiently take the encrypted text and the key and produce the original text, so if P=NP we could efficiently find the key without knowing it beforehand. Password cracking would become trivial. On the other hand, there's whole classes of planning problems and resource allocation problems that we could solve effectively.

You may have heard the description NP-complete. An NP-complete problem is one that is NP (of course), and has this interesting property: if it is in P, every NP problem is, and so P=NP. If you could find a way to efficiently solve the Traveling Salesman problem, or logic puzzles from puzzle magazines, you could efficiently solve anything in NP. An NP-complete problem is, in a way, the hardest sort of NP problem.

So, if you can find an efficient general solution technique for any NP-complete problem, or prove that no such exists, fame and fortune are yours.

answered Sep 24 '08 at 16:26
David Thornley
41.8k 8 62 125

In your second last paragraph you have "in a way, the hardest sort". You should say NP-complete are the hardest since they are NP-hard. – grom Nov 24 '08 at 5:14

I'm not sure that fortune would be yours. The government might want your head. – Millie Smith Nov 5 '14 at 3:30

A short summary from my humble knowledge:

There are some easy computational problems (like finding the shortest path between two points in a graph), which can be calculated pretty fast (O(n^k), where n is the size of the input and k is a constant (in case of graphs, it's the number of vertexes or edges)).

Other problems, like finding a path that crosses every vertex in a graph or getting the RSA private key from the public key is harder (O(e^n)).

But CS speak tells that the problem is that we cannot 'convert' a non-deterministic Turing-machine to a deterministic one, we can, however, transform non-deterministic finine automatons (like the regex parser) into deterministic ones (well, you can, but the run-time of the machine will take long). That is, we have to try every possible path (usually smart CS professors can exclude a few ones).

It's interresting, because nobody even has any idea of the sollution. Some say it's true, some say it's false, but there is no consensus. Another interresting thing is that a sollution would be harmfull for public/private key encriptions (like RSA). You could break them as easily as generating an RSA key is now.

And it's a pretty inspiring problem.

edited Jun 22 '13 at 20:08 answered Sep 21 '08 at 16:14

dansalmo
4,560 3 18 34

terminus
4,687 4 18 35

1 That's not quite true - you can convert a NDTM to a DTM, but the new machine has a running time exponential in the running time of the original (you effectively breadth first search the state transition graph of the NDTM). – Adam Wright Sep 24 '08 at 15:25

Ok, I'll update the answer. – terminus Sep 24 '08 at 15:49

http://en.wikipedia.org/wiki/Complexity_classes_P_and_NP

answered Sep 21 '08 at 16:14

community wiki
Michael Schubart

There is not much I can add to the what and why of the P=?NP part of the question, but in regards to the proof. Not only would a proof be worth some extra credit, but it would solve one of the Millennium Problems. An interesting poll was recently conducted and the published results (PDF) are definitely worth reading in regards to the subject of a proof.

edited Sep 25 '08 at 15:51
Derek Park
29.4k 7 41 68

answered Sep 24 '08 at 15:26
rjzii
9,656 7 58 93

First, some definitions:

- A particular problem is in P if you can compute a solution in time less than `n^k` for some `k`, where `n` is the size of the input. For instance, sorting can be done in `n log n` which is less than `n^2`, so sorting is polynomial time.
- A problem is in NP if there exists a `k` such that there exists a solution of size at most `n^k` which you can verify in time at most `n^k`. Take 3-coloring of graphs: given a graph, a 3-coloring is a list of (vertex, color) pairs which has size `O(n)` and you can verify in time `O(m)` (or `O(n^2)`) whether all neighbors have different colors. So a graph is 3-colorable only if there is a short and readily verifiable solution.

An equivalent definition of NP is "problems solvable by a *N*ondeterministic Turing machine in *P*olynomial time". While that tells you where the name comes from, it doesn't give you the same intuitive feel of what NP problems are like.

Note that P is a subset of NP: if you can find a solution in polynomial time, there is a solution which can be verified in polynomial time--just check that the given solution is equal to the one you can find.

Why is the question `P =? NP` interesting? To answer that, one first needs to see what NP-complete problems are. Put simply,

- A problem L is NP-complete if (1) L is in P, and (2) an algorithm which solves L can be used to solve any problem L' in NP; that is, given an instance of L' you can create an instance of L that has a solution if and only if the instance of L' has a solution. Formally speaking, every problem L' in NP is *reducible* to L.

Note that the instance of L must be polynomial-time computable and have polynomial size, in the size of L'; that way, solving an NP-complete problem in polynomial time gives us a polynomial time solution to *all* NP problems.

Here's an example: suppose we know that 3-coloring of graphs is an NP-hard problem. We want to prove that deciding the satisfiability of boolean formulas is an NP-hard problem as well.

For each vertex v, have two boolean variables v_h and v_l, and the requirement (v_h or v_l): each pair can only have the values {01, 10, 11}, which we can think of as color 1, 2 and 3.

For each edge (u, v), have the requirement that (u_h, u_l) != (v_h, v_l). That is,

> `not ((u_h and not u_l) and (v_h and not v_l) or ...)` enumerating all the equal configurations and stipulation that neither of them are the case.

`AND` 'ing together all these constraints gives a boolean formula which has polynomial size (`O(n+m)`). You can check that it takes polynomial time to compute as well: you're doing straightforward `O(1)` stuff per vertex and per edge.

If you can solve the boolean formula I've made, then you can also solve graph coloring: for each pair of variables v_h and v_l, let the color of v be the one matching the values of those variables. By construction of the formula, neighbors won't have equal colors.

Hence, if 3-coloring of graphs is NP-complete, so is boolean-formula-satisfiability.

We know that 3-coloring of graphs is NP-complete; however, historically we have come to know that by first showing the NP-completeness of boolean-circuit-satisfiability, and then reducing that to 3-colorability (instead of the other way around).

answered Feb 11 '09 at 7:26
Jonas Kölker
4,914 27 44