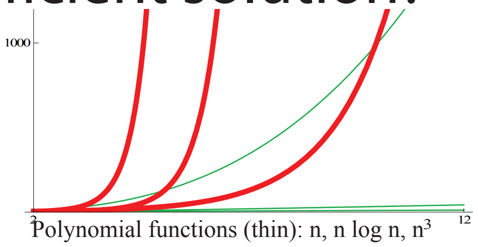
Can you tell if a problem will have an efficient solution?

If you can think of a polynomail time algorithm to solve the programming problem you are working on, then you know the problem is in P. But what if you can't think of one? Then it might be worth considering if the problem might be NP-Complete.

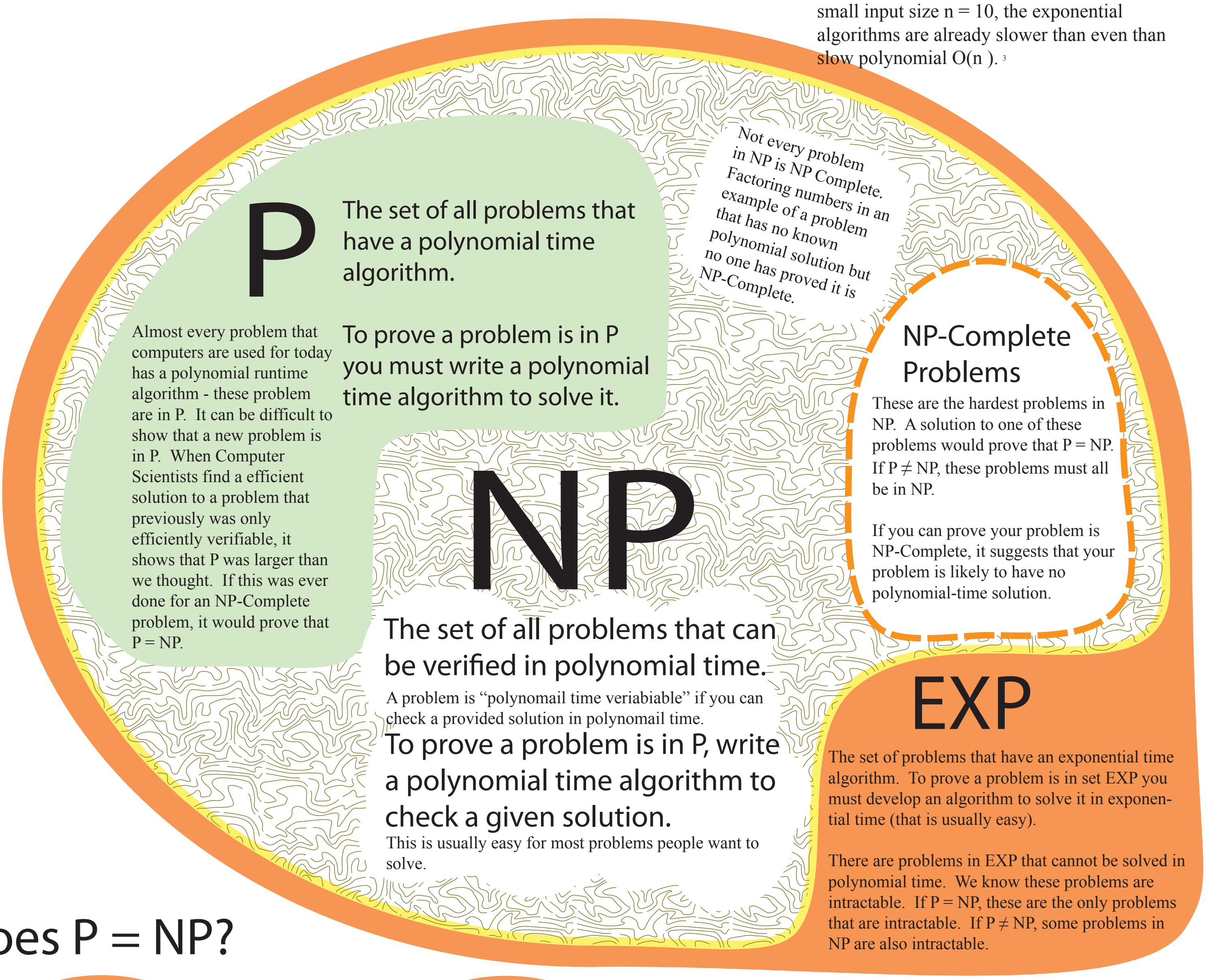


Polynomial functions (thin): n, n log n, n^3
Exponential function (thick): 2^n, n^n, n!

This graph illustrates why polynomial runtime is necessary for a solution to be practical. At small input size n = 10, the exponential algorithms are already slower than even than slow polynomial O(n^3).

Complexity Classes P, NP, and EXP

In theoretical Computer Science, problems are classified by existence of algorithms that solve them. We consider the issue of polynomial verses exponential runtime. Polynomial time algorithms (e.g. O(n) O(n log n) O(n3)) are the fastest. Exponential time algorithms are too slow to run on even moderate size data sets.



P

The set of all problems that have a polynomial time algorithm.

To prove a problem is in P you must write a polynomial time algorithm to solve it.

Almost every problem that computers are used for today has a polynomial runtime algorithm - these problem are in P. It can be difficult to show that a new problem is in P. When Computer Scientists find a efficient solution to a problem that previously was only efficiently verifiable, it shows that P was larger than we thought. If this was ever done for an NP-Complete problem, it would prove that P = NP.

Not every problem in NP is NP Complete. Factoring numbers in an example of a problem that has no known polynomial solution but no one has proved it is NP-Complete.

NP-Complete Problems

These are the hardest problems in NP. A solution to one of these problems would prove that P = NP. If P ≠ NP, these problems must all be in NP.

If you can prove your problem is NP-Complete, it suggests that your problem is likely to have no polynomial-time solution.

NP

The set of all problems that can be verified in polynomial time.

A problem is "polynomail time veriabiable" if you can check a provided solution in polynomail time.

To prove a problem is in P, write a polynomial time algorithm to check a given solution.

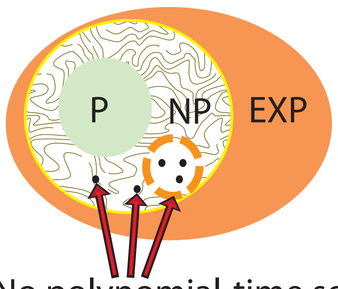
This is usually easy for most problems people want to solve.

EXP

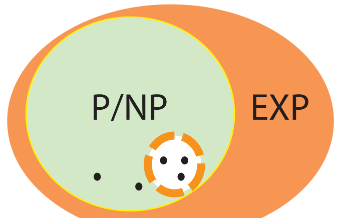
The set of problems that have an exponential time algorithm. To prove a problem is in set EXP you must develop an algorithm to solve it in exponential time (that is usually easy).

There are problems in EXP that cannot be solved in polynomial time. We know these problems are intractable. If P = NP, these are the only problems that are intractable. If P ≠ NP, some problems in NP are also intractable.

Does P = NP?



?



No polynomial-time solution?

The common wisdom is that all of the NP-Complete problems and some of the other problems in NP have no polynomial time solution. But this wisdom could be wrong.

It could be that that every problem in NP does in fact have a polynomial time solution. No one has ever proved that any polynomial time verifiable problems do not have a polynomial time solution.

This persistant question is known as the question of "Does P = NP?"

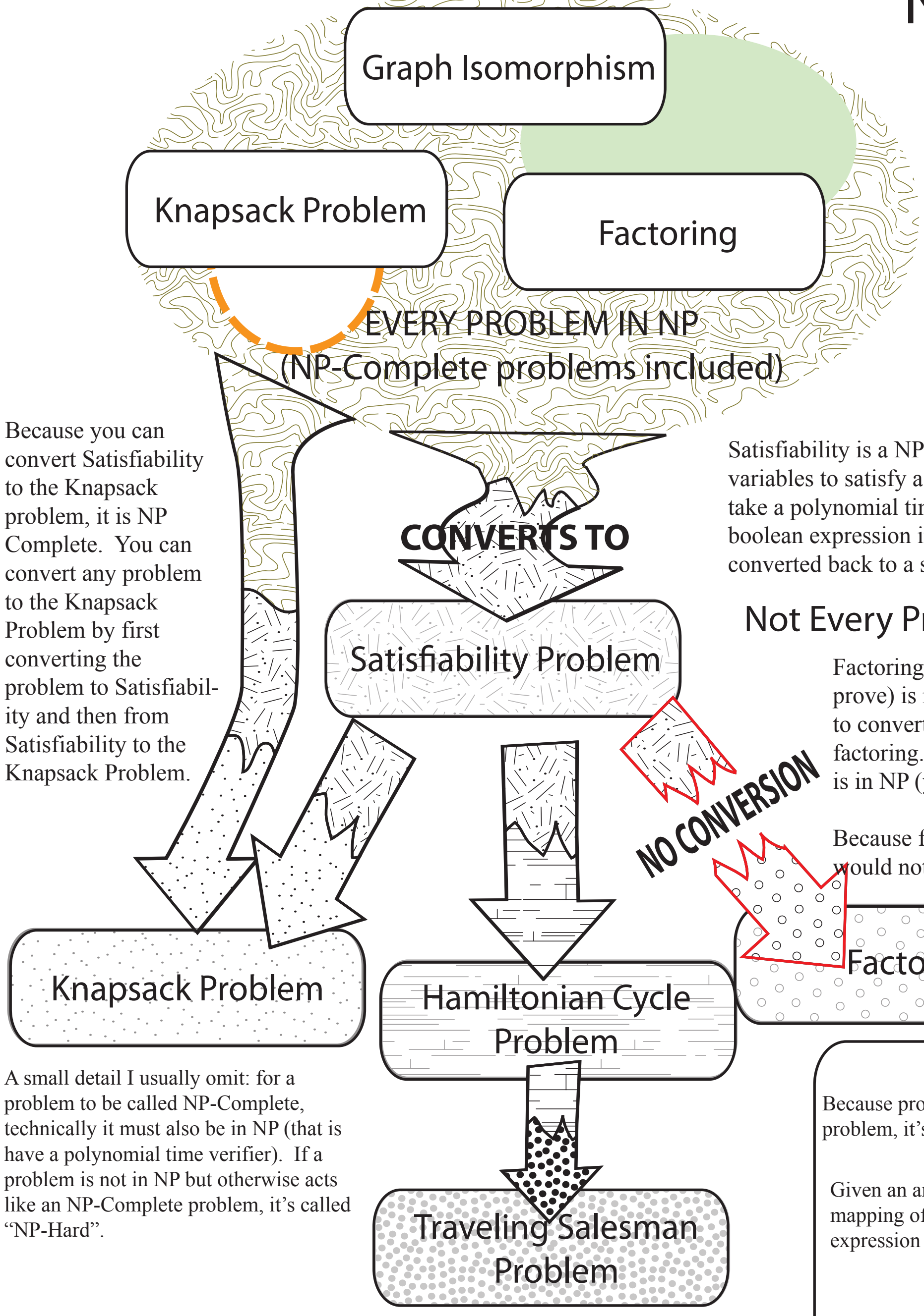
Key Questions

How do you prove a problem is in set P?
How do you prove a problem is in set NP?

What would you need to prove that P = NP?
What would you need to prove that P ≠ NP?

NP-Complete Problems

If you can convert every problem in NP to your problem in polynomial time, then your problem is NP-Complete. A polynomial time algorithm that solved an NP-Complete problem would also solve every other problem in NP.



Because you can convert Satisfiability to the Knapsack problem, it is NP Complete. You can convert any problem to the Knapsack Problem by first converting the problem to Satisfiability and then from Satisfiability to the Knapsack Problem.

A small detail I usually omit: for a problem to be called NP-Complete, technically it must also be in NP (that is have a polynomial time verifier). If a problem is not in NP but otherwise acts like an NP-Complete problem, it's called "NP-Hard".

Satisfiability

Satisfiability is a NP-Complete problem about finding mappings for true/false variables to satisfy a boolean expression. Cook's Theorm provides a algorithm to take a polynomial time verifier (which every NP problem has) and convert it to a boolean expression in polynomail time. The true/false mappings can then be converted back to a solution for the original problem in polynomial time.

Not Every Problem in NP is NP Complete

Factoring is a problem in NP that (as far has any one has been able to prove) is not NP-Complete. That means that no one has ever been able to convert Satisfiability (or any other NP-Complete problem) TO factoring. Factoring can be CONVERTED TO Satisfiability because it is in NP (you can see it there in the big oval at the top).

Because factoring is not NP-Complete, a polynomail time solution to it would not prove that P = NP.

If you can convert a NP-Complete problem TO your problem, your problem is NP Complete. This is the usual way you prove a problem is NP Complete.

It follows from the definition of NP-Completeness that any problem that an NP-Complete problem can be converted TO must be NP-Complete as well (see the text over the Knapsack problem for an example). Usually when you want to prove a problem is NP complete, you simply prove an existing known NP-Complete problem can be converted to it.

No one has ever found a polynomial time algorithm to solve an NP-Complete problem. For this reason, most Computer Scientists use "NP-Complete" as a shorthand for "no polynomial time solution".

If P ≠ NP, then no NP-Complete problem can be in P. This is not a proof that any particular NP-Complete problem has no polynomial time solution - if one is found it proves P = NP. But because no one has ever found an efficient solution to an NP-Complete problem, intution suggests that no solution may be possible. So if you can prove your problem is NP-Complete, it should be a warning that no efficient solution is likely.

Sample NP-Complete Problems

Because proofs of NP-Completeness involve converting a known NP-Complete problem, it's worthwhile to know a few.

Satisfiability

Given an arbitrary expression of variables NOTs ANDs and ORs. What is a mapping of variables to truth values (e.g. a → true, b → false) that makes the expression true? Example expresses:

a AND (b OR (NOT a))

a AND b AND ((NOT a) OR (NOT b))

Knapsack Problem

Given a set of items with different weights and values, what is the maximum value you can get in a "knapsack" with a maximum weight W?

Hamiltonian Cycle

Given a set of cities with roads between them, is there a path that visits each city exactly once and returns to the start?

Traveling Salesman Problem

Given a set of cities with roads between them, what is the path of minimum distance that visits each city exactly once and returns to the start?

Exam Scheduling Problem

Given a list of courses, a list of conflicts between them, and an integer k; is there an exam schedule consisting of k dates such that there are no conflicts between courses which have examinations on the same date?

Key Questions

If a solution were developed for an NP-Complete problem, what would that mean?

If you can convert an NP-Complete problem into your problem, what does that mean about your problem?

Using P/NP Pragmatically

Step 0: The Problem

You have some problem and the best algorithm to solve it is not obvious. Like a Big O analysis, this should be a problem in which the input size can be large.

Example: Sudoku

To solve a sudoku puzzle, you must find numbers for each of the positions so that each row column and "box" has 1 of every number 1 to 9 (or n^2 on larger boards). Though peope normally use a 3x3 board, any size is possible. A 5x5 board has 5 boxes on a side, each box contains 25 numbers.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Step 1: Try to Think of a Polynomial Time Solution Algorithm

Think of the best algorithm you can and determine its Big O. If the algorithm is polynomial, you are done (you have just proved the problem is in P). If the best algorithm you can think of its exponential, it may be worthwhile continuting.

No polynomial time algorithm to solve a sudoku puzzle is obvious. You might want to try yourself to verify this fact.

Step 2: Try to Think of a Polynomial Time Verifier Algorithm

Think of an algorithm that given a solution, verifies that solution is correct. If you can, you've just proved your problem is in NP. If you can't think of one, your problem may be very hard and not likely to have a polynomial time solution.

Note that having a way to check a solution is not the same as being able to get a solution in a reasonable time. Though you can "guess and check" those algorithms almost always take polynomial time.

Checking a soduku solution is easy. Just make sure every row has every number, every column has every number, and every "box" has every number. This can be done in polynomial time.

By way of example, the numbers 1-25 have 15511210043330985984000000 possible orders. If you check 1/nanosecond for a 5x5 Sudoku solution your program will take 490 million years.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Step 3: Try to Think of a Polynomial Time NP-Complete Conversion

To prove your problem is NP-Complete, you need an algorithm to convert an NP-Complete problem into your problem. It might be worthwhile to try and look up known NP-Complete problems in your domain to see if one is similar.

This actually has been done for Sudoku, using a reduction to Satisifability (through some other problems). The actual details though are quite complex.

Step 4: If your problem is NP-Complete, try to make it simplier

Oftentimes it is possible to find a solution to particular special cases of NP-Complete problems, or to use heuristics that generate approximate solutions.

Using these sorts of techniques you can build Sudoku solvers that will work on puzzles of any size that a human might attempt. But the best known algoritms to solve n x n Sudoku boards are all exponential.

Expanded Questions

Defining P and NP

A. If I want to prove a problem is Type P. I need to discover:
1. An efficient algorithm to solve the problem
2. A way to efficiently verify a solution that I'm given
3. A new hardness level

B. If I want to prove a problem is Type NP. I need to discover:
1. An efficient algorithm to solve the problem
2. A way to efficiently verify a solution that I'm given
3. A new hardness level

Does P = NP?

C. What would I need to show that P does not equal NP:
1. An algorithm to solve every problem in NP in polynomial time
2. A proof that at least one polynomial time verifiable problem has no polynomial time algorithm
3. A verifier that runs in exponential time
4. A proof that P is a subset of EXP

D. What would I need to show that P equals NP:
1. An algorithm to solve every problem in NP in polynomial time
2. A proof that at least one polynomial time verifiable problem has no polynomial time algorithm
3. A verifier that runs in exponential time
4. A proof that P is a subset of EXP

NP Completeness

E. Say we have a problem in NP, and we discover that we can convert this problem to a NP Complete problem. What does this tell us?
1. P does not equal NP
2. P equals NP
3. Nothing, we knew this already
4. Our problem is NP-Complete and likely quite hard

F. Say we have a problem in NP, and we discover that we can convert a known NP Complete problem to our problem. What does this tell us?
1. P does not equal NP
2. P equals NP
3. Nothing, we knew this already
4. Our problem is NP-Complete and likely quite hard

G. If a polynomial time algorithm was discovered for an NP Complete problem, it would show:
1. P does not equal NP
2. P equals NP
3. Neither 1 nor 2

H. If a polynomial time algorithm was discovered for a problem in NP (not an NP Complete one), it would show:
1. P does not equal NP
2. P equals NP
3. Neither 1 nor 2