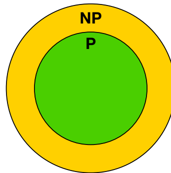
Eric Roberts
CS 54N

Handout #17
October 26, 2016

The P = NP Question

The **P** = **NP** Question



Eric Roberts
CS 54N
October 26, 2016

The **P** = **NP** Question

Clay Mathematics Institute
Dedicated to increasing and disseminating mathematical knowledge

HOME | ABOUT CMI | PROGRAMS | NEWS & EVENTS | AWARDS | SCHOLARS | PUBLICATIONS

Millennium Problems

In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) has named seven *Prize Problems*. The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. During the Millennium Meeting held on May 24, 2000 at the Collège de France, Timothy Gowers presented a lecture entitled *The Importance of Mathematics*, aimed for the general public, while John Tate and Michael Atiyah spoke on the problems. The CMI invited specialists to formulate each problem.

One hundred years earlier, on August 8, 1900, David Hilbert delivered his famous lecture about open mathematical problems at the second International Congress of Mathematicians in Paris. This influenced our decision to announce the millennium problems as the central theme of a Paris meeting.

The rules for the award of the prize have the endorsement of the CMI Scientific Advisory Board and the approval of the Directors. The members of these boards have the responsibility to preserve the nature, the integrity, and the spirit of this prize.

Paris, May 24, 2000

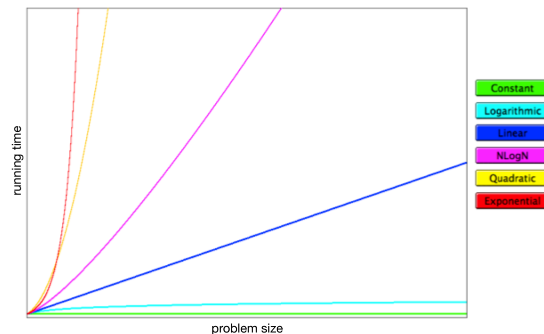
- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- P vs NP
- Poincaré Conjecture
- Riemann Hypothesis
- Yang-Mills Theory

- Rules
- Millennium Meeting Videos

Definitions

- The class **P** consists of all decision problems that can be solved in polynomial time by a deterministic Turing machine.
- The class **NP** consists of all decision problems that can be solved in polynomial time by a nondeterministic Turing machine.
  - A ***decision problem*** is one that has only a yes-or-no answer.
  - ***Polynomial time*** is a measure of computational complexity that is bounded by a polynomial.
  - A ***deterministic*** Turing machine follows only one execution path at a time.
  - A ***nondeterministic*** Turing machine can follow multiple paths in parallel.
- The **P** = **NP** question is whether these two classes are the same.

Graphs of the Complexity Classes



Recursion

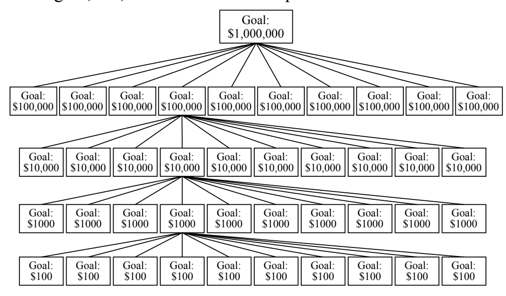
- One of the most important great ideas in computer science is the concept of ***recursion***, which is the process of solving a problem by dividing it into smaller subproblems ***of the same form***. The italicized phrase is the essential characteristic of recursion; without it, all you have is a description of stepwise refinement of the sort we teach in courses like CS 106A.
- The fact that recursive decomposition generates subproblems that have the same form as the original problem means that recursive programs will use the same function or method to solve subproblems at different levels of the solution. In terms of the structure of the code, the defining characteristic of recursion is having functions that call themselves, directly or indirectly, as the decomposition process proceeds.

A Simple Illustration of Recursion

- Suppose that you are the national fundraising director for a charitable organization and need to raise $1,000,000.
- One possible approach is to find a wealthy donor and ask for a single $1,000,000 contribution. The problem with that strategy is that individuals with the necessary combination of means and generosity are difficult to find. Donors are much more likely to make contributions in the $100 range.
- Another strategy would be to ask 10,000 friends for $100 each. Unfortunately, most of us don't have 10,000 friends.
- Recursion offers a more promising strategy. All you need to do is find ten regional coordinators and ask each one to raise $100,000. Those regional coordinators in turn delegate the task to ten local coordinators, each with a goal of $10,000, and so on until the donations can be raised individually.

A Simple Illustration of Recursion

The following diagram illustrates the recursive strategy for raising $1,000,000 described on the previous slide:



A Pseudocode Fundraising Strategy

If you were to implement the fundraising strategy in the form of a JavaScript function, it would look something like this:

```
function collectContributions(n) {
   if (n <= 100) {
      Collect the money from a single donor.
   } else {
      Find 10 volunteers.
      Get each volunteer to collect n/10 dollars.
      Combine the money raised by the volunteers.
   }
}
```

What makes this strategy recursive is that the line

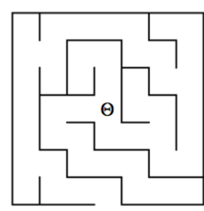
Get each volunteer to collect n/10 dollars.

will be implemented using the following recursive call:

```
collectContributions(n / 10);
```

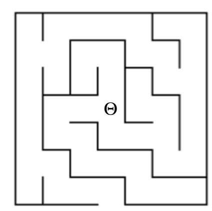
A Recursive View of Mazes

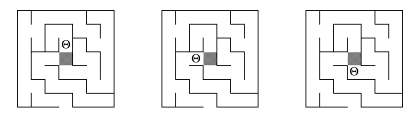
- Solving a maze algorithmically is simplest if you use recursion, but coding that solution requires you to find the right recursive insight.
- Consider the maze shown at the right. How can Theseus transform the problem into one of solving a simpler maze?
- The insight you need is that a maze is solvable only if it is possible to solve one of the simpler mazes that results from shifting the starting location to an adjacent square and taking the current square out of the maze completely.



A Recursive View of Mazes

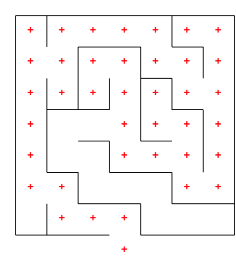
- Thus, the original maze is solvable only if one of the three mazes at the bottom of this slide is solvable.
- Each of these mazes is “simpler” because it contains fewer squares.
- The simple cases are:
  - Theseus is outside the maze
  - There are no directions left to try





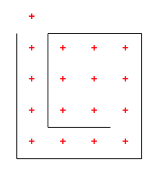
Recursion and Backtracking

- The complete recursive solution operates as follows:



Exponential Backtracking

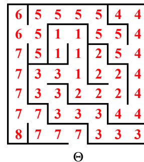
- The time required for the standard backtracking algorithm grows exponentially if there are large open areas in the maze:



- This exponential behavior is not fundamental to the maze algorithm. If the program doesn’t unmark the squares as it backtracks, the program can find the exit in linear time.

Exploiting Nondeterminism

- Another approach to solving a maze is to explore all paths concurrently as you proceed. This strategy is analogous to cloning yourself at each intersection and sending one clone down each path.

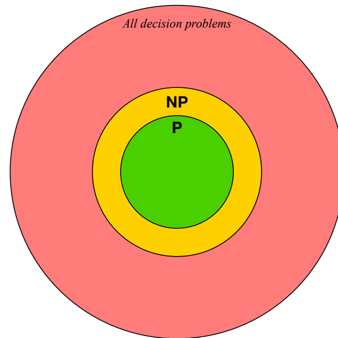


- Is this parallel strategy more efficient in the general case?

Nondeterministic Turing Machines

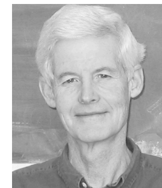
- As with the nondeterministic maze solver, a nondeterministic Turing machine can explore more than one solution strategy at once.
- In its most common formulation, a nondeterministic Turing machine is defined by allowing each instruction to transition to several new states. In effect, these multiple transitions *clone* the machine, with each of the clones continuing in a different state.
- It is conventional to define two new states: ***accept*** and ***reject***. A nondeterministic Turing Machine accepts its input if any of its cloned copies ever reaches the ***accept*** state.

Relationship between **P** and **NP**



NP-Complete Problems

- The search for an answer to the **P**=**NP** question depends on the notion of **NP**-complete problems, which was introduced by Stephen Cook in 1971. In an informal sense, a problem is **NP**-complete if it is provably as difficult to solve as any other problem in **NP**.



Stephen Cook (1939–)

- The immediate implication of this definition is that if some **NP**-complete problem can be solved in polynomial time, then ***all*** problems in **NP** can be solved in polynomial time.
- In practice, one establishes that a problem is **NP**-complete by showing that the computation of any nondeterministic Turing machine can be expressed in that domain.

Traveling Salesman Problem

- One of the classic **NP**-complete problems—noted mostly for its practical importance—is the ***Traveling Salesman Problem*** (often designated as ***TSP*** for short), which asks whether it is possible for a salesman to complete a cycle of a set of cities within some fixed cost.

BRUTE-FORCE SOLUTION: $O(n!)$

DYNAMIC PROGRAMMING ALGORITHMS: $O(n^2 2^n)$

SELLING ON EBAY: $O(1)$

STILL WORKING ON YOUR ROUTE?

SHUT THE HELL UP.

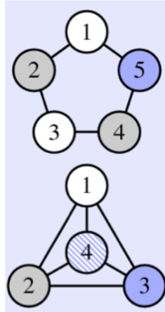
—Randall Munroe, *XKCD*

Subset Sum

Suppose that you have a set of integers called *S*. The subset-sum problem asks whether there is a subset of the elements of *S* that add up to a particular target value *t*. For example, if *S* is the set { –3, 5, 7, 10 }, the subset-sum problem when *t* is 12 returns the answer `true` because the elements in the subset { –3, 5, 10 } add up to 12. By contrast, if *t* were 11, the answer is `false` because it is impossible to choose a subset of *S* whose values adds up to 11. In his early study of NP-complete problems in 1972, Richard Karp proved that the subset-sum problem is NP-complete, although his original papers refer to the problem by a different name.

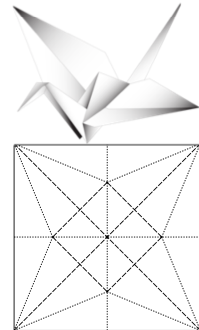
Graph Coloring

Suppose that you have a graph consisting of a set of vertices connected by a set of edges, which might, for example, look like either of the graphs to the right. The vertices of the top graph can be colored using three colors so that no two vertices connected by an edge share the same color. You could, for example, color nodes 1 and 3 white, 2 and 4 gray, and 5 blue. In the bottom graph, however, all four nodes are interconnected, which means that each must each have a different color. Deciding whether a graph can be colored with k colors is NP-complete.



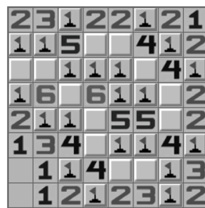
Origami Folding

The diagram at the right shows the first eight folds on the way to the creation of a classic origami crane. In some of these folds, the crease rises toward you from the paper. These are called ***mountain folds*** and appear in the diagram as dashed lines. In other folds, the crease moves away from you. These are called ***valley folds*** and appear as dotted lines. In 1996, Marshall Bern and Barry Hayes proved that deciding whether a particular pattern of mountain and valley folds will produce a flat origami figure is NP-complete.



Minesweeper

One of the most widely publicized problems in the NP-complete domain is that of determining whether a particular pattern of warning counts in the popular Microsoft Minesweeper game is consistent. In 2000, Richard Kaye published a paper proving that solving the minesweeper consistency problem is NP-complete. Because of the popularity of the game, Kaye's result was reported in newspapers and magazines throughout the world.



Satisfiability

- The problem that Steven Cook used in his proof is the *Satisfiability Problem* (commonly abbreviated as *SAT*), which asks whether any assignment of values to the variables of an expression in predicate logic makes that expression true.
- Expressions in predicate logic consist of individual *terms,* each of which can take on the value `true` or `false`, connected by *operators,* which include ∧ (and), ∨ (or), and ¬ (not). Terms appear as lowercase italic letters, such as *p, q, r,* and *s.*
- The SAT problem requires that the logical expression be in *conjunctive normal form,* in which the expression consists of individual terms, possibly preceded by ¬ (and usually written using an overbar instead of the ¬ symbol), and then combined first by the ∨ operator, and finally by the ∧ operator. It is always possible to use the rules of logic to rewrite any expression in conjunctive normal form.

Proving Satisfiability is **NP**-Complete

- The goal is to show that SAT is NP-complete, which means that a polynomial time solution to SAT implies a polynomial time solution to an arbitrary problem in NP.
- If X is an arbitrary problem in NP, that means that there must be a Turing machine M_X that solves X in time bounded by a polynomial p_X.
- The fact that the running time of M_X is bounded by p_X not only limits the number of steps M_X can execute but also puts an upper bound on how many tape squares it can reach because the tape head can move only one square per step.
- If SAT can be solved in polynomial time, it is then possible to solve X in polynomial time by taking its Turing machine M_X, transforming it into an equivalent SAT problem, and then using the polynomial-time solution of SAT to find the answer.

Constructing the SAT Expression

Step 1: Start with a Turing machine M_X and its polynomial p_X.

Step 2: Create a set of logical variables to describe the computation:

$s_{k,t}$ indicates that the machine is in state k at time t.
$p_{k,t}$ indicates that the tape head is in position k at time t.
$c_{k,t}$ indicates that the tape square k contains a 1 at time t.

Step 3: Encode the Turing machine operation as logical rules that

- Encode the initial configuration
- Ensure the machine is in exactly one state.
- Ensure the tape head is in one position.
- Restrict changes to the tape head.
- Encode all transitions of the machine.
- Guarantee that the machine ends in the *accept* state.