EASLLC 2014

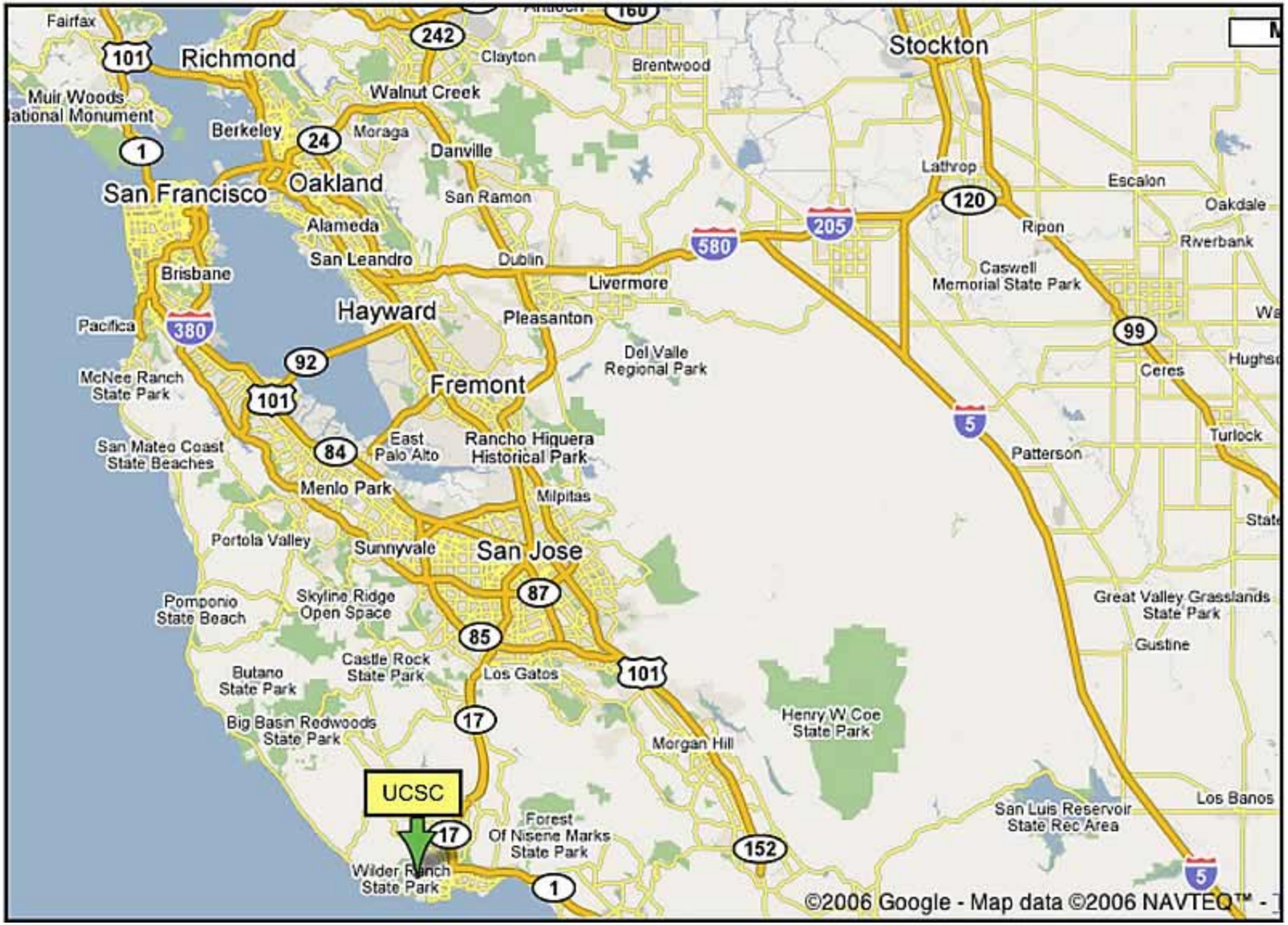
Logic and Computation

Phokion G. Kolaitis

University of California Santa Cruz

&

IBM Research - Almaden


Fairfax
101
Richmond
242
Clayton
Brentwood
Stockton
Walnut Creek
Muir Woods
National Monument
Berkeley
24
Moraga
Danville
1
San Francisco
Oakland
San Ramon
Lathrop
Escalon
120
Oakdale
Alameda
205
Ripon
Riverbank
580
San Leandro
Dublin
Brisbane
Livermore
Caswell
Memorial State Park
Hayward
Pleasanton
Pacifica
380
99
92
Del Valle
Regional Park
Ceres
McNee Ranch
State Park
Fremont
101
5
Turlock
San Mateo Coast
State Beaches
84
East
Palo Alto
Rancho Higuera
Historical Park
Patterson
Menlo Park
Milpitas
Portola Valley
Sunnyvale
San Jose
87
Great Valley Grasslands
State Park
Pomponio
State Beach
Skyline Ridge
Open Space
85
Gustine
Castle Rock
State Park
Butano
State Park
Los Gatos
101
Henry W Coe
State Park
17
Big Basin Redwoods
State Park
Morgan Hill
UCSC
Los Banos
San Luis Reservoir
State Rec Area
Forest
Of Nisene Marks
State Park
17
152
Wilder Ranch
State Park
1
5
©2006 Google - Map data ©2006 NAVTEQ™

Mathematical Logic in the 20th Century

In the 20th Century, mathematical logic was developed in an attempt to answer two questions concerning "truth" and "proof" in mathematics.

Question 1: Can we give a rigorous definition of "proof" and use it to "prove" every "true mathematical statement"?

Question 2: Is there an algorithm to decide if a given "mathematical statement" is "true"?

Note: To make sense out of these questions, careful concept formation is needed. What do we mean by:

- "mathematical statement"?
- "true mathematical statement"?
- "proof"?

Proof and Truth in Propositional Logic

Recall the two questions concerning "truth" and "proof":

Question 1: Can we give a rigorous definition of "proof" and use it to "prove" every "true mathematical statement"?

Question 2: Is there an algorithm to decide if a given "mathematical statement" is "true"?

In the context of propositional logic, we have that

- "mathematical statement" becomes
  - propositional formula
- "true mathematical statement" becomes
  - tautology (valid propositional formula).

Proof and Truth in Propositional Logic

So, the two questions about "truth" and "proof" become:

Question 1: Can we give a rigorous definition of "proof" and use it to "prove" every tautology?

Question 2: Is there an algorithm to decide if a given propositional formula is a tautology?

Algorithmic Problems in Propositional Logic

- The Satisfiability Problem:
  Given a propositional formula φ, is it satisfiable?

- The Tautology Problem:
  Given a propositional formula φ, is it a tautology?

Algorithmic Problems in Propositional Logic

- The Satisfiability Problem:
  Given a propositional formula φ, is it satisfiable?

- The Tautology Problem:
  Given a propositional formula φ, is it a tautology?

Fact: Each of these two problems can be solved via an exhaustive search algorithm that examines every truth assignment on the variables occurring in the given formulas. However, these algorithms take an exponential number of steps to terminate, namely, 2^n steps, where n is the number of variables.

Question: Are there *efficient* algorithms for these problems?

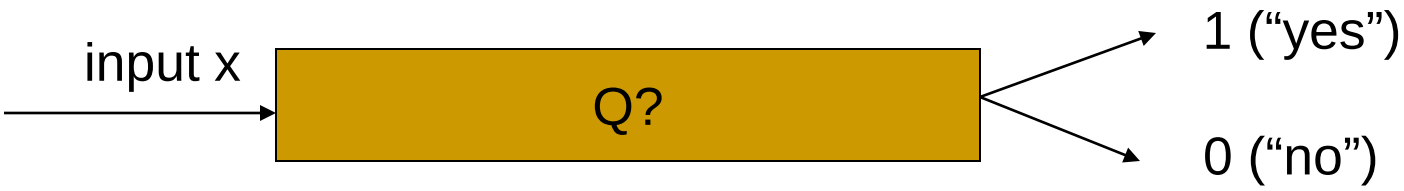
Computability and Complexity

- Before proceeding with the study of algorithmic problems in propositional logic, we will present an overview of the basic notions and results from computability theory and computational complexity.

- This overview will include a discussion of
  - Undecidable problems and the reduction method
  - Computational complexity classes and complete problems for computational complexity classes.

Decision Problems and Languages

- Definition (informal): A decision problem Q consists of a set of inputs and a question with a "yes" or "no" answer for each input.



- Definition:
  - Σ^* is the set of all strings over a finite alphabet Σ.
  - A language over Σ is a set $L \subseteq \Sigma^*$
  - Every language L gives rise to the following decision problem:
    - Given $x \in \Sigma^*$, is $x \in L$?
  - Conversely, every decision problem can be thought of as arising from a language, namely,
    the language consisting of all inputs with a "yes" answer.

Turing Computability

- Turing machines
- Turing computable (partial) functions f: $\Sigma^* \rightarrow \Sigma^*$
- Church's Thesis (aka Church-Turing Thesis): The following statements are equivalent for a (partial) function f: $\Sigma^* \rightarrow \Sigma^*$:
  - There is a Turing machine that computes f
  - There is an algorithm that computes f.
- Main Use of Church's Thesis: To show that there is no algorithm for computing a function f, it suffices to show that there is no Turing machine that computes f.

Recursive Languages

Definition: Let L $\subseteq \Sigma^*$ be a language

L is recursive if its characteristic function χ is Turing computable, where

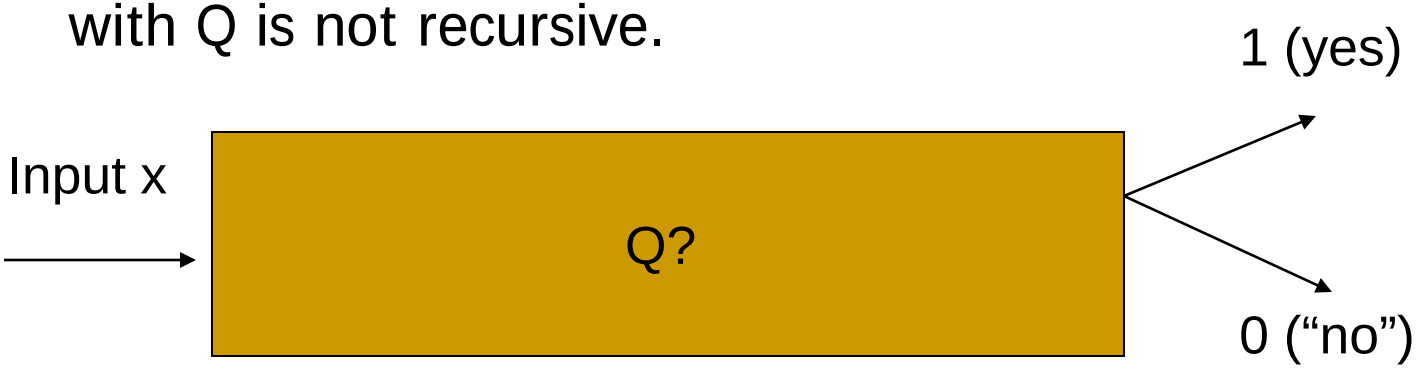
- $\chi_L(x) = 1$ if $x \in L$
- $\chi_L(x) = 0$ if $x \notin L$.

Note:

Via Church's Thesis, a language is recursive if and only if there is an algorithm that solves the membership problem for L.

Decidable and Undecidable Problems

- Definition: Let Q be a decision problem.
  - Q is decidable (solvable) if the language associated with Q is recursive.
  - Q is undecidable (unsolvable) if the language associated with Q is not recursive.



Q is undecidable means that there is **no** algorithm for this problem

Undecidable Problems

Theorem: The following problems is undecidable:

The Halting Problem (A. Turing – 1936): Given a Turing machine M and an input x, does M halt on x?

Implications of Undecidability of the Halting Problem:

- The undecidability of the Halting Problem implies that there is **no** algorithm such that, given a C program p and an input x, the algorithm determines whether the program p produces an output on input x or goes into an infinite loop.
- Of course, it may still be possible to show that a particular program terminates on a given input (or even on every input), but it is **not** possible to automate this process for every program.

Undecidable Problems

- The Halting Problem was the first fundamental decision problem shown to be undecidable.

- By now, there is a vast library of undecidable problems that includes the following two problems, variants of which are often encountererd in computer science.

  - Post's Correspondence Problem (1946)

  - Hilbert's 10^{th} Problem (1900).

The Reduction Method

- By now there is a vast library of undecidable problems.
- The Reduction Method is the main technique for establishing undecidability.
- The Reduction Method: To show that a language L^* is not recursive, it suffices to find a non-recursive language L and a total Turing computable function f such that for every string x, we have that

$$x \in L \Leftrightarrow f(x) \in L^*.$$

  - Such a function f is called a reduction of L to L^*
  - $L \preccurlyeq L^*$ means that there is a reduction of L to L^*.

The Reduction Method

- The Halting Problem was the first fundamental decision problem shown to be undecidable.

- The reduction method was used to show that the following problems are undecidable:
  - Post's Correspondence Problem
  - Hilbert's 10th Problem.

- In turn, Post's Correspondence Problem and Hilbert's 10th Problem have been used to show, via the reduction method, that numerous other problems are undecidable.

- The advantage of working with Post's Correspondence Problem is that it is a purely combinatorial problem.

Computability and Complexity

- From the 1930s on, an extensive investigation of the boundary between decidability and undecidability has been carried out by mathematical logicians.
  - This investigation has given rise to the field of computability theory.

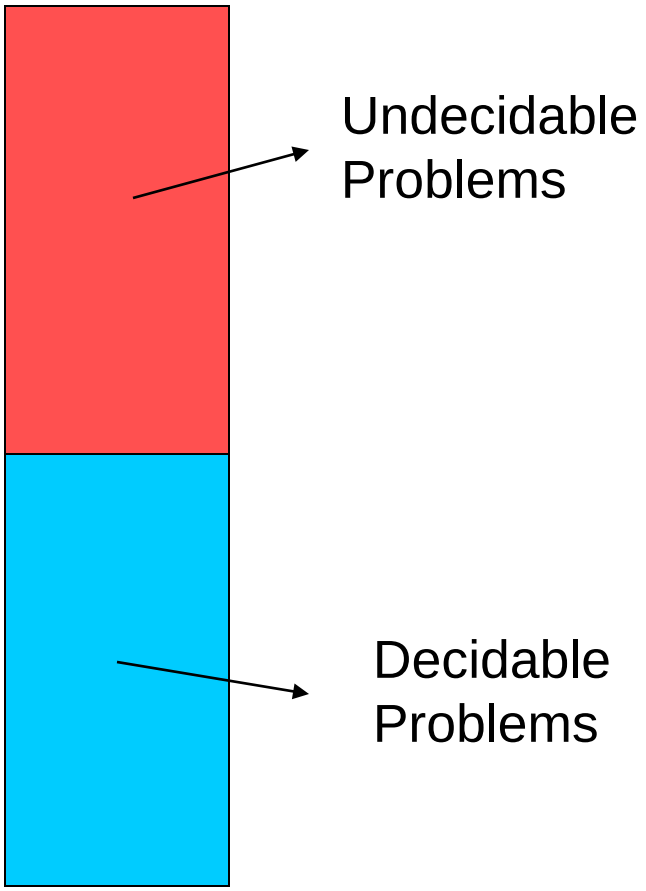
    (What can be automated?)

- From the 1960s on, an extensive investigation of decidable problems has been carried out by computer scientists aiming to identify the boundary between tractability and intractability.
  - This investigation has given rise to the field of computational complexity.

    (What can be efficiently automated?)

- Logic has played a major role in both computability theory and computational complexity.

Decidable Problems and Computational Complexity



- Computational Complexity is the quantitative study of decidable problems.

- "From these and other considerations grew our deep conviction that **there must be quantitative laws that govern the behavior of information and computing**. The results of this research effort were summarized in our first paper on this topic, which also named this new research area, "On the computational complexity of algorithms"."

J. Hartmanis, Turing Award Lecture, 1993

Computational Complexity Classes

- Decidable problems are grouped together in computational complexity classes.
- Each computational complexity class consists of all problems that can be solved in a computational model under certain restrictions on the resources used to solve the problem.
- Examples of computational models:
  - Turing Machine TM (deterministic Turing machine)
  - Non-deterministic Turing machine NTM
  - ...
- Examples of resources:
  - Amount of time needed to solve the problem
  - Amount of space (memory) needed to solve the problem.
  - ...

The Five Basic Computational Complexity Classes

- LOGSPACE (or, L): All decision problems solvable by a TM using extra memory bounded by a logarithmic amount in the input size.
- NLOGSPACE (or, NL): All decision problems solvable by a NTM using extra memory bounded by a logarithmic amount in the input size.
- P (or, PTIME): All decision problems solvable by a TM in time bounded by some polynomial in the input size.
- NP: All decision problems solvable by a NTM in time bounded by some polynomial in the input size.
- PSPACE: All decision problems solvable by a TM using memory bounded by a polynomial in the input size.

The Complexity Class P

- P occupies a central place in computational complexity.

- A. Cobham and J. Edmonds (around 1965):
  P can be identified with the class of all "tractable" decision problems, that is, all decision problems for which "good" algorithms exist.
  Justification: If p(n) is a polynomial, then for all large n, we have that $p(n) << 2^n$.

The Complexity Class P

- P occupies a central place in computational complexity.

- A. Cobham and J. Edmonds (around 1965):
  P can be identified with the class of all "tractable" decision problems, that is, all decision problems for which "good" algorithms exist.
  Justification: If p(n) is a polynomial, then for all large n, we have that $p(n) << 2^n$.

- "Tractable" does not mean "practical", since a n^{100} algorithm is not "practical" (but is better than a 2^n algorithm on all sufficiently large inputs).

The Five Basic Computational Complexity Classes

Theorem:

- The following inclusions hold:
  $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$.

- Moreover, it is known that $\text{LOGSPACE} \subset \text{PSPACE}$.

- **No** other proper inclusion between these classes is known at present. In particular, it is **not** known whether P = NP.

Note:

- The question: "is P = NP?" is the central open problem in computational complexity.
- It is one of the Millennium Prize Problems – see http://www.claymath.org/millennium/

P vs. NP

Problems in P:

- Model Checking for Propositional Logic: Given a formula of propositional logic and a truth assignment, does the assignment satisfy the formula? (Exercise: give a polynomial-time algorithm)
- Connectivity: Given a graph, is it connected?
- 2-Colorability: Given a graph, is it 2-colorable
- Linear Inequalities: Given a system of linear inequalities with integer coefficients, does it have a solution (consisting of rational numbers)?

Problems in NP: (none of the problems below is known to be in P)

- Satisfiability: Given a propositional formula, is it satisfiable?
- 3-Colorability: Given a graph, is it 3-colorable?
- Integer Linear Inequalities: Given a system of linear inequalities with integer coefficients, does it have an all-integer solution?
  - Very special case of Hilbert's 10th Problem;
  - Membership in NP is not obvious – proof is required.

Another Perspective on the classes P and NP

- Intuitively, P is the class of all decision problems for which we can find a "solution" or a "proof" efficiently (i.e., in time bounded by a polynomial in the size of the input).
- Intuitively, NP is the class of all decision problems for which we can guess a "candidate solution (proof)" and verify efficiently (i.e., in polynomial time) that it is indeed a "solution (proof)".

Problem	Candidate Solution (Proof)
Satisfiability	An assignment of boolean values to the variables
3-Colorability	An assignment of colors **B**, **R**, **G** to the nodes.

P vs. NP

- The “P = NP?” question is equivalent to the following:

- **Question:**

  Is it true that every decision problem for which a “candidate solution” or “candidate proof” can be verified efficiently has the property that a “solution” or “proof” can also be found efficiently?

  - In particular, “verified efficiently” means that the “candidate solution” or “candidate proof” must be “small” (bounded by a polynomial in the size of the input).

  Informally, this question amounts to:

  - Is proof discovery more difficult than proof checking?

- **Note:**

  The prevailing belief is that the answer to the above question is “**No**”, which means that P ≠ NP.

Sanity Check

Question: What is wrong with the "proof" of the following claim?

Claim: Hilbert's 10th Problem is in NP (hence it is decidable).

"Proof": Given a polynomial $p(x_1,...,x_n)$ with integer coefficients,

- guess a candidate all-integer solution $(a_1,...,a_n)$;
- substitute a_i for x_i, $1 \leq i \leq n$;
- evaluate $p(a_1,...,a_n)$ and check that it is equal to 0.

Since evaluating a polynomial on a given input is a polynomial-time task, it follows that Hilbert's 10th Problem is in NP.

Sanity Check

Question: What is wrong with the "proof" of the following claim?

Claim: Hilbert's 10th Problem is in NP (hence it is decidable).

"Proof": Given a polynomial $p(x_1,...,x_n)$ with integer coefficients,

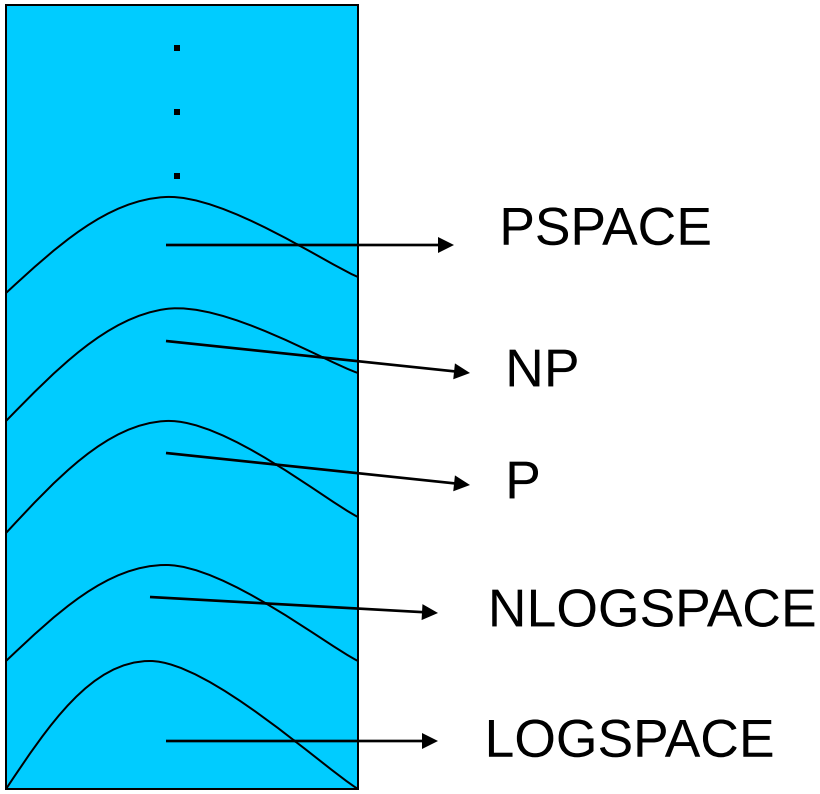
- guess a candidate all-integer solution $(a_1,...,a_n)$;
- substitute a_i for x_i, $1 \leq i \leq n$;
- evaluate $p(a_1,...,a_n)$ and check that it is equal to 0.

Since evaluating a polynomial on a given input is a polynomial-time task, it follows that Hilbert's 10th Problem is in NP.

Answer: It is **not** true that if an all-integer solution exists, then a "small" one exists (bounded by a polynomial in the size of p).

Computational Complexity Classes

Classification of Decidable Problems (not on scale)



There are many other complexity classes. For a comprehensive catalog, visit the Complexity Zoo at

qwiki.stanford.edu/wiki/Complexity_Zoo

Computational Complexity Classes

- Recall that the following inclusions hold:

  $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$.

  Moreover, it is known that $\text{LOGSPACE} \subset \text{PSPACE}$. However, no other proper inclusion between these classes is known at present.

- **Questions:**
  - What evidence do we have that these inclusions are proper?
  - What are "representative" problems that we believe belong to one of these complexity classes, but to no lower complexity class?
  - If we come across a decidable problem of interest, what tools do we have to place it in the "right" complexity class?

Complete Problems

- A key property of most complexity classes is that they possess complete problems.

- Intuitively, complete problems are the "hardest" problems in the class in the sense that every other problem can be reduced to it. Hence, they are the "most representative" problems in the class.

- Definition: Let $\mathcal{C}$ be a complexity class.

  A decision problem Q is $\mathcal{C}$-complete if the following conditions hold:
  - Q is in $\mathcal{C}$.
  - Q is $\mathcal{C}$-hard: If Q' is in $\mathcal{C}$, then there is a "suitable" total Turing computable function f such that for every string x, we have that

    $$x \in Q' \Leftrightarrow f(x) \in Q.$$

    - "suitable" means that f can be computed with fewer resources than those used to define $\mathcal{C}$.
    - So, f is a reduction of a restricted nature.

Complete Problems and Reductions

Complexity Class	Reductions for Complete Problems
PSPACE	Polynomial-time computable
NP	Polynomial-time computable
P	Logspace-computable
NL	Logspace-computable

- Definition: Let $\mathcal{C}$ be a complexity class.
  A decision problem Q is $\mathcal{C}$-complete if the following conditions hold:
  - Q is in $\mathcal{C}$.
  - Q is $\mathcal{C}$-hard: If Q' is in $\mathcal{C}$, then there is a "suitable" total Turing computable function f such that for every string x, we have that
    $$x \in Q' \Leftrightarrow f(x) \in Q.$$
    - "suitable" means that f can be computed with fewer resources than those used to define $\mathcal{C}$.

Complete Problems for Complexity Classes

- Definition: A decision problem Q is NP-complete if
  - Q is in NP.
  - Q is NP-hard: If Q' is in NP, then there is a polynomial-time computable function f such that for every string x, we have that

    $$x \in Q' \Leftrightarrow f(x) \in Q.$$

    - Such an f is a polynomial-time reduction of Q' to Q ($Q' \preccurlyeq_p Q$)
- Definition: A decision problem Q is P-complete if
  - Q is in P.
  - Q is P-hard: If Q' is in P, then there is a logarithmic-space computable function f such that for every string x, we have that

    $$x \in Q' \Leftrightarrow f(x) \in Q.$$

    - Such an f is a logarithmic-space reduction of Q' to Q ($Q' \preccurlyeq_{log} Q^*$)

Complete Problems for Complexity Classes

Fact:

- Each of the complexity classes NL, P, NP, PSPACE possesses complete problems.
- Moreover, logic provides “natural” complete problems for each of these classes.

Theorem: (all undefined notions to be explained later on)

- QBF (Quantified boolean formulas) is PSPACE-complete.
- SAT (Satisfiability of CNF formulas) is NP-complete.
- Horn SAT (Satisfiability of Horn formulas) is P-complete.
- 2-SAT (Satisfiability of 2CNF-formulas) is NL-complete.