Classifying Problems into Complexity Classes

William Gasarch [*]

Univ. of MD at College Park

November 21, 2015

Abstract

A fundamental problem in computer science is, stated informally: *Given a problem, how hard is it?*. We measure hardness by looking at the following question: *Given a set A whats is the fastest algorithm to determine if "$x \in A$?"* We measure the speed of an algorithm by how long it takes to run on inputs of length n, as a function of n. For example, sorting a list of length n can be done in roughly $n \log n$ steps.

Obtaining a fast algorithm is only half of the problem. Can you prove that there is no better algorithm? This is notoriously difficult; however, we can classify problems into *complexity classes* where those in the same class are of roughly the same complexity.

In this chapter we define many complexity classes and describe natural problems that are in them. Our classes go all the way from regular languages to various shades of undecidable. We then summarize all that is known about these classes.

1 Introduction

A fundamental problem in computer science is, stated informally:

Given a problem, how hard is it?

For a rather concrete problem the answer might be *it will take 2 hours of computing time on a supercomputer* or *this will take a team of 10 programmers 2 years to write the program.* For a class of problems of the same type (e.g., sort a list) the complexity usually depends on the input size. These are the kinds of problems we will consider. Our concern will usually be how much time or space the problem takes to finish *as a function of the input size.* Our problems will be static: usually set membership: Given a string x, is $x \in A$?

Example 1.1 Given a string $x \in \{0,1\}^n$ we want to know if it is in 0^* (a string of all 0's). An algorithm for this problem is to scan the string and keep track of just one thing: have you seen a 1 or not? As soon as you do, stop and output NO. If you finish the scan and have not seen a 1 then output YES. Note that this take $O(n)$ steps and $O(1)$ space, and scanned

[*]University of Maryland, College Park, MD 20742. `gasarch@cs.umd.edu`

the input once. Languages like this are called *regular* or DSPACE($O(1)$) (we will define this later).

Example 1.2 Given a string $x \in \{0,1\}^n$ we want to know if the number of 0's equals the number of 1's. An algorithm for this problem is to scan the string and keep track of just two things: the number of 0's and the number of 1's. At the end of the scan see if they are the same. If so then output YES else output NO. This again takes $O(n)$ steps. How much space does it take? We have to store 2 numbers that are between 0 and n so this takes $O(\log n)$ space. Languages like this are called DSPACE($O(\log n)$) (we will define this later). This particular language is also called *context free*; however we will not be discussing that class in this chapter.

Most of the sections of this chapter define a complexity class and gives some natural problems in it. In all cases we are talking about worst case. For example, if we say that a problem requires n^2 steps we mean that for any algorithm there is an input of length n where it takes n^2 steps. As such some of the problems discussed may not be as complex in real life if the inputs are not the bad ones. We won't discuss this further except to say that a problem might not be quite as bad as it appears here.

We then have additional sections: (1) a look at other complexity measures, (2) a summary of what we've done, (3) a literal discussion *what is a natural problem*,

The natural problems we consider are mainly from graph theory, games, formal langauge theory, and logic. A good reference for some of the problems in logic (with proofs) is a book by Ferrante and Rackoff [22]. There are many natural problems in other areas (e.g., model checking, artificial intelligence, Economics, Physics); however, to even define these problems is beyond the scope of this chapter.

There are many complexity classes that we do not discuss in this chapter. How many complexity classes are there? Literally hundreds. The website *Complexity Zoo* [2] currently lists around 500.

2 Time and Space Classes

The material in this chapter is due to Hartmanis and Stearns [29].

We want to classify problems by how much time or space they take to solve as a function of the length of the input. Say the input is of size n. If the algorithm takes n steps or $n/2$ steps or $10n$ steps we do not want to care about those differences. While the difference between n and $100n$ matters in the real world, as a first cut at the complexity it does not. We need a way to say we don't care about constants.

Def 2.1 Let f be a monotone increasing function from $\mathbb{N}$ to $\mathbb{N}$.

1. $O(f)$ is the class of all functions g such that there exists a constants n_0, c such that $(\forall n \geq n_0)[g(n) \leq cf(n)]$.

2. $\Omega(f)$ is the class of all functions g such that there exists a constants n_0, c such that $(\forall n \ge n_0)[g(n) \ge cf(n)]$.

When we define problems we code everything into strings over an alphabet. We are concerned with the complexity of a set of strings.

Notation 2.2 Let A and B be sets.

1. $AB = \{xy \mid x \in A \text{ AND } y \in B\}$.
2. A^i is $A \cdots A$ (i times). If $i = 0$ then A^0 is the empty string.
3. $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$

Notation 2.3 Let Σ be a finite alphabet (often $\{a, b\}$ or $\{0, 1\}$). A *problem* is a set $A \subseteq \Sigma^*$. The problem is to, given x, determine if $x \in A$.

Convention 2.4 We will use the term *Program* informally. To formalize it we would define a Turing Machine.

Def 2.5 Let T be a monotone increasing function from $\mathbb{N}$ to $\mathbb{N}$. DTIME($T(n)$) is the set of all sets $A \subseteq \Sigma^*$ such that there exists a program M such that

1. If $x \in A$ then $M(x) = YES$.
2. If $x \notin A$ then $M(x) = NO$.
3. For all x, $M(x)$ takes time $\le O(T(|x|))$.

Def 2.6 Let S be a monotone increasing function from $\mathbb{N}$ to $\mathbb{N}$. DSPACE($S(n)$) is the set of all sets $A \subseteq \Sigma^*$ such that there exists a program M such that

1. If $x \in A$ then $M(x) = YES$.
2. If $x \notin A$ then $M(x) = NO$.
3. For all x, $M(x)$ uses space $\le O(S(|x|))$.

Def 2.7 One can define a function being in DTIME($T(n)$) or DSPACE($S(n)$) similarly.

The program referred to in Definition 2.5 is deterministic. On input x there is only one way for a computation to go. We now define nondeterministic programs. We consider them mathematical devices. We do not consider them to be real. However, they will be useful for classifying problems.

Def 2.8 A *Nondeterministic Program* is a program where, in any state, there is a choice of actions to take. For example, a line might read

$$x := x + 1 \text{ OR } y := y + 4$$

If M is a nondeterminism program then what does it mean to run $M(x)$? We do not define this. However, we do say what it means for $M(x)$ to accept.

Def 2.9 Let M be a nondeterministic program. $M(x)$ *accepts* if there is *some* choice of instructions so that it accepts. $M(x)$ *rejects* if there is no choice of instructions that makes it accept.

Def 2.10 Let T be a monotone increasing function from $\mathbb{N}$ to $\mathbb{N}$. NTIME($T(n)$) is the set of all sets $A \subseteq \Sigma^*$ such that there exists a program M such that

1. If $x \in A$ then $M(x)$ accepts.
2. If $x \notin A$ then $M(x)$ rejects.
3. For all x any computation path of $M(x)$ takes time $\le O(T(|x|))$.

Def 2.11 Let S be a monotone increasing function from $\mathbb{N}$ to $\mathbb{N}$. NSPACE($S(n)$) is the set of all sets $A \subseteq \Sigma^*$ such that there exists a nondeterministic program M such that

1. If $x \in A$ then $M(x) = YES$.
2. If $x \notin A$ then $M(x) = NO$.
3. For all x any computation path of $M(x)$ uses space $\le O(S(|x|))$.

Note 2.12 There is no really useful way to define a nondeterministic device computing a function.

Notation 2.13 The class DTIME($n^{O(1)}$) is $\bigcup_{i=1}^{\infty}$ DTIME(n^i). We may use $O(1)$ inside other time or space classes. The meaning will be clear from context.

We will be interested in seeing which time or space class a problem is in. Within a class there may be harder and easier problems. There will be problems that are (informally) the hardest in that class. We do not define *completeness* rigorously; however we state the following property of it;

Fact 2.14 *Let X and Y be complexity classes such that $X \subset Y$ (proper containment). If a problem A is Y-complete then $A \notin X$.*

3 Relations Between Classes

Throughout this section think of $T(n)$ and $S(n)$ as increasing.

The following theorem is trivial.

Theorem 3.1 *Let $T(n)$ and $S(n)$ be computable functions.*

1. DTIME($T(n)$) $\subseteq$ NTIME($T(n)$).
2. DSPACE($S(n)$) $\subseteq$ NSPACE($S(n)$).
3. DTIME($T(n)$) $\subseteq$ DSPACE($T(n)$).
4. NTIME($T(n)$) $\subseteq$ NSPACE($T(n)$).

The following theorem is easy but not trivial.

Theorem 3.2 *Let $T(n)$ and $S(n)$ be computable functions.*

1. NTIME($T(n)$) $\subseteq$ DTIME($2^{O(T(n))}$). *(Just simulate all possible paths.)*
2. NTIME($T(n)$) $\subseteq$ DSPACE($O(T(n))$). *(Just simulate all possible paths— keep a counter for which path you are simulating.)*

The following theorems have somewhat clever proofs.

Theorem 3.3 *Let $S(n)$ be a computable functions.*

1. NSPACE($S(n)$) $\subseteq$ DSPACE($O(S(n)^2)$). *This was proven by Savitch [62] and is in any textbooks on complexity theory.*
2. NSPACE($S(n)$) $\subseteq$ DTIME($O(2^{S(n)}$). *This seems to be folklore.*

The following are by diagonalization. Hence the sets produced are not natural. Even so, the existence of such sets will allow us to later show natural sets that are in one complexity class and not in a lower one.

Theorem 3.4 *For all $T(n)$ there is a set $A \in$ DTIME($T(n)\log T(n)$)) $-$ DTIME($T(n)$). (The $T(n)\log T(n)$ comes from some overhead in simulating a k-tape Turing Machine with a 2-tape Turing Machine.) This is The Time Hierarchy Theorem and is due to Hartmanis and Stearns [29].*

Theorem 3.5 *Let S_1 and S_2 be computable functions. Assume $\lim_{n\rightarrow\infty} \frac{S_1(n)}{S_2(n)} = \infty$. Then there exists a set $A \in$ DSPACE($S_1(n)$) $-$ DSPACE($S_2(n)$). Hence DSPACE($S_2(n)$) $\subset$ DSPACE($S_1(n)$). This is The Space Hierarchy Theorem and seems to be folklore.*

4 DSPACE(1)=Regular Languages

There are many different definitions of regular languages that are all equivalent to each other. We present them in the next definition Recall that DSPACE(1) allows for a constant amount of time, not just 1 step.

Def 4.1 A language A is *regular* (Henceforth REG) if it satisfies any of the equivalent conditions below.

1. $A \in$ DSPACE(1).
2. $A \in$ NSPACE(1).
3. A is in DSPACE(1)) by a program that, on every computation path, only scans the input once. (This is equivalent to being recognized by a deterministic finite automata, abbreviated DFA.)
4. A is in NSPACE(1) by a program that, on every computation path, only scans the input once. (This is equivalent to being recognized by a nondeterministic finite automata, abbreviated NDFA. When you convert an NDFA to a DFA you may get an exponential blowup in the number of states.)
5. A is generated by a regular expression (we define this later).

The equivalence of DSPACE(1)-scan-once and NSPACE(1)-scan-once is due to Rabin and Scott [56] and is usually stated as DFA=NDFA. It is in all textbooks on formal language theory. The equivalence of DSPACE(1) and DSPACE(1)-scan once is folklore but has its origins in the Rabin-Scott paper.

We define regular expressions α and the language they generate $L(\alpha)$.

Def 4.2 Let Σ be a finite alphabet.

1. $\emptyset$ (the empty set) is a regular expression. $L(\emptyset) = \emptyset$.
2. e (the empty string) is a regular expression. $L(e) = \{e\}$.
3. For all $\sigma \in \Sigma$, σ is a regular expression. $L(\sigma) = \{\sigma\}$.
4. If α and β are regular expressions then:
   (a) $(\alpha \cup \beta)$ is a regular expression. $L(\alpha \cup \beta) = L(\alpha) \cup \mathrm{L}(\beta)$.
   (b) $\alpha\beta$ is a regular expression. $L(\alpha\beta) = L(\alpha)L(\beta)$. (Recall that if A is a set and B is a set then $AB = \{xy \mid x \in A \text{ AND } y \in B\}$.)
   (c) α^* is a regular expression. $L(\alpha^*) = L(\alpha)^*$. (Recall that if A is a set then $A^* = A^0 \cup A \cup AA \cup AAA \cdots$.

We give examples or regular sets after the next bit of notation.

Def 4.3 Let Σ be a finite set. Let $w \in \Sigma^*$. Let $\sigma \in \Sigma$. Then $\#_\sigma(w)$ is the number of σ's in w.

Def 4.4 Let $x, y, z \in \mathbb{N}$. Then $x \equiv y \pmod{z}$ means that z divides $x - y$.

Example 4.5 The following sets are regular.

$$\{w \in \{a,b\}^* \mid \#_a(w) \equiv \#_b(w) + 10 \pmod{21}\}$$

You can replace 10 and 21 with any constants.

$$\{w \in \{a,b\}^* \mid abab \text{ is a prefix of } w\}$$

$$\{w \in \{a,b\}^* \mid abab \text{ is a suffix of } w\}$$

$$\{w \in \{a,b\}^* \mid abab \text{ is a substring of } w\}$$

You can replace *abab* with any finite string.

If A_1, A_2 are regular languages then so are $A_1 \cap A_2$, $A_1 \cup A_2$ and $\overline{A_1}$. Hence any Boolean combination of the above is also a regular language. For example

$$\{w \in \{a,b\}^* \mid abab \text{ is a substring of } w \text{ AND } \#_a(w) \not\equiv \#_b(w) + 10 \pmod{21}\}.$$

Example 4.6 Throughout this example $w = d_n d_{n-1} \cdots d_0 \in \{0,1,2,3,4,5,6,7,8,9\}^*$ is thought of as a number in base 10.

Is it easy to tell if $w \equiv 0 \pmod{2}$? Yes: $w \equiv 0 \pmod{2}$ iff $d_0 \equiv 0 \pmod{2}$. Hence

$$\{w \mid w \equiv 0 \pmod{2}\} \text{ is regular.}$$

Is it easy to tell if $w \equiv 0 \pmod{3}$? Yes: $w \equiv 0 \pmod{3}$ iff $d_0 + d_1 + \cdots + d_n \equiv 0 \pmod{3}$. By keeping a running total mod 3 one can show that

$$\{w \mid w \equiv 0 \pmod{3}\} \text{ is regular.}$$

There are also well known divisibility tricks for divisibility by 4,5,6,8,9,10,11. What about 7? There are two questions to ask here

- Is there a trick for divisibility by 7? (This question is not rigorous.)
- Is the set $DIV7 = \{w \mid w \equiv 0 \pmod{7}\}$ regular?

One can interpret the second question as a rigorous restatement of the first. When you see the answer you may want to reconsider that interpretation.

We show that $\{w \mid w \equiv 0 \pmod{7}\}$ is regular. Note that

$$\begin{array}{lll} 10^0 & \equiv 1 & \pmod{7} \\ 10^1 & \equiv 3 & \pmod{7} \\ 10^2 & \equiv 2 & \pmod{7} \\ 10^3 & \equiv 6 & \pmod{7} \\ 10^4 & \equiv 4 & \pmod{7} \\ 10^5 & \equiv 5 & \pmod{7} \\ 10^6 & \equiv 1 & \pmod{7} \end{array}$$

Hence $d_n d_{n-1} d_{n-2} \cdots d_0$ is equivalent mod 7 to the following:

$$\begin{array}{ccccccccccccc} d_0 & + & 3d_1 & + & 2d_2 & + & 6d_3 & + & 4d_4 & + & 5d_5 & + \\ d_6 & + & 3d_7 & + & 2d_8 & + & 6d_9 & + & 4d_{10} & + & 5d_{11} & + \\ d_{12} & + & 3d_{13} & + & 2d_{14} & + & 6d_{15} & + & 4d_{16} & + & 5d_{17} & + \\ \vdots & + & \vdots & + & \vdots & + & \vdots & + & \vdots & + & \vdots & + \end{array}$$

We can use this to show that the set $DIV7$ is regular. To determine if $w \in DIV7$, when scanning w, one only needs to keep track of (1) the weighted sum mod 7, and (2) the index mod 6 of i. This would lead to a 42-state finite automata. Whether you want to consider this a *trick* for divisibility by 7 or not is a matter of taste.

Example 4.7 We want to look at sets like

$$\{(b, c, A) \mid b \in A \text{ AND } c + 1 \notin A\}.$$

Are such sets regular? We first need to have a way to represent such sets. We represent a number x by a string of x 0's and then a 1 and then we do not care what comes next. So for example 000100 represents 3 and so does 000110. we will denote this by saying that $0001 * *$ represents 3 (we may have more $*$'s). We represent finite sets by a bit vector. For example 11101 represents the set $\{0, 1, 2, 4\}$

How do we represent a triple? We use the alphabet $\{0, 1\}^3$. We give an example. The triple $(3, 4, \{0, 1, 2, 4, 7\})$ is represented by the following (The top line and the b, c, A are not there. They are, as we say in the Ed biz, visual aids.)

	0	1	2	3	4	5	6	7
b	0	0	0	1	0	$*$	$*$	$*$
c	0	0	0	0	1	$*$	$*$	$*$
A	1	1	1	0	1	0	0	1

With this representation the set

$$\{(b, c, A) \mid b \in A \text{ AND } c + 1 \notin A\}$$

is regular.

Much more can be said. We define a class of formulas $\phi(\vec{x}, \vec{X})$, such that the set of $(\vec{a}, \vec{A})$ that make them true is regular. We will use this again in Section 18.

We will only use the following symbols.

1. The logical symbols $\wedge$, $\neg$, $(\exists)$.
2. Variables $x_1, x_2, x_3, \ldots$ that range over $\mathbb{N}$. (We use x, y, z when there are less than 4 variables.)
3. Variables $X_1, X_2, X_3, \ldots$ that range over finite subsets of $\mathbb{N}$. (We use X, Y, Z when there are less than 4 variables.)
4. Symbols: $=$, $<$, $\in$, S (meaning $S(x) = x + 1$).
5. Constants: 0,1,2,3,....
6. Convention: We write $x + c$ instead of $S(S(\cdots S(x))\cdots)$. Note that $+$ is not in our language.

We call this WS1S: Weak Second order Theory of One Successor. Weak Second order means quantify over finite sets. What Does One Successor Mean? Our basic objects are numbers. We could view numbers as strings in unary. In that case $S(x) = x1$. If our basic objects were strings in $\{0, 1\}^*$ then we could have two successors $S_0(x) = x0$ and $S_1(x) = x1$. The theory of those strings is WS2S.

Def 4.8 An *Atomic Formulas* is:

1. For any $c \in \mathbb{N}$, $x = y + c$ is an Atomic Formula.
2. For any $c \in \mathbb{N}$, $x < y + c$ is an Atomic Formula.
3. For any $c, d \in \mathbb{N}$, $x \equiv y + c \pmod{d}$ is an Atomic Formula.
4. For any $c \in \mathbb{N}$, $x + c \in X$ is an Atomic Formula.
5. For any $c \in \mathbb{N}$, $X = Y + c$ is an Atomic Formula.

Def 4.9 A *WS1S Formula* is:

1. Any atomic formula is a WS1S formula.
2. If ϕ_1, ϕ_2 are WS1S formulas then so are
   (a) $\phi_1 \wedge \phi_2$,
   (b) $\phi_1 \vee \phi_2$

   (c) $\neg\phi_1$

3. If $\phi(x_1,\ldots,x_n,X_1,\ldots,X_m)$ is a WS1S-Formula then so are
   (a) $(\exists x_i)[\phi(x_1,\ldots,x_n,X_1,\ldots,X_m)]$
   (b) $(\exists X_i)[\phi(x_1,\ldots,x_n,X_1,\ldots,X_m)]$

For any WS1S formula $\phi(\vec{x},\vec{X})$ the following set is regular:

$$\{(\vec{a},\vec{A}) \mid \phi(\vec{a},\vec{A}) \text{ is true }\}.$$

The proof uses the closure of regular languages under union (for $\vee$), intersection (for $\wedge$), complementation (for $\neg$), and projection (for $(\exists)$). The closure under projection involves taking an NDFA and converting it to a DFA. This results in an exponential blowup in the number of states. Hence the DFA's one obtains can be quite large.

5 L = DSPACE($\log n$)

For this section we let L = DSPACE($\log n$). It is known that $REG \subset$ L. We give examples of sets in L − REG.

Example 5.1 Intuitively, any set where you need to keep track of the number of a's or any unbounded quantity is not regular. Formally you would prove the following non-regular using the pumping lemma (perhaps together with closure properties). We do not state or use this lemma. The following sets are in L − REG.

$$\{a^n b^n \mid n \in \mathbb{N}\}$$

$$\{a^n b^m \mid n,m \in \mathbb{N} \text{ AND } n \le m\}$$

$$\{w \mid \#_a(w) = \#_b(w)\}$$

All of these are in L since you need only keep track of the number of a's and b's which will take $O(\log n)$ space.

Example 5.2 Consider the following problem. The input is an undirected graph together with two nodes.

$$CONN = \{(G,s,t) \mid \text{ there is a path in } G \text{ from } s \text{ to } t\ \}.$$

$CONN$ is in NSPACE($\log n$): start with a pointer to s and guess a neighbor x_1 to goto. Then guess a neighbor x_2 of x_1 to goto. Keep doing this. If you ever get to t then stop and accept.

Is $CONN$ in L? This problem was open for a long time. There were reasons to thing that $CONN$ is in L, namely that $CONN$ is in a randomized version of L. Omer Reingold [59] proved $CONN \in$ L in 2008. The proof is very difficult.

What if the graph is directed? This problem is thought to be harder and will be discussed in the next section.

Example 5.3 The following problems are in L − REG:

1. Given a graph, is it planar? (See [6].)
2. Given two trees are they isomorphic? (See [39].)
3. Given two planar graphs, are they isomorphic? (See [17].)
4. Given n permutations $p_1, \ldots, p_n$, is their product the identity. (See [68].)

6 NL = NSPACE($\log n$)

For this section we let NL = NSPACE($\log n$). Clearly L $\subseteq$ NL. It is not known if this inclusion is proper; however, most theorists think L $\neq$ NL.

Example 6.1 Consider the problem

$$DCONN = \{(G, s, t) \mid \text{ there is a path in } G \text{ from } s \text{ to } t\ \}.$$

(The graph G is directed. This is important.)

This problem may *look* similar to $CONN$; however, it is not. Thought experiment: let $A \in$ NSPACE($\log n$). Let $x \in \Sigma^n$. View the space that the program uses while computing on x to be on a tape of length $O(\log n)$, which we call the *worktape*. Since the worktape is of length $O(\log n)$ there are only a polynomial number of possibilities for it. One can form a directed graph by taking the vertices to be the possible worktapes, and put an edge from u to v if it is possible to go (recall that the machine is nondeterministic), in one step of M, from u to v This directed graph has a path from the start state to an accept state iff $M(x)$ accepts. Hence we can reduce *any* problem in NSPACE($\log n$) to the problem $DCONN$. Formally $DCONN$ is NL-complete.

If $DCONN \in$ L then L = NL. Hence most theorists think $DCONN \notin$ L.

7 P = DTIME($n^{O(1)}$)

Let P = DTIME($n^{O(1)}$), also called *Polynomial Time.* NL $\subseteq$ P by Theorem 3.3.2. It is not known if this inclusion is proper; however, most theorists think NL $\neq$ P.

P is considered by theorists to be the very definition of *feasible* (though see the next section on randomized polynomial time). Why is polynomial time so revered?

Polynomial time is usually contrasted with brute force search. Lets say you want to, given a Boolean formula $\phi(x_1, \ldots, x_n)$, determine if there is some truth assignment that makes it true. The naive approach is to look at all 2^n possibilities. Lets say you could use symmetries to cut it down to 2^{n-10}. You are still doing brute force search, with a few tricks. But if you got an algorithm in n^{100} steps then you are most definitely **not** doing brute force search. Even though the exponent is large it is likely that the cleverness used to avoid brute force search can be further exploited to obtain a practical algorithm.

We present several natural problems in P. Some are expressed as functions rather than sets as that is more natural for them. They are not believed to be in NL.

Example 7.1 If $G = (V, E)$ is a graph then $U \subseteq V$ is a *vertex cover* if every edge in E has some vertex of U as an endpoint. Let

$$VC_{17} = \{G \mid G \text{ has a vertex cover of size 17 }\}.$$

$VC_{17} \in$ P by the following simple algorithm: look at all subsets of 17 vertices and for each one check if it's a vertex cover. This take $O(n^{17})$ time. Can we do better? We'll consider this in Section 9.

Example 7.2 Given a weighted graph $G = (V, E)$ (no negative weights) and a source node s, find, for each node t, the shortest path from s to t. The weights are in binary and have maximum length L. Hence the input is of size $O(|V|^2L)$. All of the arithmetic operations the algorithm does take time $O(L)$. Dijkstra's algorithm [20] takes $O(|V|^2)$ arithmetic operations, hence time $O(|V|^2L)$ which is polynomial in the length of the input. A later implementation using a Fibonacci heap takes $O(|E| + |V|\log(|V|)$ arithmetic operations, so time $O((|E| + |V|\log(|V|)L)$ which is an improvement if the graph is sparse ($|E| = O(|V|)$).

Example 7.3 Given a weighted graphs $G = (V, E)$ find, for all pairs of vertices $\{s, t\}$ the shortest path between s and t. The weights are in binary and have maximum length L. Hence the input is of size $O(|V|^2L)$. All of the arithmetic operations the algorithm does take time $O(L)$. The Floyd-Warshall algorithm solves this problem in $O(|V|^3)$ arithmetic operations so time $O(|V|^3L)$ which is polynomial in the length of the input. The algorithm was discovered independently by Floyd [24], Warshall [74], and Roy [61] (it is in many algorithms textbooks).

Example 7.4 Given a weighted graph $G = (V, E)$ find a min weight spanning tree. The weights are in binary and have maximum length L. Hence the input is of size $O(|V|^2L)$. All of the arithmetic operations the algorithms do take time $O(L)$. There are basically two algorithms for this, one due to J. Kruskal [35] and one due to Prim [53] (both are in many algorithms textbooks). We omit the time bound obtained by multiplying the number of arithmetic operations by $O(L)$. Kruskal's algorithm originally took $O(E \log V)$ arithmetic operations. Prim's algorithm originally took $O(|V|^2)$; however, a later implementation using a Fibonacci heap and adjacency lists takes $O(|E| + |V| \log |V|)$. The best known algorithm for this problem is due to Chazelle [14] and runs in time $O(n\alpha(m,n))$ where $\alpha(m,n)$ is the inverse of the Ackermann function (see Section 19). Note that this is very close to linear. If this is also a lower bound then the result would be optimal and Ackermann's function would have popped up in a natural place. Alas, Chazelle thinks this is unlikely.

Example 7.5 Linear programming: Given a matrix A and a two vectors b and c find the vector of x that maximizes $c \cdot x$ while satisfying the constraint $Ax \leq b$.

Linear programming is particularly interesting. This problem is extremely practical. The Simplex method, developed by Dantzig in 1947, solves it quickly in most cases but is not polynomial time. It is widely used and in all operations research textbooks and some algorithms textbooks. In 1979 Khachiyan [33] showed it is in polynomial time using the ellipsoid method. This algorithm is important theoretically in that the problem is now in P; however, it is slow in practice. In 1984 Karmarkar [32] produced a method that is fast in both theory and practice.

8 Randomized Polynomial Time: R

Def 8.1 A problem A is in *Randomized Polynomial Time* (henceforth R) if there is a program that flips coins such that the following happens:

1. On all inputs of length n, the program halts in time polynomial in n.
2. If $x \in A$ then the program will ACCEPT with probability $\geq 2/3$.
3. If $x \notin A$ then the program will REJECT.

Note 8.2 The 2/3 can be replaced by any $\epsilon > 0$ and even by $\frac{1}{2^n}$ where n is the length of the input.

Clearly P $\subseteq$ R. Before 1988 the theory community did not have a strong opinion on if P = R, however the opinion would have been a tendency towards P $\neq$ R. Michael Sipser [65] was an exception in that he believed P = R. In 1988 Nisan and Wigderson [50] showed that, given certain quite reasonable unproven hypothesis from complexity theory, P = R. Since then the consensus has been that P = R. This remains unproven.

At one time the quintessential natural problem in R that was not known to be in P was primality. Solovay and Strassen [67] and Rabin [57] showed primality is in R. Their algorithms are practical and used. Rabin has pointed out that if the error is (say) $1/2^{100}$ then that is less than the probability that a cosmic ray will hit a computer and flip a bit to make it incorrect. The algorithm by Rabin is sometimes called the Miller-Rabin primality test since Miller had a similar deterministic algorithm that depended on the extended Riemann hypothesis, an unproven conjectures in Number Theory.

In 2002, Agrawal-Kayal-Saxena [5] proven that primality is in P. Their algorithm is slow and not in use. However, it is very interesting to see that primality really is in P.

There is still one natural problem that is in R that is not yet known to be in P:

Example 8.3 *Given a polynomial $q(x_1, \ldots, x_n)$ and a prime p, is the polynomial identically 0 over mod p?*

Here is the randomized algorithm: Pick a random $b_1, \ldots, b_n \in \{0, \ldots, p-1\}$. Evaluate $q(b_1, \ldots, b_n) \pmod{p}$. If it is not zero then we KNOW that $q(x_1, \ldots, x_n)$ is not identically zero. If it is zero then we are not sure. So we plug in another random $b_1, \ldots, b_n$. Do this n times. If you ever get a nonzero value then you know $q(x_1, \ldots, x_n)$ is not identically zero. If you always get a zero then you know with high probability that $q(x_1, \ldots, x_n)$ is identically zero.

The following randomized class has also been defined; however, there are no natural problems in it that are not also in R.

Def 8.4 A problem A is in *Bounded Probabilistic Polynomial Time* (henceforth BPP) if there is a program that flips coins such that the following happens:

1. On all inputs of length n, the program halts in time polynomial in n.
2. If $x \in A$ then the program will ACCEPT with probability $\geq 2/3$.
3. If $x \in A$ then the program will REJECTS with probability $\geq 2/3$.

Note 8.5 The 2/3 can be replaced by any $\epsilon > 0$ and even by $\frac{1}{2^n}$ where n is the length of the input.

Clearly R $\subseteq$ BPP. Everything above about "P = R?" also applies to "P = BPP?". In particular theorists currently think P = BPP but this remains unproven. We will have a bit more to say about BPP in Section 10.

9 NP = NTIME($n^{O(1)}$)

Let NP = NTIME($n^{O(1)}$), also called *Nondeterministic Polynomial Time.* Clearly P $\subseteq$ R $\subseteq$ NP. It is not known if these inclusions are proper; however, most theorists think P = R $\subset$ NP. We will discuss their thoughts on P vs NP in more depth later.

What about BPP? It is not known if BPP $\subseteq$ NP. Since most theorists think P = BPP and P $\neq$ NP, most theorists think BPP $\subset$ NP. But it's not even known that BPP $\subseteq$ NP. In Section 10 we will state an upper bound for BPP.

NP is the most important class in computer science. It contains natural problems that we want to solve but currently seem hard to solve. Alas, there are reasons to think they will always be hard to to solve. But there are ways around that. Maybe.

We give two equivalent definitions of NP.

Def 9.1 Let A be a set.

1. $A \in$ NP if $A \in$ NTIME$(n^{O(1)})$.
2. $A \in$ NP if there exists a polynomial p and a set $B \in$ P such that

$$A = \{x \mid (\exists y)[|x| \le p(|x|) \text{ AND } (x,y) \in B]\}.$$

   The intuition here is that y is a short easily verifiable proof that $x \in A$. We often call *y the witness.*

Note that if $A \in$ NP then it is quite possible that $\overline{A} \notin$ NP. Most theorists think that NP is *not* closed under complementation. Hence we need a name for the complement of NP.

Def 9.2 A set A is in co-NP if $\overline{A}$ is in NP.

Most theorists think NP $\ne$ co-NP.

Example 9.3 A Boolean Formula $\phi(\vec{x})$ is *satisfiable* if there exists $\vec{b}$ such that $\phi(\vec{b}) = TRUE$. Let SAT be the set of all satisfiable formulas. $SAT \in$ NP. The intuition is that the satisfying assignment $\vec{b}$ is the witness for $\phi \in SAT$. Formally $p(\phi(x_1, \ldots, x_n)) = n$ and

$$B = \{(\phi, \vec{b}) \mid \phi(\vec{b}) = TRUE\}.$$

Note that while *finding* the assignment $\vec{b}$ such that $\phi(\vec{b}) = TRUE$ may be hard, *verifying* that $\phi(\vec{b}) = TRUE$ is easy. The easy verification *is not* good news for SAT, this *is not* a first step to showing that SAT is easy or in P. But it does indicate why this problem may be hard: finding the right $\vec{b}$ is hard.

You might think that SAT requires a long time to solve since you seem to need to go through all 2^n possible assignments. And this may be true. But we do not know it to be true. What haunts many complexity theorists is that someone will find a very clever way to avoid the brute force search. What comforts many complexity theorists is that SAT is NP-complete. Hence it is unlikely to be in P.

Example 9.4 A graph G is *Eulerian* if there is a path that hits every *edge* at exactly once. Let $EULER$ be the set of all Eulerian graphs. $EULER \in$ NP. The cycle that hits every edge at least once is the witness that G is Eulerian.

You might think that $EULER$ requires a long time to solve since you seem to need to go through all possible cycles. And this may be true. But we do not know it to be true.

What haunts many complexity theorists is that someone will be find a very clever way to avoid the brute force search.

If you believed the last paragraph then YOU"VE BEEN PUNKED! *EULER can* be solved quickly! It turns out that a graph is in *EULER* iff every vertex has even degree. Hence $EULER \in$ P. Euler, who was quite clever, figured this out in 1736. (though he did not use the terminology of *polynomial time*). This is just the kind of thing I warned about when talking about SAT. There could just some clever idea out there we haven't thought of yet!

Example 9.5 A graph G is *Hamiltonian* if there is a path that hits every *vertex* exactly once. Let HAM be the set of all Hamiltonian graphs. $HAM \in$ NP. The cycle that hits every vertex at least once is the witness that G is Hamiltonian.

You might think that HAM requires a long time to solve since you seem to need to go through all possible cycles. You may also be thinking, given that I fooled you with $EULER$, that you and The Who *don't get fooled again[76]*. However this time, for better or worse, HAM does really seem unlikely to be in P. In particular HAM is NP-complete and hence unlikely to be in P.

Example 9.6 If $G = (V, E)$ is a graph then $U \subseteq V$ is a *vertex cover* if every edge in E has some vertex of U as an endpoint. Let

$$VC = \{(G, k) \mid G \text{ has a vertex cover of size } k\ \}.$$

$VC \in$ NP. The vertex cover itself is the witness. VC is NP-complete and hence unlikely to be in P.

Example 9.7 The *Set Cover Problem)* is as follows: Given $S_1, \ldots, S_m \subseteq \{1, \ldots, n\}$ and a number L, is there a set $I \subseteq \{1, \ldots, m\}$ of size L such that $\bigcup_{i \in I} S_i = \bigcup_{i=1}^{n} S_i$.

The L subsets together is the witness. Set Cover is NP-complete and hence unlikely to be in P.

SAT, HAM, VC, SC are all NP-complete. So are thousands of natural problems from many different fields. Actually this means that, from the viewpoint of polynomial time, *They are all the same problem!* Are these problems not in P? Does P = NP? This is still not known.

9.1 Reasons to Think $\mathrm{P} \neq \mathrm{NP}$ and some Intelligent Objections

Scott Aaronson [3] gives very good reasons to think that $\mathrm{P} \neq \mathrm{NP}$. William Gasarch [27] gives a simplified version of Scott's reasons. Richard Lipton [41] gives some intelligent objections. We summarize some of their thoughts, and others, below.

1) For $\mathrm{P} \neq \mathrm{NP}$
Many of the problems that are NP-complete have been worked on for many years, even before these terms were formally defined. Mathematicians knew that graphs had an Euler cycle iff every vertex had even degree and were looking for a similar characterization for HAM graphs. If $\mathrm{P} = \mathrm{NP}$ then we would have found the algorithm by now.

2) For $\mathrm{P} = \mathrm{NP}$
We keep getting better and better algorithms in surprising ways. We give an example. Recall from Section 7:

$$V_{17} = \{G \mid G \text{ has a vertex cover of size 17 }\}.$$

As noted in Section 7 VC_{17} can be solved in time $O(n^{17})$. It would seem that one cannot do better. AH- but one can! We give two ways to do better to illustrate how surprising algorithms are.

Using the Graph Minor Theorem: Robertson and Seymore proved *The Graph Minor Theorem* in a series of 25 papers titled *Graph Minors I*, *Graph Minors II*, etc. Suffice it to say that this theorem is difficult. We do not state the theorem; however, we state a definition and a corollary.

Def 9.8 If G is a graph then H is a *minor* of G if one can obtain H by performing the following operations on G in some order (1) remove a vertex and all the adjacent vertices, (2) remove an edge, (3) contract an edge— that is, remove it but then merge the two endpoints into one vertex.

Def 9.9 Let $\mathcal{G}$ be a set of graphs. $\mathcal{G}$ is *closed under minors* if, for all $G \in \mathcal{G}$ if H is a minor of G then $H \in \mathcal{G}$. Examples: (1) planar graphs, (2) graphs that can be drawn in the plane with at most 100 crossings, (3) V_{17}.

Def 9.10 Let $\mathcal{G}$ be a set of graphs. $\mathcal{G}$ has a *finite obstruction set* (FOS) if there exists a finite set of graphs $H_1, H_2, \ldots, H_m$ such that $G \in \mathcal{G}$ iff none of the H_i are a minor of G. Intuitively, if $G \notin \mathcal{G}$ then there must be a solid reason for it. It was known (before the Graph Minor Theorem) that the set of planar graphs has FOS $\{K_5, K_{3,3}\}$.

Fact 9.11 *Fix H. There is an $O(n^3)$ algorithm to tell if H is a minor of G. (This was also proven by Robertson and Seymour).*

We now state the important corollary of the Graph Minor Theorem:

Corollary 9.12 *If $\mathcal{G}$ is a set of graphs that is closed under minors then it has a finite obstruction set. Using the fact above, any set of graphs closed under minors is in time $O(n^3)$.*

In particular, VC_{17} is in $DTIME(n^3)$. Note that we got a problem into better-time-bound-than-we-thought class using an incredibly hard theorem in math. Could the same happen with SAT?

Before the Graph Minor Theorem most algorithms were very clever but didn't use that much math and certainly not that much hard math (algorithms in number theory may be an exception). Hence it was plausible to say *if* P = NP *then we would have found the algorithm by now.* After the Graph Minor Theorem this is a hollow argument. It has been said:

The class P lost its innocence with the Graph Minor Theorem.

We note that the algorithm given above is insane. The constant is ginormous and the algorithm itself is nonconstructive. It can be made constructive but only be making the constant even bigger.

Using Bounded Tree Search: There is a clever way to solve VC_{17} in a bound far better than $O(n^{17})$ that does not use hard math. We form a binary tree. At the root put the graph and the empty set. Take an edge (a, b) of G. One of $\{a, b\}$ must be in the vertex cover. Make the left subchild of the root the graph without a and the set $\{a\}$. Make the right subchild of the root the graph without b and the set $\{b\}$. Repeat this process. Every node will have a graph and a set. Do this for 17 levels. If any of them lead to the empty graph then you are done and the set is the vertex cover of size ≤ 17. This takes $O(n)$ but note the constant is roughly 2^{17}.

This algorithm is clever but was not known for a long time. I would like to tell you that the Graph-Minor-Theorem-algorithm came first, and once it was known to be in far less than $O(n^{17})$ people were inspired and thus found the clever algorithm. However, the actually history is murkier than that. Oh well.

The best known algorithm for VC_k is due to Chen, Kanj, and Jia [15] and runs in time $O(1.2738^k + kn)$.

3) For P $\neq$ NP

Let us step back and ponder how one makes conjectures that are reasonable.

Do Popperian experiments. Karl Popper [52] proposed that scientists should set up experiments that could disprove their theories. That is, experiments that can actually fail. Their failure to fail gives you more evidence in your conjecture. I do not know how one can do this for P vs NP. This would be an interesting approach to P vs NP; however, it is not clear how you would set up such experiments.

Paradigms. Thomas Kuhn [36] proposed that scientists operate within a paradigm and try to fit everything into that paradigm. Great science happens when you have enough evidence for the paradigm to shift. However, most of the time the paradigm is fine. If a theory fits

well into a paradigm, that cannot be ignored. (I do realize that if you take this too seriously you may end up with group-think). With regard to P vs NP we *do* know what theorists believe in a more precise way than usual. There have been two polls taken. In 2002 around 60% of all theorists believed P $\neq$ NP [25] and in 2012 around 80% of all theorists believed P $\neq$ NP [26]. Whether or not you see that as evidence is a matter of taste. We will mention this poll later in Section 9.2.
3) Explanatory power. If a theory explains much data then perhaps the theory is true. This is how evolution is verified. It would be hard to do experiments; however, given Fossil and DNA evidence, evolution seems to explain it pretty well. (I know that it's not as simple as that.) Are there a set of random facts that P $\neq$ NP would help explain? Yes.

The obvious one: P $\neq$ NP explains why we have not been able to solve all of those NP-complete problems any faster!

More recent results add to this:

1. Chvatal [16] in 1979 showed that there is an algorithm for Set Cover that returns a cover of size $(\ln n) \times OPT$ where OPT is the best one could do.

2. Moshkovitz [48] in 2013 proved that, assuming P $\neq$ NP, this approximation cannot be improved.

Why can't we do better than $\ln n$? Perhaps because P $\neq$ NP. If this was the only example it would not be compelling. But there are many such pairs where assuming P $\neq$ NP would explain why we have approached these limits.

4) For P = NP**:**

Fool me once, shame on you, fool me twice, shame on me. There have been surprises in mathematics and computer science before. And there will be more in the future. We mention one: NSPACE($S(n)$) closed under complementation. While this is not really an argument for P = NP it is an argument for keeping an open mind.

An intriguing Question: Most people in the theory community think (a) P $\neq$ NP, (b) we are very far from being able to prove this, (c) if P = NP then this might be by an algorithm we can figure out today. I offer the following thought experiment and my answer. You are told that P vs NP has been solved but *you are not told in what direction!* Do you believe:

- Surely P $\neq$ NP has been shown since of course P $\neq$ NP.

- Surely P = NP has been shown since we are nowhere near being able to show anything remotely like P $\neq$ NP. (See Section 9.4 for more on this.)

Personally I would think P = NP was shown.

9.2 NP Intermediary Problems

Are there any natural problems in NP $-$ P that are not NP-complete? Such sets are called *intermediary*. If we knew such sets existed then we would have P $\neq$ NP. Are there any candidates for intermediary sets?

Ladner [37] showed in 1975 that if P $\neq$ NP then there is an intermediary set. While this is good to know, the set is not natural.

We now give natural problems that may be intermediary.

Example 9.13 Factoring Consider the set

$$FACT = \{(n, m) \mid (\exists a \leq m)[m \text{ divides } n]\}.$$

1. $FACT$ is clearly in NP. There is no known polynomial time algorithm for $FACT$. There is no proof that $FACT$ is NP-complete. If $FACT$ is in P then this could probably be used to crack many crypto systems, notably RSA. Hence the lack of a polytime algorithm is not from lack of trying.

2. Using the unique factorization theorem one can show that $FACT$ is in co-NP. Hence if $FACT$ is NP-complete then NP = co-NP. Hence most theorists think $FACT$ is not NP-complete.

3. The best known algorithm for factoring n is the Number Field Sieve due to Pollard (see [51] for the history) and runs in time $O(\exp(c(\log n)^{1/3}(\log\log n)^{2/3}))$ where $c = (\frac{32}{9})^{1/3} = 1.922999\ldots$. Note that the length of the input is $\log n$ so this algorithm runs in time roughly $2^{O(L^{1/3})}$ where L is the length of the input. This is still exponential but still better than $2^{O(L)}$.

4. Peter Shor [63] proved that $FACT$ is in Quantum-P. Some people think this is evidence that $FACT$ is easier than we thought, perhaps in P. Others think that its evidence that quantum computers can do things that are not in P.

5. In the poll [26] about P vs NP, respondents were also asked to comment on other problems. Of the 21 who commented on factoring 8 thought it is in P and 13 thought it is not in P.

6. Gary Miller and others have said: *Number theorists think factoring is in* P*, whereas cryptographers hope factoring is not in* P.

Example 9.14 The Discrete Log Problem Let p be a prime. Let g be such that, calculating mod p,

$$\{g^0, g^1, g^2, \ldots, g^{p-2}\} = \{1, 2, 3, \ldots, p-1\}$$

(This is a set inequality. We are not saying that $g^0 = 1$, $g^1 = 2$, etc.)

Given a number $x \in \{1, \ldots, p-1\}$ we want to know the unique z such that $g^z \equiv x \pmod{p}$. Note that p, g, x are given in binary so their lengths are bounded by $\log_2 p$. Hence we want to find z in time poly in $\log_2 p$.

Consider the set

$$DL = \{(p, g, x, y) \mid (\exists z \le y)[g^z \equiv x \pmod{p}]\}.$$

1. DL is in NP. (There is one non-obvious part of this: verifying that g is a generator.) There is no known polynomial time algorithm for DL. There is no proof that DL is NP-complete. If DL is in P then this could probably be used to crack many crypto systems, notably Diffie-Helman. Hence the lack of a polytime algorithm is not from lack of trying.

2. DL is in co-NP. Hence if DL is NP-complete then NP $=$ co-NP which is unlikely. Hence most theorists think DL is not NP-complete.

3. There are several algorithms for finding the discrete log that take time $O(\sqrt{p})$. See the Wikipedia Entry on Discrete Log for a good overview.

4. Peter Shor [63] proved that DL is in Quantum-P.

5. I have not heard much talk about this problem. In particular, nobody commented on it for the poll.

Note 9.15 (This note is purely speculative. I am invoking the definition of an intellectual: *One who is an expert in one area and pontificates in another.*) Since factoring and discrete log are important for national security I used to say things like *factoring is not known to be in Polynomial time, or maybe that's just what the NSA wants us to think!*. However, one thing I glean from reading about the Snowden leaks is that the NSA seems more interested in bugging your computer *before* you you encrypt a message, and convincing you to use keys that aren't long enough to be secure, than it is in hard number theory.

The sociology of research in crypto has changed enormously in the last 50 years. At one time only the NSA worked on it, so they could be way ahead of academia and the private sector. Now many academics, private labs, and businesses work on it. Hence the NSA cannot be too far ahead. They can read the papers that academics write so they can keep pace. But they cannot talk to people outside of NSA (and perhaps not even to people inside NSA) about what they do, which may be a hindrance.

Hence I no longer say anything hinting that the NSA may have solved these problems. Nor do I think they have a quantum computer in their basement.

Note again that this is all speculative.

Example 9.16 Graph Isomorphism

$$GI = \{(G_1, G_2) \mid G_1 \text{ and } G_2 \text{ are isomorphic } \}.$$

1. GI is clearly in NP. There is no known polynomial time algorithm for it. There is no proof that it is NP-complete.

2. Even though it has no immediate application there has been much work on it. The following special cases are known to be in P: (1) if there is a bound on the degree, (2) if there is a bound on the genus, (3) if there is a bound on the multiplicity of the eigenvalues for the matrix that represents the graph. There have been connections to group theory as well.

3. There is an algorithm, due to Luks [43], that runs in time $2^{O(\sqrt{n}\log n)}$.

4. There is an algorithm, due to Babai [8], runs in time $2^{(\log n)^{O(1)}}$.

5. If GI is NP-complete then $\Sigma_2^{\rm p} = \Pi_2^{\rm p}$ (see Section 10 for the definition). Hence most theorists think GI is not NP-complete.

6. In the poll [26] about P vs NP respondents were also asked to comment on other problems. Of the 21 who commented on Graph Isomorphism (they were not the same 21 who commented on factoring) 14 thought it was in P and 8 thought it was not in P.

7. I give my opinion: Someone will prove P $\neq$ NP between 200 and 400 years from now; however, we will still not know if GI is in P. I pick this opinion not because it's the most likely but because its the most bizarre.

Example 9.17 Group isomorphism You are given representations of elements $g_1, \ldots, g_n$ and $h_1, \ldots, h_n$ you are also given two $n \times n$ tables, one that tells you, for all i, j what $g_i * g_j$ is and one that tells you, for all i, j, what $h_i * h_j$ is. First check if both tables are for groups (there is an identity element, every element has an inverse, and $*$ is associative). This can be done in polynomial time. The real question is then: Are the two groups isomorphic? We call this problem GPI.

1. GPI is clearly in NP. There is no known polynomial time algorithm for it. There is no proof that it is NP-complete.

2. A long time ago Lipton, Tarjan, and Zalcstein observed that this problem is in time $n^{\log_2 n + O(1)}$ (they never published it but see [40]). Hence if GPI is NP-complete then everything in NP would be in time $n^{O(\log n)}$. This seems unlikely though not as devastating as P = NP. Rosenbaum [60] in 2013 obtained a better algorithm for GPI that runs in time $n^{0.5\log_2 n + O(1)}$. This was rather difficult. Lipton is quite impressed with it (see the citation above).

Example 9.18 Grid Coloring: Imagine coloring every point in the 5×5 grid (formally all points (i,j) where $1 \le i,j \le 5$). A *monochromatic rectangle* (henceforth mono-rect) are four points that form a rectangle (e.g., $(2,2)$, $(2,5)$, $(4,2)$, $(4,5)$) that are all the same color. The following is known [21]: For all c there exists n such that for all c-colorings of the $n \times n$ grid there exists a mono-rect. How big does n have to be? We call a grid c-colorable if you can color it with c colors and not get any mono-rects.

Consider the following set

$$GRID = \{(n,c) \mid \text{ The } n \times n \text{ grid is } c\text{-colorable } \}.$$

This set seems to be in NP. But it is not. The input (n,c) is of size $\log n + \log c$ since they are written in binary. The witness is a c-coloring of $n \times n$ which is of size roughly cn^2. This witness is of size exponential in the input size.

We get around this problem by writing n, c in unary.

$$GRIDUNARY = \{(1^n, 1^c) \mid \text{ The } n \times n \text{ grid is } c\text{-colorable } \}.$$

This problem is in NP. Is it NP-complete? This is unlikely since the set is sparse (see definition below).

Def 9.19 A set $S \subseteq \Sigma^*$ is *sparse* if there exists a polynomial p such that $(\forall n)[|S \cap \Sigma^n| \le p(n)]$. Note that this is a good notion of a skinny set since $S \cap \Sigma^n$ could be as large as 2^n.

Mahaney in 1982 [44] proved that if a sparse set is NP-complete then P = NP. Hence it is unlikely that $GRIDUNARY$ is NP-complete. Even so, $GRIDUNARY$ is believed to be hard.

Consider the following non-sparse variant of the problem: $GRIDEXT$ is the set of all $(1^n, 1^c, \rho)$ such that

- ρ is a partial c-coloring of the $n \times n$ grid.
- ρ can be extended to a c-coloring of the entire grid.

$GRIDEXT$ was shown to be NP-complete by Apon, Gasarch, and Lawler [7].

$GRIDUNARY$ and $GRIDEXT$ are examples of problems in Ramsey theory. Most of them have this same property: they seem to be hard, the natural version is sparse (hence unlikely to be NP-complete), but the version where you have a partial coloring is NP-complete.

9.3 Have We Made Any Progress on P vs NP?

No.

9.4 Seriously, Can you give a more enlightening answer to *Have We Made Any Progress on* P *vs* NP*?*

1. There have been strong (sometimes matching) lower bounds on very weak models of computation. Yao [81] showed (and later Hastad [31, 30] had an alternative proof) that PARITY of n bits cannot be computed with an AND-OR-NOT circuit that has a polynomial number of gates and constant depth. Smolensky [66] extended this (with an entirely different proof) to include Mod m gates where m is a power of an odd prime [66].

2. Let ACC be the class of functions that can be computed with a polynomial number of gates and constant depth where we allow AND, OR, NOT and MOD m gates (they return 0 if the sum of the inputs is $\equiv 0 \pmod{m}$ and 1 otherwise). In 2014 Williams [79] showed that ACC does not contain $\mathrm{NTIME}(2^{n^{O(1)}})$. This is an impressive achievement. This makes one pause to think how much we have to do to get $\mathrm{P} \neq \mathrm{NP}$.

3. There have been some weak lower bounds on space-bounded models of computation. Ryan Williams [77, 78], proved that (essentially) if your machine has very little space to work with then SAT requires $n^{1.8019377\cdots}$ where the exponent approaches $2\cos(2\pi/7)$ as the space goes down. Buss and Williams [13] later proved that the techniques used could not yield a better lower bound.

4. There are proofs that certain techniques will not suffice. These include techniques from computability theory [9], current methods with circuits [58], and a hybrid of the two [4].

5. Ketan Mulmuley has devised a research program called *Geometric Complexity Theory* which, to it credit, recognizes the obstacles to proving $\mathrm{P} \neq \mathrm{NP}$ and *seems* to have the *potential* to get around them. Ketan himself says the program will take a long time- not within his lifetime. For an overview see [49] and other papers on his website.

9.5 So You Think You've Settled P versus NP

The following is Lance Fortnow's blog post from January 14, 2009, see `blog.computationalcomplexity.org/2009/01/so-you-think-you-settled-p-vs-np.html` which is titled

So You Think You've Settled P versus NP

1. You are wrong. Figure it out. Sometimes you can still salvage something interesting out of your flawed proof.

2. You believe your proof is correct. Your belief is incorrect. Go back to step 1.

3. Are you making any assumptions or shortcuts, even seemingly small and obvious ones? Are you using words like "clearly", "obviously", "easy to see", "should", "must", or "probably"? You are claiming to settle perhaps the more important question in all of mathematics. You don't get to make assumptions. Go back to step 1.

4. Do you really understand the P versus NP problem? To show P $\neq$ NP you need to find a language L in NP such that for every k and every machine M running in time n^k ($n =$ input length), M fails to properly compute L. L is a set of strings. Nothing else. L cannot depend on M or k. M can be *any* program that processes strings of bits. M may act differently than one would expect from the way you defined L. Go back to step 1.

5. You submit your paper to an on-line archive. Maybe some people tell you what is missing or wrong in your paper. This should cause you to to to step 1. But instead you make a few meaningless changes to your paper and repost.

6. Eventually people ignore your paper. You wonder why you aren't getting fame and fortune.

7. You submit your paper to a journal.

8. The paper is rejected. If you are smart you would go back to step 1. But if you were smart you would never have gotten to step 7.

9. You complain to the editor that either the editor doesn't understand the proof of that it is easily fixed. You are shocked a respectable editor would treat your paper this way.

10. You are convinced "the establishment" is purposely suppressing your paper because our field would get far less interesting if we settle P versus NP so we have to keep it open at all costs.

11. If I tell you otherwise would you believe me?

9.6 Eight Signs a Claimed P $\neq$ NP Proof is Wrong

In 2010 Vinay Deolalikar claimed to have a proof that P $\neq$ NP. After much discussion, some of it in blogs, the proof is now thought to be incorrect and not even close to a real proof. See the first chapter of Lipton and Reagan's book [42] for a full account. The incident inspired Scott Aaronson to post a blog on

Eight Signs a Claimed P $\neq$ NP Proof is Wrong

which can be found here:

`www.scottaaronson.com/blog/?p=458`

Below are the eight signs, followed by some comments from me on the signs. Note that they are written in Scott's voice. So if it reads *every attempt I've ever seen* ... it means every attempt Scott has ever seen.

1. The author can't immediately explain why the proof fails for 2SAT, XOR-SAT, or other slight variants of NP-complete problems that are known to be in P. Historically, this has probably been the single most important "sanity check" for claimed proofs that P $\neq$ NP: in fact, I'm pretty sure that every attempt I've ever seen has been refuted by it.

2. The proof doesn't "know bout" all known techniques for polynomial time algorithms, including dynamic programming, linear and semidefinite programming, and holographic algorithms. This is related to sign 1, but is much more stringent. Mulmuley's GCT (Geometric Complexity Theory) program is the only approach to P vs. NP I've seen that even has serious aspirations to "know about" lots of nontrivial techniques for solving problems in P (at the least, matching and linear programming). For me, that's probably the single strongest argument in GCT's favor.

3. The paper doesn't prove any weaker results along the way: for example P $\neq$ PSPACE, NEXP $\not\subseteq$ P/poly, NP $\not\subseteq$ TC^0, permanent not equivalent to determinant by linear projection, SAT requires superlinear time .... P vs. NP is a staggeringly hard problem, which one should think of as being dozens" of steps beyond anything that we know how to prove today. So then the question arises: forget steps 30 and 40, what about steps 1,2, and 3?

4. Related to the previous sign, the proof doesn't encompass the *known* lower bound results as special cases. For example: where, inside the proof, are the known lower bounds against constant-depth circuits? where's Razborov's lower bound against monotone circuits? Where's Raz's lower bound against multilinear formulas? All these things (at least the uniform version of them) are implied by P $\neq$ NP, so any proof of P $\neq$ NP should imply them as well. Can we see more-or-less explicitly why it does so?

5. The paper lacks the traditional lemma-theorem-proof structure. This sign was pointed out (in the context of Deolalikar's paper) by Impagliazzo. Say what you like about the lemma-theorem-proof structures, there are excellent reasons why it's used— among them that, exactly like modular programming, it enormously speeds up the process of finding bugs.

6. The paper lacks a coherent overview, clearly explaining how and why it overcomes the barriers that foiled previous attempts. Unlike most P $\neq$ NP papers, Deolalikar's *does* have an informal overview (and he recently released a separate synopsis). But reading the overview felt like reading Joseph Conrad's *Heart of Darkness*: I've reread the same paragraph over and over because the words would evaporate before they could stick to my brain. Of course, maybe that just means I was too dense to understand the argument, but the fact that I couldn't form a mental image of how the proof was supposed to work wasn't a promising sign.

7. The proof hinges on subtle issues in descriptive complexity. Before you reach for your axes: descriptive complexity is a beautiful part of TCS, full of juicy results and open

problems, and I hope that someday it might even prove useful for attacking the great separation questions. Experience has shown, however, that descriptive complexity is also a powerful tool for fooling yourself into thinking you've proven things you haven't. The reason for this seems to be that subtle differences in encoding schemes—for example whether you do or don't have an order relation- can correspond to *huge* differences complexity. As soon as I saw how heavily Deolalikar's proof relied on descriptive complexity, I guessed that he probably made a mistake in applying the results from that field that characterize complexity classes like P in terms of first-order logic. I'm almost embarrassed to relate this guess, given how little actual understanding went into it. Intellectual honesty does, however, compel me to point out that it was correct.

8. Already in the first draft the author waxes philosophically about meaning of his accomplishments, profusely thanks those who made it possible, etc. He says things like "confirmations have already started coming in." To me, this sort of overconfidence suggests a would-be P $\neq$ NP prover who hasn't grasped the sheer number of mangled skeletons and severed heads that line his path.

I agree with all of Scott's signs. Sign 1 I have used to debunk a paper that claimed to show that P $\neq$ NP. The paper claimed to show that the HAM is not in P; however, the techniques would also show that $EULER$ is not in P. Since $EULER$ actually IS in P, the proof could not be correct. Not that I thought it had any chance of being correct anyway. Lance Fortnow has an easier sign: any proof that claims to resolve P vs NP is just wrong.

Scott uses the male pronoun *He*. This could be because there is no genderless pronoun in English; however, I also note that I have never known a female to claim to have a proof of P $\neq$ NP. Perhaps they know better.

9.7 How to Deal with Proofs that P = NP

Alleged proofs that P = NP are usually code or an algorithm that the author claims works *most of the time*. If its a program for SAT then the following class of formulas will likely take it a long time and thus disprove the authors claim.

First some preparation. The following seems obvious and indeed is obvious: If you try to put $n+1$ items into n boxes then one of the boxes will have 2 items. It is often referred to as the *Pigeon Hole Principle for* n, or PHP_n.

We write the negation of PHP_n as a Boolean formula. The items are $\{1, 2, \ldots, n+1\}$. The boxes are $\{1, 2, \ldots, n\}$. The Boolean variable x_{ij} is TRUE if we put it item i into box j. Consider the formula that is the AND of the following:

1. For each $1 \le i \le n+1$ $x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}$. This says that each item is in some box.

2. For each $1 \le i_1 < i_2 \le n+1$ and $1 \le j \le n$ $\neg(x_{i_1 j} \wedge x_{i_2 j})$ This says that no box has two items.

The Boolean formula $\neg PNP_n$ is not satisfiable. How would one show that? One way is to list out the truth table. This is of course quite long. It is know that in some logical systems this is the best you can do. While these systems are weak, it is likely that the P = NP guy is essentially using one of those systems. So challenge him to run his system on say PHP_{20}. That will shut him up and get him out of your hair.

9.8 A Third Category

I have also gotten papers that claim to resolve P vs. NP but from what they write you cannot tell in what direction. Some hint that its the wrong problem or that its model dependent or that its independent of Set Theory; however, even ascribing those aspirations is being generous in that such papers are incoherent.

10 PH: The Polynomial Hierarchy

We want to generalize the definition of NP. We first need a better notation.

Def 10.1

1. $(\exists^p y)[B(x,y)]$ means that there exists a polynomial p such that $(\exists y, |y| = p(|x|)[B(x,y)]$.
2. $(\forall^p y)[B(x,y)]$ means that there exists a polynomial p such that $(\forall y, |y| = p(|x|)[B(x,y)]$.

With this notation we define NP again.

Def 10.2 $A \in$ NP if there exists a set $B \in P$ such that

$$A = \{x \mid (\exists^p y)[(x,y) \in B]\}.$$

Why stop with one quantifier?

Def 10.3

1. $A \in \Sigma_1^{\rm p}$ if there exists a set $B \in P$ such that

$$A = \{x \mid (\exists^p y)[(x,y) \in B]\}.$$

   This is just NP.

2. $A \in \Pi_1^{\rm p}$ if $\overline{A} \in \Sigma_1^{\rm p}$. This is just co-NP.
3. $A \in \Sigma_2^{\rm p}$ if there exists a set $B \in P$ such that

$$A = \{x \mid (\exists^p y)(\forall^p z)[(x,y,z) \in B]\}.$$

4. $A \in \Pi_2^{\mathrm{p}}$ if $\overline{A} \in \Sigma_2^{\mathrm{p}}$.

5. $A \in \Sigma_3^{\mathrm{p}}$ if there exists a set $B \in P$ such that

$$A = \{x \mid (\exists^p y)(\forall^p z)(\forall w)[(x, y, z, w) \in B]\}.$$

6. $A \in \Pi_3^{\mathrm{p}}$ if $\overline{A} \in \Sigma_3^{\mathrm{p}}$.

7. One can define $\Sigma_4^{\mathrm{p}}, \Pi_4^{\mathrm{p}}, \Sigma_5^{\mathrm{p}}, \Pi_5^{\mathrm{p}}, \ldots$.

8. These sets form what is called *the Polynomial Hierarchy.* We define $PH = \bigcup_{i=1}^{\infty} \Sigma_i^{\mathrm{p}} = \bigcup_{i=1}^{\infty} \Pi_i^{\mathrm{p}}$.

Clearly

$$\Sigma_1^{\mathrm{p}} \subseteq \Sigma_2^{\mathrm{p}} \subseteq \Sigma_3^{\mathrm{p}} \cdots$$

and

$$\Pi_1^{\mathrm{p}} \subseteq \Pi_2^{\mathrm{p}} \subseteq \Pi_3^{\mathrm{p}} \cdots .$$

and

$$(\forall i)[\Pi_i^{\mathrm{p}} \subseteq \Sigma_{i+1}^{\mathrm{p}} \text{ and } \Sigma_i^{\mathrm{p}} \subseteq \Pi_{i+1}^{\mathrm{p}}].$$

These containments are not known to be proper. If there is an i such that $\Sigma_i^{\mathrm{p}} = \Pi_i^{\mathrm{p}}$ then $(\forall j \geq i)[\Sigma_j^{\mathrm{p}} = \Sigma_i^{\mathrm{p}}]$. In this case we say PH *collapses.* Most theorists think that PH does not collapse.

Clearly NP $\subseteq$ PH and R $\subseteq$ PH. What about BPP? Since most theorists think P = R = BPP, most theorists think BPP $\subseteq$ PH. But is it not even clear that BPP $\subseteq$ PH. However, Sipser [64] obtained BPP $\subseteq \Sigma_2^{\mathrm{p}} \cap \Pi_2^{\mathrm{p}}$ by developing a new theory of time-bounded Kolmogorov complexity, and shortly thereafter, Lautemann [38] proved the same containment with a very clever trick. One might think *Oh, so a problem can be open for a long time and then all of a sudden it's solved. Maybe* P *vs* NP *will go that way.* However, I am skeptical of this notion. For clever algorithms and clever collapses of classes that has happened, but never for a separation of classes.

The following are examples of natural problems that are in these various levels of PH.

Example 10.4 This will just be a rewriting of the SAT problem. QBF stands for *Quantified Boolean Formula.* $\phi(\vec{x})$ will be a Boolean Formula.

$$QBF_1 = \{\phi(\vec{x}) \mid (\exists \vec{b})[\phi(\vec{b}) = TRUE]\}.$$

QBF_1 is Σ_1^{p}-complete and hence unlikely to be in Π_1^{p}. This is just a fancy way of saying that SAT is NP-complete and hence unlikely to be in co-NP.

Example 10.5 $\phi(\vec{x},\vec{y})$ means there are two sets of variables that are distinguished.

$$QBF_2 = \{\phi(\vec{x},\vec{y}) \mid (\exists\vec{b})(\forall\vec{c})[\phi(\vec{b},\vec{c}) = TRUE]\}.$$

QBF_2 is $\Sigma_2^{\rm p}$-complete and hence unlikely to be in $\Pi_2^{\rm p}$.

Example 10.6 One can define QBF_i. QBF_i is $\Sigma_i^{\rm p}$-complete and hence unlikely to be in $\Pi_i^{\rm p}$.

Example 10.7 Boolean Formula Minimization. Given a Boolean Formula ϕ, is there a shorter equivalent Boolean Formula? Let

$$MIN = \{\phi(\vec{x}) \mid (\forall\psi(\vec{x}), |\psi(x)| < |\phi(x)|])(\exists\vec{b})[\phi(\vec{b}) \neq \psi(\vec{b})]\}.$$

Clearly $MIN \in \Pi_2^{\rm p}$. It is believed to not be $\Pi_2^{\rm p}$-complete but to also not be in $\Sigma_1^{\rm p}$ or $\Pi_1^{\rm p}$. See the paper of Buchfuhrer and Umas [19] for more information.

11 #P

Leslie Valiant defined #P and proved most of the results in this section [72, 73].

Def 11.1 A function f is in #P if there is a nondeterministic program M that runs in polynomial time such that $f(x)$ is the number of accepting paths in the $M(x)$ computation. A set A is in $\mathrm{P}^{\#\mathrm{P}}$ if membership of $x \in A$ can be determined by a program in poly time that can ask questions to a #P function.

When #P was first defined it was not clear if it was powerful. Clearly NP $\subseteq \mathrm{P}^{\#\mathrm{P}}$ but it was not clear if $\Sigma_2^{\rm p} \subseteq \mathrm{P}^{\#\mathrm{P}}$. However, Toda [71] proved the somewhat surprising result that PH $\subseteq \mathrm{P}^{\#\mathrm{P}}$. It is not know if this containments is proper. If PH $= \mathrm{P}^{\#\mathrm{P}}$ then PH collapses, hence most theorists think PH $\subset \mathrm{P}^{\#\mathrm{P}}$.

We give examples of natural problems in #P.

Example 11.2 Let $f(\phi)$ be the number of satisfying assignments of ϕ. This problem is clearly in #P. Of more importance is that its #P-complete and hence unlikely to be computable in PH.

Example 11.3 For most NP-complete problems the function that returns the number of solutions (e.g., the number of Hamiltonian cycles) is #P-complete.

Example 11.4 There are some problems in Polynomial time where finding the number of solutions is #P-complete. In particular (1) finding the number of matchings in a graph, and (2) finding the number of Eulerian cycles in a directed graph, are #P-complete. Strangely enough, finding the number of Eulerian cycles in an undirected graph can be done in polynomial time.

Example 11.5 The *Permanent* of a matrix is just like the determinant but without the negative signs. Valiant's motivation was as follows: computing the determinant is easy (polynomial time), but computing the permanent seemed hard. Valiant showed that computing the permanent is #P-complete and hence likely quite hard.

12 PSPACE

Def 12.1 PSPACE is the set of problems that can be solved using space bounded by a polynomial in the length of the input. Formally PSPACE = DSPACE($\mathrm{n}^{\mathrm{O}(1)}$). By Theorem 3.3.1 PSPACE = NSPACE($\mathrm{n}^{\mathrm{O}(1)}$.

Clearly $\mathrm{P}^{\#\mathrm{P}} \subseteq$ PSPACE. It is not known if this inclusion is proper; however, if $\mathrm{P}^{\#\mathrm{P}} =$ PSPACE then PH collapses. Hence most theorists think $\mathrm{P}^{\#\mathrm{P}} \neq$ PSPACE.

The following problems are PSPACE-complete. Hence the are in PSPACE and unlikely to be in $\mathrm{P}^{\#\mathrm{P}}$.

Example 12.2 Given two regular expressions, are they equivalent? Formally

$$REGEXPEQUIV = \{(\alpha, \beta) \mid L(\alpha) = L(\beta)\}.$$

(α and β are regular expressions.)

Example 12.3 HEX is a simple two-player game. Given a position, determining if the player whose move it is wins. Note that we allow any sized board.

Example 12.4 GO is a popular game in Japan and China. There are several versions. Given a position (on an $n \times n$ board) determine if the player whose move it is wins the ko-free version. (The version with ko-rules is EXPTIME complete.)

13 EXPTIME

Def 13.1 EXPTIME = DTIME($2^{n^{O(1)}}$).

The following problems are in EXPTIME-complete and hence not in P.

Example 13.2 Generalized Chess. Given an $n \times n$ chess board with pieces on it, does the player whose move it is win?

Example 13.3 Generalized Checkers. Given an $n \times n$ checker board with pieces on it, does the player whose move it is win?

Example 13.4 Generalized Go (with Japanese Ko rules). Given an $n \times n$ Go board with pieces on it, does the player whose move it is win, playing Japanese Ko rules?

14 EXPSPACE = NEXPSPACE

Def 14.1 EXPSPACE = DSPACE($2^{n^{O(1)}}$). By Theorem 3.3.1 EXPSPACE = NSPACE($2^{n^{O(1)}}$).

Clearly EXPTIME $\subseteq$ EXPSPACE. It is not known if this inclusion is proper; however, most theorists think EXPTIME $\neq$ EXPSPACE. By Theorem 3.5 PSPACE $\subset$ EXPSPACE.

We present a natural problem that is NEXPSPACE-complete and hence not in PSPACE. The statement is due to Meyer and Stockmeyer [47].

In textbooks one often sees expressions like a^5b^2. These are not formally regular expressions; however, there meaning is clear and they can be rewritten as such: *aaaaabb*. The difference in representation matters. If we allow exponents then Regular Expressions can be represented far more compactly. Note that a^n is written in $O(\log n)$ space, where as $aaa \cdots a$ (n times) takes $O(n)$ space.

Def 14.2 Let Σ be a finite alphabet. A *Textbook Regular Expression* (henceforth t-Reg Exp) is defined as follows.

- For all $\sigma \in \Sigma$, σ is a t-reg exp.
- $\emptyset$ is a t-reg exp
- If α and β are t-reg exps then so is $\alpha \cup \beta$, $\alpha\beta$ and α^*
- If α is a t-reg exp and $n \in \mathbb{N}$ then α^n is a t-reg exp.

If α is a t-reg exp then $L(\alpha)$ is the set of strings that α generates.

Here is the question which we call *t-reg expression equivalence*

$$TRE = \{(\alpha, \beta) \mid \alpha, \beta \text{ are t-reg expressions and } L(\alpha) = \mathrm{L}(\beta)\}.$$

Note 14.3 In the original paper this is called *Regular expression with squaring.* They originally had a formulation like mine but since people thought maybe they were coding things into bits (they weren't) they changed the name. Frankly I think the formulation of t-reg exp is more natural.

Meyer and Stockmeyer showed that TRE is NEXPSPACE-complete and hence not in PSPACE. Note that it is also not in P. Is it natural? See Section 25 for a discussion of that issue.

15 DTIME($TOW_i(n)$)

Def 15.1

1. $TOW_0(n) = n$
2. For $i \ge 1$ let $TOW_i(n) = 2^{TOW_{i-1}(n)}$.

By Theorem 3.4 we have that, for all i, DTIME($TOW_i(n^{O(1)})$) $\subset$ DTIME($TOW_{i+1}(n^{O(1)})$).

We give a natural problem that is in DTIME($TOW_3(n)$ and requires at least $2^{2^{cn}}$ time for some constant c. Its exact complexity is known but is somewhat technical.

The problem will be given a set of sentences in a certain restricted mathematical langauge, determine if it's true. We need to define the language.

We will only use the following symbols.

1. The logical symbols $\wedge$, $\neg$, $(\exists)$.
2. Variables $x, y, z, \ldots$ that range over $\mathbb{N}$.
3. Symbols: $=$, $<$, $+$
4. Constants: 0,1,2,3,....

We call this *Presburger Arithmetic* in honor of the man who proved it was decidable.

Def 15.2 A *term* is:

1. If t is a variable or a constant then t is a term.
2. If t_1 and t_2 are terms then $t_1 + t_2$ is a term.

Def 15.3 An *Atomic Formulas* is:

1. If t_1, t_2 are terms then $t_1 = t_2$ is an Atomic Formula.
2. If t_1, t_2 are terms then $t_1 < t_2$ is an Atomic Formula.

Def 15.4 A *Presburger Formula* is defined similar to how a WS1S formula was defined, given that we have defined Atomic formulas.

Is $x < y + z$ true? This is a stupid question since we don't know what x, y, z are. But if we quantify over all of the variables then a truth value exists. For example

$$(\exists x)(\exists y)(\exists z)[x < y + z] \text{ is true}$$

$$(\exists x)(\exists y)(\forall z)[x < y + z] \text{ is true}$$

$$(\exists x)(\forall y)(\exists z)[x < y + z] \text{ is true}$$

$$(\exists x)(\forall y)(\forall z)[x < y + z] \text{ is false}$$

$$(\forall x)(\exists y)(\exists z)[x < y + z] \text{ is true}$$

$$(\forall x)(\exists y)(\forall z)[x < y + z] \text{ is true}$$

$$(\forall x)(\forall y)(\exists z)[x < y + z] \text{ is true}$$

$$(\forall x)(\forall y)(\forall z)[x < y + z] \text{ is false}$$

A *sentence* is a formula where all of the variables are quantified over. We can now (finally!) define our problem: Given a sentence ϕ in Presburger arithmetic is it true?

- Presburger proved that this problem is decidable. His proof did not yield time bounds.
- Later a proof was found that involved quantifier elimination. Given a sentence we can find an equivalent one with one less quantifier. This algorithm puts this problem in DTIME($TOW_3(n)$).
- Fisher and Rabin showed that there exists a constant c such that this problem requires time at least $2^{2^{cn}}$.

16 DSPACE($TOW_i(n^{O(1)})$)

By Theorem 3.5 we have that, for all i, DSPACE($TOW_i(n^{O(1)})$) $\subset$ DSPACE($TOW_{i+1}(n^{O(1)})$). For each i we give an example that is arguably natural. It is a variant of the problem TRE from Section 14.

Def 16.1 Let Σ be a finite alphabet. Let $i \in \mathbb{N}$. An *i-Textbook Regular Expression* (henceforth i-t-Reg Exp) is defined as follows.

- For all $\sigma \in \Sigma$, σ is an i-t-reg exp.
- $\emptyset$ is a i-t-reg exp
- If α and β are i-t-reg exps then so is $\alpha \cup \beta$, $\alpha\beta$ and α^*

- If α is an i-t-reg exp and $n,k \in \mathbb{N}$ then $\alpha^{TOW_i(n^k)}$ is an i-t-reg exp.

Here is the question which we call *i-t-reg expression equivalence*

$$TRE = \{(\alpha,\beta) \mid \alpha,\beta \text{ are i-t-reg expressions and } L(\alpha) = \mathrm{L}(\beta)\}.$$

This problem can be proven to be in DSPACE($TOW_i(n^{O(1)})$)−DSPACE($TOW_{i-1}(n^{O(1)})$) similar to the proof of Meyer and Stockmeyer that TRE is not in PSPACE. I believe this is the first time this fact was noted.

17 Elementary

Def 17.1 The complexity class EL (for Elementary) is defined by

$$\mathrm{EL} = \bigcup_{i=0}^{\infty} \mathrm{DTIME}(TOW_i(n)).$$

It is known that, for all i, DSPACE($TOW_i(n)$) $\subset$ EL.

Virtually everything one would ever want to compute is Elementary. In the next section we give an example of a problem which is computable (in fact, primitive recursive) but not elementary.

18 Primitive Recursive

We will define the primitive recursive functions in stages.

Def 18.1 Let PR_0 be the following functions:

1. Let $n,c \in \mathbb{N}$. Then the function $f(x_1,\ldots,x_n) = c$ is in PR_0.
2. Let $n \in \mathbb{N}$ and $1 \le i \le n$. Then the function $f(x_1,\ldots,x_n) = x_i$ is in PR_0.
3. Let $n \in \mathbb{N}$ and $1 \le i \le n$. Then the function $f(x_1,\ldots,x_n) = x_i + 1$ is in PR_0.

Def 18.2 For $i \ge 1$ the following functions are in PR_i.

1. All $h \in \mathrm{PR}_{i-1}$.
2. Let $k,n \in \mathbb{N}$. Let $f \in \mathrm{PR}_{i-1}$ where $f : \mathbb{N}^n \to \mathbb{N}$. Let $g_1,\ldots,g_n \in \mathrm{PR}_{i-1}$ where $g_i : \mathbb{N}^k \to \mathbb{N}$. Then $h(x_1,\ldots,x_k) = f(g_1(x_1,\ldots,x_k),\ldots,g_n(x_1,\ldots,x_k))$ is in PR_i. (This is just composition.)
3. Let $n \in \mathbb{N}$. Let $f,g \in \mathrm{PR}_{i-1}$ where $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$. Let $h : \mathbb{N}^{n+1} \to \mathbb{N}$ be defined by

(a) $h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$

(b) $h(x_1, \ldots, x_n, x+1) = g(x_1, \ldots, x_n, x, h(x_1, x_2, \ldots, x_n, x))$

(This is just recursion.)

Def 18.3 A function is *Primitive Recursive* if it is in $\bigcup_{i=0}^{\infty} \mathrm{PR}_i$. We denote the set of sets in $DTIME(f)$ where f is primitive recursive by PRIMREC.

One can show that addition is in PR_1, multiplication is in PR_2, Exponentiation is in PR_3, $TOW_n(2)$ is in PR_4. More to the point, virtually any function encountered in normal mathematics is primitive recursive.

Clearly EL $\subseteq$ PRIMREC. In fact EL $\subset$ PRIMREC. We give a example of a natural problem that is in PRIMREC but not EL.

Example 18.4 The problem will be given a sentences in a certain restricted mathematical langauge, determine if it's true. We need to define the language.

Recall that in Section 4 we defined WS1S formulas.

A *sentence* is a formula where all of the variables are quantified over. As noted in the discussion of Presburger arithmetic, formulas do not have a truth value but sentences do. We can now define our problem: Given a sentence ϕ in WS1S is it true?

- Buchi [11] showed that this problem is decidable using finite automata. This involves using the fact that formulas give rise to regular sets (see Section 4). Using this method, every time there is an alternation of quantifiers you need to do an NDFA to DFA transformation. Hence this procedure takes roughly $TOW_n(2)$ steps where n is the number of alternations of quantifiers. Therefore the algorithm is primitive recursive; however, since the subscript depends on the input, the function $TOW_n(2)$ is not in EL.

- Meyer [46] showed that the algorithm sketched above is optimal. Hence the problem is not in EL.

- One can define $S1S$ which allows quantification of infinite sets. Buchi [12] showed that this theory is decidable. The proof uses ω-automata which run on infinite strings. In the algorithm for deciding $WS1S$ DFA's are manipulated and tested but never actually ran. So the fact that an ω-automata takes an infinite string as input is not a problem. The proof that $S1S$ is decidable is rather difficult.

- $WS1S$ and $S1S$ both involve having one successor function. What does it mean to have two successors? Our basic objects are numbers. We could view numbers as strings in unary. In that case $S(x) = x1$. If our basic objects were strings in $\{0,1\}^*$ then we could have two successors $S_0(x) = x0$ and $S_1(x) = x1$. This yields two theories: $WS2S$ and $S2S$. Rabin [55] proved that both are decidable. The proofs for $S2S$ used transfinite induction and is likely the hardest proof of a theory being decidable. Easier proofs were later found by Gurevich and Harrington [28, 10].

- How expressive is $WS1S$ (and $S1S$, $WS2S$, $S2S$)? Is having them decidable useful? There are two answers to this.
  - $WS1S$ and $WS2S$ have been coded up and used [1]. Even though these theories are not in EL the coding is very clever and the problems they input to it are not that large. The proof that these theories are difficult produce instances that are hard. These instances are somewhat contrived and do not come up. The application has been to search patters, temporal properties or reactive systems, parse tree constraints. It has not been applied to solving open mathematical conjecture. This us unlikely to happen as $WS1S$ seems unable to express anything of interest mathematically.
  - The decidability of $S1S$ and $S2S$ has been use to prove other theories decidable. We do not know of an implementation of either. It is possible to state interesting theorems in $S2S$ (See [55]). Great! So perhaps we can input to an $S2S$ decider an open question in Mathematics and get the answer! There are two problems with this (1) Coding up $S2S$ would be extremely difficult, and getting it to run quickly might be impossible. (2) *Be careful what you wish for— you might just get it:* Lets say we really did have such a decider and its fast. Lets say we input statements of The Goldbach Conjecture, the Riemann Hypothesis, and P vs NP. Lets say it outputs YES, YES, YES. Then we would know that these are all true. Oh. We already sort of know that. It is not the purpose of math to just establish whats true, but also *why* its true. The hope is that the proof of (say) P $\neq$ NP will give great insight into computation. Just the one bit YES would not.
- According to the last item, even if a theory was decidable this would not be that useful. So why do we want prove a theory is decidable? (1) Hilbert wanted to (in today's terminology) show that mathematics is decidable to give it a rigorous foundation. Even though mathematics is undecidable it is of intellectual interest to see how big a fragment of math is decidable. (2) As the work on $WS1S$ has shown there may be fragments of those fragments that are decidable in good time and can be used elsewhere (though unlikely used for mathematics itself).

We give another example, again a logical theory. We need to define the language.
We will only use the following symbols.

1. The logical symbols $\wedge$, $\neg$, $(\exists)$.
2. Variables $x, y, z, \ldots$ that range over $\mathbb{R}$.
3. Symbols: $=$, $<$, $+$

We call this *Theory of the Reals.*

Def 18.5 A *term* is:

1. If t is a variable t is a term.
2. If t_1 and t_2 are terms then $t_1 + t_2$ and $t_1 t_2$ are terms.

Def 18.6 An *Atomic Formulas* is:

1. If t_1, t_2 are terms then $t_1 = t_2$ is an Atomic Formula.
2. If t_1, t_2 are terms then $t_1 < t_2$ is an Atomic Formula.

Def 18.7 A *Formula* is:

1. Any atomic formula is a Presburger formula.
2. If ϕ_1, ϕ_2 are Presburger formulas then so are
   (a) $\phi_1 \wedge \phi_2$,
   (b) $\phi_1 \vee \phi_2$
   (c) $\neg\phi_1$
   (d) If $\phi(x_1, \ldots, x_n)$ is a formula then so is $(\exists x_i)[\phi(x_1, \ldots, x_n)]$

A *sentence* is a formula where all of the variables are quantified over. We can now (finally!) define our problem: Given a sentence ϕ in the theory of the Reals is it true?

- Tarski [70] showed that this problem is decidable. His proof gave no time bounds.
- There were several different proofs that gave time bounds. Some of the people involved are Seidenberg, Cohen, Collins, Renegar, Heintz, Roy, and Solerno. The papers of Renegar and Heintz-Roy-Solerno both obtain the best known results: time $TOW_2(n)$ where n is the number of quantifier alternations. See [75] for history and details.
- Fisher and Rabin [23] showed that the problem requires time $2^{\Omega(n)}$.

19 Ackermann's Function

We define a somewhat natural computable function that is not primitive recursive.

Note that any primitive recursive function uses the recursion rule some fixed finite number of times. Ackermann's function (below) intentionally uses recursion a non-constant number of times. We note that this is the intuition behind why Ackermann's function is not primitive recursive; however, it is not a proof. The proof involves showing that Ackermann's function grows faster than any primitive recursive function.

Def 19.1 *Ackermann's function* is defined as follows

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m \geq 1 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m \geq 1 \text{ and } n \geq 1 \end{cases} \tag{1}$$

Ackermann's function is an example of a function that is computable but not primitive recursive. We have not been able to find a more natural example. This raises the question: How natural is Ackermann's function.

Ackermann's function was originally defined for the sole purpose of obtaining a computable function that was not primitive recursive. Hence it can be considered unnatural. However, over time they have shown up in natural places. We gave an example of this, the minimial spanning tree problem, in Section 7. We give another one here.

Example 19.2 Data Structures for Union Find

A *Union-Find Data Structure* is a data structure which supports a set of sets. The basic operations are (1) FIND which will, given an item x will determine if it is in the data structure, and if so which set it's in, and (2) UNION given two sets replace them with the union of the two. One could ask how many steps a FIND costs and how many steps a UNION costs. This is not the right question. One anticipates doing many FINDs and UNIONs. So here is the right question: how much time does it take to do n operations? Note that it could be that one of them takes a long time but then many take very little time.

Tarjan and Van Leeuwen [69] showed that this problem (1) can be done in time $O(n\alpha(n))$,and (2) requires time $\Omega(n\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackerman function. This means the problem cannot be done in $O(n)$ time but it can be done in just barely more than that. Of interest to us is that Ackermann's function appears in the matching upper and lower bounds of this natural problem!

Def 19.3 Let ACK = DTIME($A(n)$).

20 The Goodstein Function

We do not define a complexity class in this section. We define a somewhat natural computable function that grows much faster than Ackermann's function.

We first define a function that doesn't grow that fast but contains many of the ideas. We do this by example. Say the input is 213. We write this as $(213)_{10}$ to indicate that the number is in base 10. Keep subtracting 1 from the number but increasing the base by 1 to obtain $(212)_{11}$, $(211)_{12}$, $(210)_{13}$. Now what? Note that if you subtract 1 from 210 in base 13 you get (20(12) where the (12) is a digit. Hence our next number is $(20(12)_{14}$. Keep doing this until you get to $(200)_{26}$. This number is quite large. In base 10 it is 1352, much larger than the 213 we started with. Perhaps this sequence goes to infinity. No, it does not. In fact, note that the next number is $(1(25)(26))_{27}$. After many many iteration the leading 1 becomes a 0. Eventually this process leads to 0. $f(213)$ is the number of iterations of this process you need to get down to 0. This function grows fast but is still primitive recursive.

We now define the Goodstein function. First off, we will begin in base 2 (this is not important but lets us give a real example). We'll again take the input 213. First write it in base 2:

$$213 = 2^7 + 2^6 + 2^4 + 2^2 + 2^0.$$

We now write the exponents in base 2:

$$213 = 2^{2^2+2^1+2^0} + 2^{2^2+2^1} + 2^{2^2} + 2^{2^1} + 2^0.$$

We can stop here since all of the exponents are 0,1, or 2. If they were bigger we would again write them in base 2.

We again subtract 1 but then rather than increase the base we increase all of the bases. So in the next iteration we have

$$3^{3^2+3^1+3^0} + 3^{3^2+3^1} + 3^{3^2} + 3^{3^1}.$$

This process will initially increase but eventually decrease to 0. $f(213)$ is the number of iterations before 0 is reached. This function grows much faster than Ackermann's function.

Is the Goodstein function natural? Goodstein used them to investigate various phenomena in logic. Later Paris and Kirby [34] showed that the statement that the Goodstein function always exists (that is, the process always terminates) cannot be proven in Peano Arithmetic. Hence the Goodstein function is natural *to logicians!* However, since I can explain the function easily, and show it exists easily, and it's fun, I call that natural.

Def 20.1 Let GOOD be DTIME($G(n)$) where $G(n)$ is the function defined above.

21 Decidable, Undecidable and Beyond

Def 21.1 A set A is *Decidable* (henceforth DEC) if there exists a program M such that

1. If $x \in A$ then $M(x)$ outputs YES.
2. If $x \notin A$ then $M(x)$ outputs NO.

Note that there are no time or space bounds.

Clearly all the classes defined so far in this chapter are subsets of DEC.

Are there any problems that are undecidable? That is, are there any problems that no computer can solve. We give two natural ones.

Example 21.2 The Halting Problem: Given a program M and an input x, does $M(x)$ terminate? We write it as a set:

$$HALT = \{(M,x) \mid (\exists s)[\text{ If you run } M(x) \text{ for } s \text{ steps then it will halt }]\}.$$

One attempt to solve $HALT$ is to run $M(x)$; however, if $M(x)$ does not halt you will never know. This failed attempt *is not* a proof that $HALT \notin$ DEC. However it is true: $HALT \notin$ DEC. The proof is in most textbooks on Formal Language theory or computability theory. Alternatively, there is a proof in the style of Dr. Seuss [54].

$HALT$ is natural but it refers to programs. Is there a natural problem that is not in DEC that does not refer to programs? Yes!

Example 21.3 Diophantine Polynomials: Given a polynomial $p(x_1,\ldots,x_n)$ with integer coefficients, does there exist $b_1\ldots,b_n \in \mathbb{N}$ such that $p(b_1,\ldots,b_n)=0$. This problem turns out to be undecidable.

In 1900 David Hilbert, a very prominent mathematician, proposed 23 problems for mathematicians to work on for the next 100 year. Some of the problems were not quite well defined (e.g., *Problem 6: Make Physics Rigorous*) so it's hard to say how many have been solved; however, experts say that about 90% have been solved. See [80] for more information.

Hilbert's tenth problem was the following: given a polynomial $p(x_1,\ldots,x_n)$ with integer coefficients, determine if there exist $b_1\ldots,b_n \in \mathbb{N}$ such that $p(b_1,\ldots,b_n)=0$. To express this as a set

$$H10 = \{p(x_1,\ldots,x_n) \mid (\exists b_1,\ldots,b_n s)[p(b_1,\ldots,b_n)=0]\}.$$

Hilbert thought this problem was a solvable problem in Number Theory. He was incorrect. Two papers together, one by Davis, Putnam, and Robinson [18] and one by Matijasevic [45] showed that $H10 \notin DEC$. They essentially showed that if this could be solved then the Halting problem could be solved.

How do these problems compare to each other? Can there be even harder problems? What does harder mean in this context? For problems of this type we cannot talk about time or space bounds. But we can talk about how easy it is to express them. We can write the halting problems as membership in the following set:

We rewrite $HALT$. Let

$$B = \{((M,x),s) \mid \text{ If you run } M(x) \text{ for } s \text{ steps then it will halt }\}.$$

Note that B is decidable and

$$HALT = \{(M,x) \mid (\exists s)[((M,x),s) \in B]\}.$$

We can write $HALT$ as a there exists quantifier followed by something decidable. This is analogous to writing SAT as a poly-bounded quantifier followed by something in P. As such we can define analogies of PH from Section 10. While this is true mathematically this is false historically. The hierarchy we are about to define came first.

Def 21.4

1. $A \in \Sigma_1$ if there exists a set $B \in$ DEC such that
$$A = \{x \mid (\exists y)[(x,y) \in B]\}.$$
This class is often called *compatibly enumerable* (c.e.) or *recursively enumerable (r.e.)*. Both $HALT$ and $H10$ are Σ_1-complete. Hence they are really the same problem.

2. $A \in \Pi_1$ if $\overline{A} \in \Sigma_1$.

3. $A \in \Sigma_2$ if there exists a set $B \in$ DEC such that
$$A = \{x \mid (\exists y)(\forall^p z)[(x,y,z) \in B]\}.$$

4. $A \in \Pi_2$ if $\overline{A} \in \Sigma_2$.

5. $A \in \Sigma_3$ if there exists a set $B \in$ DEC such that
$$A = \{x \mid (\exists y)(\forall^p z)(\forall w)[(x,y,z,w) \in B]\}.$$

6. $A \in \Pi_3$ if $\overline{A} \in \Sigma_3$.

7. One can define $\Sigma_4, \Pi_4, \Sigma_5, \Pi_5, \ldots$.

8. These sets form what is called *the Arithmetic Hierarchy*. We define $AH = \bigcup_{i=1}^{\infty} \Sigma_i = \bigcup_{i=1}^{\infty} \Pi_i$.

Clearly
$$\Sigma_1 \subseteq \Sigma_2 \subseteq \Sigma_3 \cdots$$
and
$$\Pi_1 \subseteq \Pi_2 \subseteq \Pi_3 \cdots .$$
and
$$(\forall i)[\Pi_i \subseteq \Sigma_{i+1} \text{ and } \Sigma_i \subseteq \Pi_{i+1}].$$
In contrast to PH these containments are known to be proper.

The following are examples of problems that are in these classes. Throughout the examples *poly* means *polynomial with integer coefficients*. The quantifiers are over the natural numbers.

Example 21.5 This will just be a rewriting of the $H10$ problem. QP stands for *Quantified Poly* . $\phi(\vec{x})$ will be a poly.

$$QP_1 = \{\phi(\vec{x}) \mid (\exists \vec{b})[\phi(\vec{b}) = 0]\}.$$

QP_1 is Σ_1-complete and hence not in Π_1.

Example 21.6 $\phi(\vec{x}, \vec{y})$ means there are two sets of variables that are distinguished.

$$QP_2 = \{\phi(\vec{x}, \vec{y}) \mid (\exists \vec{b})(\forall \vec{c})[\phi(\vec{b}, \vec{c}) = 0]\}.$$

QP_2 is Σ_1-complete and hence not in Π_1.

Example 21.7 One can define QP_i. QP_i is Σ_i-complete and hence not in Π_i.

For the next few examples let $M_1, M_2, M_3, \ldots$ be the list of all programs in some reasonable programming langauge. From the index i we should be able to recover the code for the program.

Example 21.8 As noted earlier

$$HALT = \{(M, x) \mid (\exists s)[\text{ If you run } M(x) \text{ for } s \text{ steps then it will halt }]\}$$

is Σ_1-complete and hence not in Π_1.

Example 21.9 Let TOT be the set of program that halt on every input. Formally

$$TOT = \{M \mid (\forall x)(\exists s)[\text{ If you run } M(x) \text{ for } s \text{ steps then it will halt }]\}.$$

TOT is Π_2-complete and hence not in Σ_2. Note that we have not proven this, but it is true.

Example 21.10 Let COF be the set of program that halt on all but a finite set of inputs. Formally

$$COF = \{M \mid (\exists y)(\forall x \geq y)(\exists s)[\text{ If you run } M(x) \text{ for } s \text{ steps then it will halt }]\}.$$

COF is Σ_3-complete and hence not in Π_3. Note that we have not proven this, but it is true.

Are there any natural problems that are not in AH? We give one.

Example 21.11 The problem will be given a set of sentences in a certain restricted mathematical langauge, determine if it's true. We need to define the language.

We will only use the following symbols.

1. The logical symbols $\wedge$, $\neg$, $(\exists)$.
2. Variables $x, y, z, \ldots$ that range over $\mathbb{N}$.
3. Symbols: $+$, $\times$.
4. Constants: $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$.

We call this *Arithmetic.*

Def 21.12 A *term* is:

1. If t is a variable or a constant then t is a term.
2. If t_1 and t_2 are terms then $t_1 + t_2$ is a term and $t_1 \times t_2$ is a term.

Def 21.13 An *Atomic Formulas* is: If t_1, t_2 are terms then $t_1 = t_2$ is an Atomic Formula.

Def 21.14 A *Formula* is defined the exact same way as for Presburger arithmetic except that the atomic formulas are different.

A *sentence* is a formula where all of the variables are quantified over. We can now define our problem: Given a sentence ϕ in arithmetic is it true?

- This problem is not in AH.
- There are theories even harder that involved quantification over sets.

22 Summary of Relations Between Classes

Known Inclusions

$$\mathrm{REG} \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{R} \subseteq \mathrm{NP}$$

$$\mathrm{NP} = \Sigma_1^{\mathrm{p}} \subseteq \Sigma_2^{\mathrm{p}} \subseteq \Sigma_3^{\mathrm{p}} \subseteq \cdots \subseteq \mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PSPACE}$$

$$(\forall i)[\Sigma_i^{\mathrm{p}} \subseteq \Pi_{i+1}^{\mathrm{p}} \wedge \Pi_i^{\mathrm{p}} \subseteq \Sigma_{i+1}^{\mathrm{p}}]$$

$$\mathrm{BPP} \subseteq \Sigma_2^{\mathrm{p}} \cap \Pi_2^{\mathrm{p}}$$

$$\mathrm{PSPACE} \subseteq \mathrm{DTIME}(\mathrm{TOW}_1(\mathrm{n})) \subseteq \mathrm{DTIME}(\mathrm{TOW}_2(\mathrm{n})) \subseteq \cdots \subseteq \mathrm{EL}$$

$$\mathrm{PSPACE} \subseteq \mathrm{DSPACE}(\mathrm{TOW}_1(\mathrm{n})) \subseteq \mathrm{DSPACE}(\mathrm{TOW}_2(\mathrm{n})) \subseteq \cdots \subseteq \mathrm{EL}$$

$$(\forall i)[\mathrm{DTIME}(TOW_i(n)) \subseteq \mathrm{DSPACE}(TOW_i(n)) \subseteq \mathrm{DTIME}(TOW_{i+1}(n))]$$

$$\mathrm{EL} \subseteq \mathrm{PRIMREC} \subseteq \mathrm{ACK} \subseteq \mathrm{GOOD} \subseteq \mathrm{DEC}$$

$$\mathrm{DEC} \subseteq \Sigma_1 \subseteq \Sigma_2 \subseteq \Sigma_3 \subseteq \cdots \mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{AH}$$

Known Proper Inclusions

$$\mathrm{REG} \subset \mathrm{L} \subset \mathrm{PSPACE} \subset \mathrm{DSPACE}(\mathrm{TOW}_1(\mathrm{n})) \subset \mathrm{DSPACE}(\mathrm{TOW}_2(\mathrm{n})) \subset \cdots \subset \mathrm{EL}$$

$$\mathrm{NPDTIME}(TOW_2(n)) \subset \mathrm{DTIME}(TOW_3(n)) \subset \cdots \subset \mathrm{EL}$$

$$(\forall i)[\mathrm{DTIME}(TOW_i(n)) \subset \mathrm{DSPACE}(TOW_{i+1}(n)) \subset \mathrm{DTIME}(TOW_{i+2}(n))]$$

$$\mathrm{EL} \subset \mathrm{PRIMREC} \subset \mathrm{ACK} \subset \mathrm{GOOD} \subset \mathrm{DEC}$$

$$\mathrm{DEC} \subset \Sigma_1 \subset \Sigma_2 \subset \Sigma_3 \subset \cdots \mathrm{P}^{\#\mathrm{P}} \subset \mathrm{AH}$$

$$(\forall i)[\Sigma_i \neq \Pi_i]$$

What Most Theorists Think

$$\mathrm{L} \subset \mathrm{NL} \subset \mathrm{P} = \mathrm{R} = \mathrm{BPP} \subset \mathrm{NP} \subset \Sigma_2^{\mathrm{p}} \subset \Sigma_3^{\mathrm{p}} \subset \cdots \subset \mathrm{PSPACE}$$

$$\mathrm{NP} \subset \Pi_2^{\mathrm{p}} \subset \Pi_3^{\mathrm{p}} \subset \cdots \subset \mathrm{PSPACE}$$

23 Other Complexity Measures

This chapter has focused on worst case analysis where we are interested in time or space. There are other ways to measure complexity which may be more realistic.

1. *Average case analysis:* There has been some work on formalizing average case analysis. Rather than see how an algorithm works in the worst case, one looks at how it works relative to a distribution. But what distribution is realistic? This is very hard to determine.

2. *Approximation Algorithms:* For many NP-complete problems there are approximation algorithms that are fast and give an answer that is close to the optimal (e.g., within twice). Ther are also lower bounds as well. Some of these algorithms are useable in the real world.

3. *Heuristic algorithms:* There are some rules-of-thumb that seem to work on particular problems. Such approaches tend to work well in the real world but are very hard to analyze.

4. *Fixed Parameter Tractable:* In Section 9 we looked at the Vertex Cover problem. For general k it is NP-complete. For fixed k it is *not* $O(n^k)$ but instead just $O(1.2738^k+kn)$. Many NP-complete problems are *Fixed Parameter Tractable* meaning that if you fix a parameter they can be solved quite fast.

5. *Streaming Algorithms:* The input is a sequence of n numbers where n is quite large. So large that you cannot store n in main memory. We model this by saying we can only pass over the sequence p times and only use $f(n)$ space where $f(n)$ is much less than n. If we want to find the most common element, can we do that with 2 passes and $O(\log n)$ space? Algorithms for these kinds of problems are randomized and approximate. They are called *Streaming Algorithms*

24 Summary

In this chapter we defined many complexity classes. The classes spanned a rather large swath of complexities from $O(1)$ space to various shades of undecidability. For each class we gave examples of natural problems that are in them but likely (or surely) not in lower classes. Hence we have been able to determine how hard many natural problems are.

This classification is a good first cut at getting to the real issue of how hard these problems are. But they are not the entire story since once a problem is discovered to be hard it still needs to be solved. What do you do? W.C. Fields said

If at first you don't succeed, give up. No use making a damn fool of yourself.

We respectfully disagree and counter with what Piet Hein said

Problems worthy of attack prove their worthy by hitting back.

If a problem is hard all that means is that finding an exact solution quickly is hard in the worst case. It could well be that the problem you really want to solve, perhaps a subcase, perhaps an approximation, may still be doable. This is not a pipe dream— many NP-complete problems can be approximated quite well. Section 23 discusses this and other possible ways around hardness results.

We speculate that theory and practice will come closer together as theorists define more realistic classes, and practitioners discover that the size of problems they are working is large enough so that asymptotic results really are useful.

25 What is Natural?

Darling: Bill, since we still don't know that P $\neq$ NP, are there *any* problems that are provably not in P?

Bill: Yes there are such problems! (Thinking of using a diagonalization proof to create one that exists for the sole purpose of not being in P.)

Darling: Great! Unless it's one of those dumb-ass sets that you construct for the sole purpose of not being in P.

Bill: Oh. You nailed it. Okay, so you want a natural problem that's not in P. How about $HALT$.

Darling: Nice try Bill. I want a decidable natural problem that is known to not be in P.

Bill: Do you consider fragments of arithmetic, like Presburger Arithmetic or WS1S, to be natural?

Darling: If it requires a page of definitions then not.

Bill: Oh. OH, I have it! I know a problem that is natural, decidable, easy to describe, and known to not be in P.

Darling: Do tell!

Bill: Add to regular expressions the ability to use exponents like a^{100} instead of writing $a \cdots a$ (100 times). We'll call these t-reg exps. Given two t-reg exp do they generate the same set? This problem is EXPSPACE-complete hence not in PSPACE, hence not in P.

Darling: Why is that problem natural?

Bill: Good question. On the one hand, I didn't construct the problem for the sole purpose of not being in P. So it's not a dumb ass problem. Does it then raise to the level of being natural?

Darling: Perhaps it's intermediary between dumb ass and natural. An intermediary problem. Like graph isomorphism is likely not in P nor NP-complete.

Bill: Okay, I'll take that. Now here is one that might be more natural: Given an $n \times n$ chess board with pieces— (interrupted)

Darling: Unless $n = 8$ this isn't really chess.

Bill: I find both t-reg exps and generalized chess natural because people *could have* worked on those problems. The fact that people didn't is not the point. They both use notions people did study.

Darling: You'll call them natural, I'll call them (0.5)natural, and we can agree to disagree.

Bill: Yeah!

26 Acknowledgement

I thank (1) Raymond Ng for proofreading, (2) Eric Allender for correcting a subtle point about P and NL, and (3) Scott Aaronson, Lance Fortnow, Richard Lipton, and Ken Regan for many interesting discussions on these points over the years.

References

[1] The MONA project, 2002–current. `http://www.brics.dk/mona/index.html`.

[2] S. Aaaronsn, G. Kuperberg, and C. Granade. Complexity zoo. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

[3] S. Aaronson. The scientific case for P$\neq$NP, 2014. `http://www.scottaaronson.com/blog/?p=1720`.

[4] S. Aaronson and A. Wigderson. Algebraization: a new barrier to complexity theory. *ACM Transactions on Computing Theory*, 1, 2009. Prior version in STOC08.

[5] M. Agrawal, N. Kayal, and N. Saxena. PRIMES in p. *Annals of Mathematics*, 160:781–793, 2004.

[6] E. Allender and M. Mahajan. The complexity of planarity testing. In *Seventh International Symposium on Theoretical Aspects of Computer Science: Proceedings of STACS 1990,* Rouen, France, Lecture Notes in Computer Science, New York, Heidelberg, Berlin, 2000. Springer-Verlag. `http://ftp.cs.rutgers.edu/pub/allender/stacs0.pdf`.

[7] D. Apon, W. Gasarch, and K. Lawler. An NP-complete problem in grid coloring, 2012. `http://arxiv.org/abs/1205.3813`.

[8] L. Babai. Graph isomorphism is in quasipolynomial time, 2015. http://people.cs.uchicago.edu/~laci/2015-11-10talk.mp4.

[9] T. Baker, J. Gill, and R. Solovay. Relativizations of the $P =? NP$ question. *SIAM Journal on Computing*, 4:431–442, 1975.

[10] E. Borger, E. Gradel, and Y. Gurevich. *The classical decision problem.* Springer Verlag, New York, Heidelberg, Berlin, 2000.

[11] J. R. Büchi. Weak second order arithmetic and finite automata. *Zeitschrift fuer Mathematik und Physik*, 6:66–92, 1960.

[12] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. of the International Congress on logic, Math, and Philosophy of Science (1960)*, pages 1–12, Stanford, California, 1962. Stanford University Press.

[13] S. Buss and R. Williams. Limits on alternation-trading proofs for time-space lower bounds. In *Twenty-Seventh Conference on Computational Complexity: Proceedings of CCC '12*, pages 181–191. IEEE Computer Society, 2012. http://eccc.hpi-web.de/report/2011/031/.

[14] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47:1028–1047, 2000. Prior version in FOCS 1997.

[15] J. Chen, L. Kanj, and G. Xia. Improved upper bounds for vertex cover. *TCS*, 411:3736–3756, 2010.

[16] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Opeartions Research*, 4:233–235, 1979.

[17] S. Datta, N. Limaye, P. Nimbhorkar, T. Thieraf, and F. Wagner. A log-space algorithm for canonization of planer graphs. In *Twenty-Fourth Conference on Computational Complexity: Proceedings of CCC '09*, pages 162–167. IEEE Computer Society Press, 2009. http://arxiv.org/abs/0809.2319.

[18] M. Davis, H. Putnam, and J. Robinson. The decision problem for exponential diophantine equations. *Annals of Mathematics*, 74:425–436, 1961.

[19] D.Buchfuhrer and C. Umans. The complexity of boolean formula minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011. on Umans Homepage, also in ICALP 2008.

[20] E. Dijkstra. A note in two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[21] S. Fenner, W. Gasarch, C. Glover, and S. Purewal. Rectangle free colorings of grids, 2012. http://arxiv.org/abs/1005.3750.

[22] J. Ferrante and C. Rackoff. *The computational complexity of logical theories.* Springer, New York, Heidelberg, Berlin, 1979.

[23] M. Fischer and M. Rabin. String matching and other products. *Complexity of Computation, Richard Karp (editor), SIAM-AMS Proc.*, 7:27–41, 1974.

[24] R. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962. doi:10.1145/367766.368168.

[25] W. Gasarch. Computational complexity column 36: The P=NP poll. *SIGACT News*, 33(2):34–47, 2002.

[26] W. Gasarch. Computational complexity column 74: The P=NP poll. *SIGACT News*, 43, 2012.

[27] W. Gasarch. Why do we think P$\neq$NP, 2014. `blog.computationalcomplexity.org/2014/03/why-do-we-think-p-ne-np-inspired-by.html`.

[28] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proceedings of the Fourteenth Annual ACM Symposium on the Theory of Computing,* San Francisco CA, pages 60–65, 1982.

[29] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Math Society*, 117:285–306, 1965.

[30] J. Håstad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing,* Berkeley CA, pages 6–20, 1986.

[31] J. Håstad. Almost optimal lower bounds for small depth circuits. In S. Micali, editor, *Randomness and Computation*, pages 143–170, Greenwich, CT, 1989. JAI Press.

[32] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[33] L. Khachiyan. A polynomial algorithm for linear programming. *Doklady Academy Nauk, SSSR*, 244:1093–1096, 1979. Translation in Soviet Math Doklady, Vol 20, 1979.

[34] L. Kirby and J. Paris. Accessible independent results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:285–293, 1982. `http://blms.oxfordjournals.org/content/by/year`.

[35] J. Kruskal. On the shortest spanning subtree of a graph and the travellings salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956. doi:10.1090/S0002-9939-1956-0078686-7.

[36] T. Kuhn. *The structure of scienfic revolutions.* University of Chicago Press, Chicago, 1962.

[37] R. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, Jan. 1975.

[38] Lautemann. BPP and the polynomial hierarchy. *Information Processing Letters*, 17(4), 1983.

[39] S. Lindell. A log-space algorithm for canonization of planer graphs. In *STOC92*, pages 400–404, New York, 1992. ACM.

[40] R. Lipton. Advances on group isomorphism, 2013. `http://rjlipton.wordpress.com/2013/05/11/advances-on-group-isomorphism`.

[41] R. Lipton. Could we have felt evidence for SDP≠p?, 2014. `http://rjlipton.wordpress.com/2014/03/15/could-we-have-felt-evidence-for-sdp-p/`.

[42] R. Lipton and K. Regan. *People, problems, and proof: essays from Godel's last letter: 2010.* Springer, New York, Heidelberg, Berlin, 2013.

[43] E. Luks. Isomorphism of bounded valence graphs can be tested in polynomial time. *Journal of Computer and System Sciences*, pages 42–65, 1982.

[44] S. Mahaney. Sparse complete sets for NP: Solution to a conjecture of Berman and Hartmanis. *Journal of Computer and System Sciences*, 25:130–143, 1982.

[45] Y. Matijasevic. Enumerable sets are diophantine (Russian). *Doklady Academy Nauk, SSSR*, 191:279–282, 1970. Translation in Soviet Math Doklady, Vol 11, 1970.

[46] A. Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic Colloquium; 1975*, 1978.

[47] A. Meyer and L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proc. of the 13th Annual IEEE Sym. on Switching and Automata Theory*, pages 125–129, 1972.

[48] D. Moshkovitz. The projection games conjecture and the NP-hardness of $\ln n$-apprxoimating set-cover, 2013.

[49] K. Mulmuley. On P vs. NP, and geometric complexity complexity theory. *Journal of the ACM*, 58(2):5.1–5.25, 2011. Earlier version from 2009 at `http://arxiv.org/abs/0903.0544` this version at `http://doi.acm.org/10.1145/1667053.1667060`.

[50] N. Nisan and A. Wigderson. Hardness vs randomness. *Journal of Computer and System Sciences*, 49:149–167, 1994. Prior version in FOCS88. Full Version at `http://www.math.ias.edu/~avi/PUBLICATIONS/`.

[51] C. Pomerance. A tale of two sieves. *Notices of the American Mathematical Society*, 43:1473–1485, 1996.

[52] K. Popper. *The logic of scientific discovery.* Routledge, 1959. The original in German was published in 1934. The English version is online for free.

[53] R. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, 36:1389–1401, 1957.

[54] G. Pullum. Sccoping the loop snooper, 2000. `http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html`.

[55] M. Rabin. Decidabilty of second-order theories of automta on infinite trees. *Transactions of the American Math Society*, 141:1–35, 1969. `www.jstor.org/stable/1995086`.

[56] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Res. and Dev.*, 3:114–125, 1959.

[57] M. O. Rabin. A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[58] A. A. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, Aug. 1997. Prior version in *A*CM Sym on Theory of Computing, 1994 (STOC).

[59] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17.1–17.26, 2008. doi: 10.1145/1391289.1391291.

[60] D. Rosenbaum. Bidirectional collision detection and faster determinsitic isomorphism testing, 2013. `http://arxiv.org/abs/1304.3935`.

[61] B. Roy. Transitivite et connexite. *C.R. Acad. Sci Paris*, 249:216–218, 1959.

[62] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[63] P. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science,* Santa Fe NM, pages 121–134, 1994.

[64] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing,* Boston MA, pages 330–335, 1983.

[65] M. Sipser. Expanders, randomness, or time versus space. *JCSS*, 36:379–383, 1988. Earlier version in CCC 1986, then called Structures.

[66] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on the Theory of Computing,* New York, pages 77–82, 1987.

[67] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[68] P. M. Stephen A. Cook. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385–394, 1987.

[69] R. Tarjan. Worst-caes analysis of set union algorihtms. *Journal of the ACM*, 31(2):245–281, 1984.

[70] A. Tarski. A decision method for elemenary algebra and geometry, 1948. `http://www.rand.org/content/dam/rand/pubs/reports/2008/R109.pdf`.

[71] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20:865–877, 1991. Prior version in *I*EEE Sym on Found. of Comp. Sci., 1989 (FOCS).

[72] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[73] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[74] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962. doi:10.1145/321105.321107.

[75] V. Weispfenning. Tarski's decision procedure FO theory of reals, 2012. `http://www.cfdvs.iitb.ac.in/meetings/files/tarski.pdf`.

[76] T. Who. Don't get fooled again. `https://www.youtube.com/watch?v=zYMD_W_r3Fg`.

[77] R. Williams. Time space tradeoffs for counting *np* solutions modulo integers. In *Computational Complexity*, pages 179–219, New York, 2008. IEEE. `http://www.stanford.edu/~rrwill/projects.html`.

[78] R. Williams. Alternation-tradings, linear programming, and lower bounds. In *Twenty seventh International Symposium on Theoretical Aspects of Computer Science: Proceedings of STACS 2010,* Nancy, France, 2010.

[79] R. Williams. Non-uniform ACC lower bounds. *Journal of the ACM*, 61:2.1–2.31, 2014. Prior version in CCC2011.

[80] B. Yandell. *The honor class: Hilbet's problems and their solvers.* A.K. Peters, 2002.

[81] A. C. Yao. Separating the polynomial-time hierarchy by oracles. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science,* Portland OR, pages 1–10, 1985.