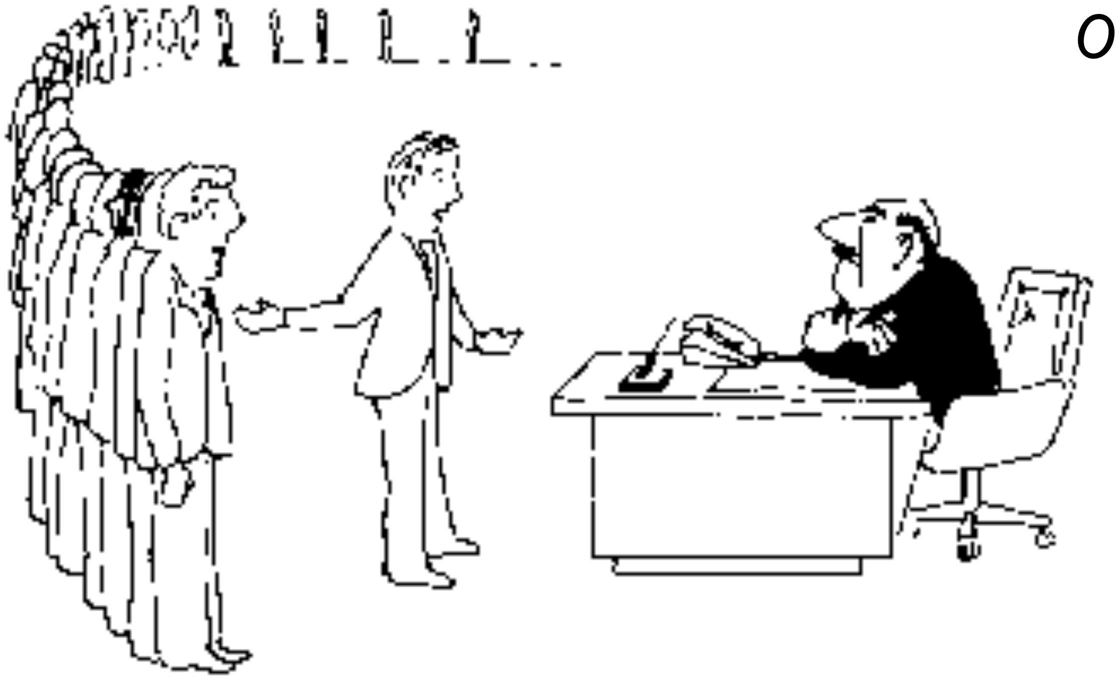
15-251
Great Theoretical Ideas in Computer Science

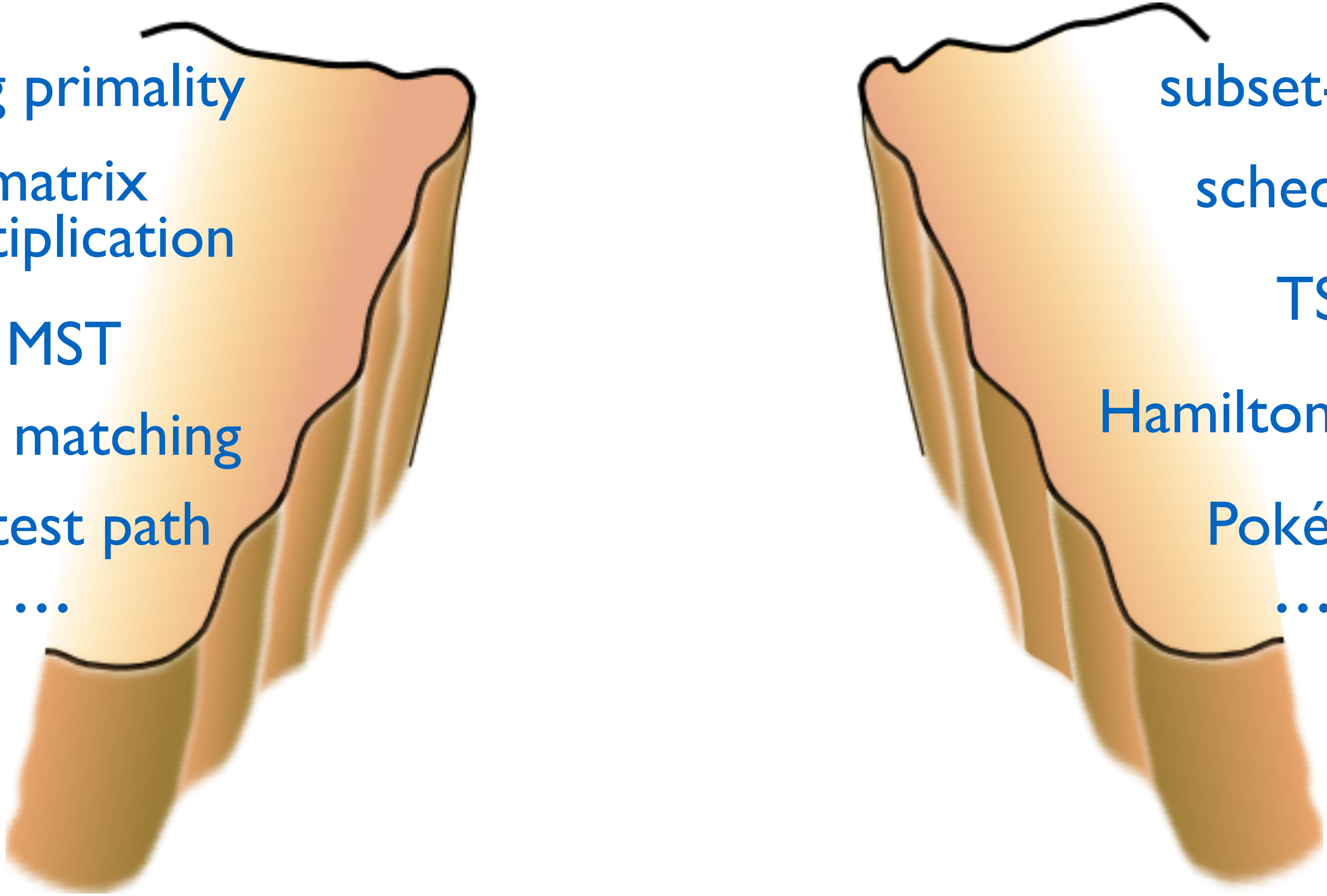
Lecture 13:
NP and NP-completeness 1

October 11th, 2016



I can't find an efficient algorithm, but neither can all these famous people.

There is a big chasm between poly-time and exp-time.



testing primality

matrix multiplication

MST

max matching

shortest path

…

poly-time solvable

subset-sum

scheduling

TSP

Hamiltonian cycle

Pokémon

…

best we can say:
exp-time solvable

Exponential running time examples

Subset Sum Problem

Given a list of integers, determine if there is a subset of the integers that sum to 0.

4	-3	-2	7	99	5	1

Exponential running time examples

Subset Sum Problem

Given a list of integers, determine if there is a non-empty subset of the integers that sum to 0.

4	-3	-2	7	99	5	1

Exhaustive Search (Brute Force Search):

> Try every possible subset and see if it sums to 0.

subsets is 2^n $\implies$ running time at least 2^n

Note: checking if a given subset sums to 0 is **easy**.

Exponential running time examples

Theorem Proving Problem
(informal description)

Given a mathematical proposition P and an integer k, determine if P has a proof of length at most k.

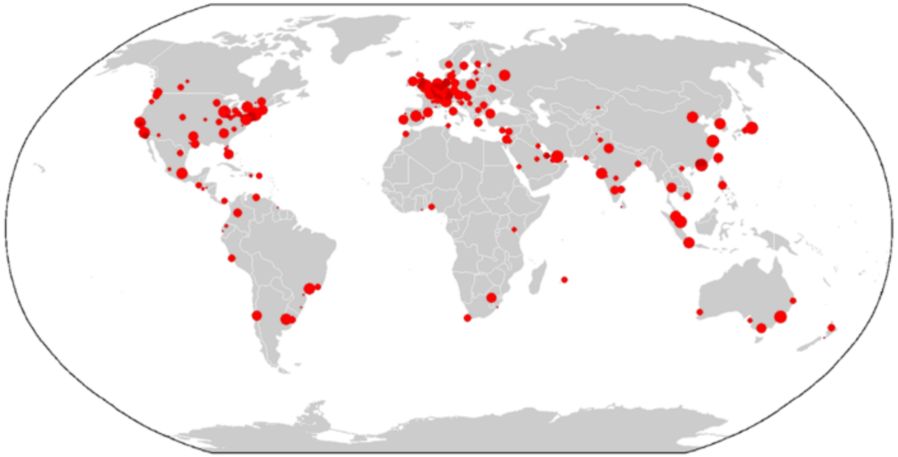
Exhaustive Search (Brute Force Search):

> Try every possible "proof" of length at most k, and check if it corresponds to a valid proof.

Note: checking if a given proof is correct is **easy**.

Exponential running time examples

Traveling Salesperson Problem (TSP)



Is there an order in which you can visit the cities so that ticket cost is < $50000?

Exhaustive Search (Brute Force Search):

> Try every possible order and compute the cost.

Note: checking if a given solution has the desired cost is **easy**.

Exponential running time examples

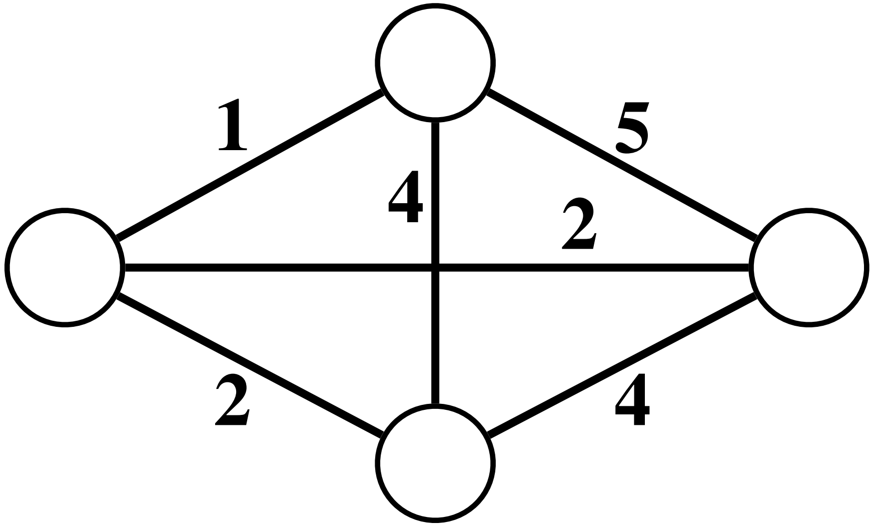
Traveling Salesperson Problem (TSP)

Input:

A graph $G = (V, E)$, edge weights w_e (non-negative, integral) and target t.

Output:

Yes, iff there is a cycle of cost at most t that visits every vertex exactly once.



Exponential running time examples

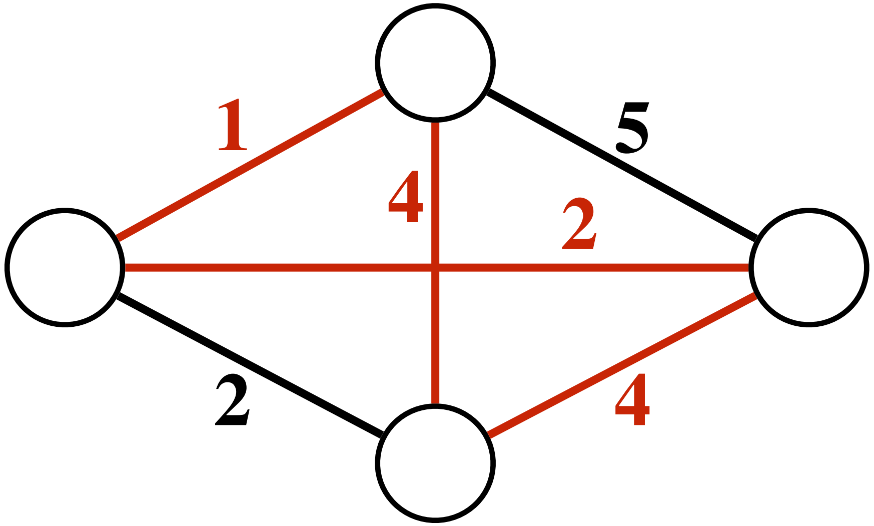
Traveling Salesperson Problem (TSP)

Input:

A graph $G = (V, E)$, edge weights w_e (non-negative, integral) and target t.

Output:

Yes, iff there is a cycle of cost at most t that visits every vertex exactly once.



Exponential running time examples

Satisfiability Problem (SAT)

Input: A Boolean propositional formula.

e.g. $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3 \wedge x_4) \vee x_3$

Output: Yes iff there is an assignment to the variables that makes the formula True.

Exhaustive Search (Brute Force Search):

> Try every possible truth assignment to the input variables. Evaluate the formula to see the output.

Note: checking if a given truth assignment makes the formula True is **easy**.

Exponential running time examples

Circuit Satisfiability Problem (Circuit-SAT)

Input: A Boolean circuit.

Output: Yes iff there is an assignment to the input gates that makes the circuit output 1.

Exhaustive Search (Brute Force Search):

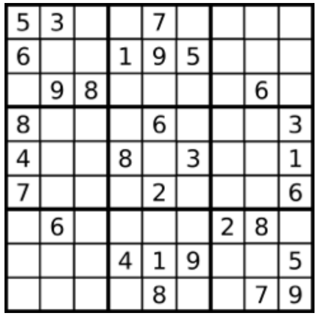
> Try every possible truth assignment to the input gates. Evaluate the circuit to see the output.

Note: checking if a given assignment makes the circuit output 1 is **easy**.

Exponential running time examples

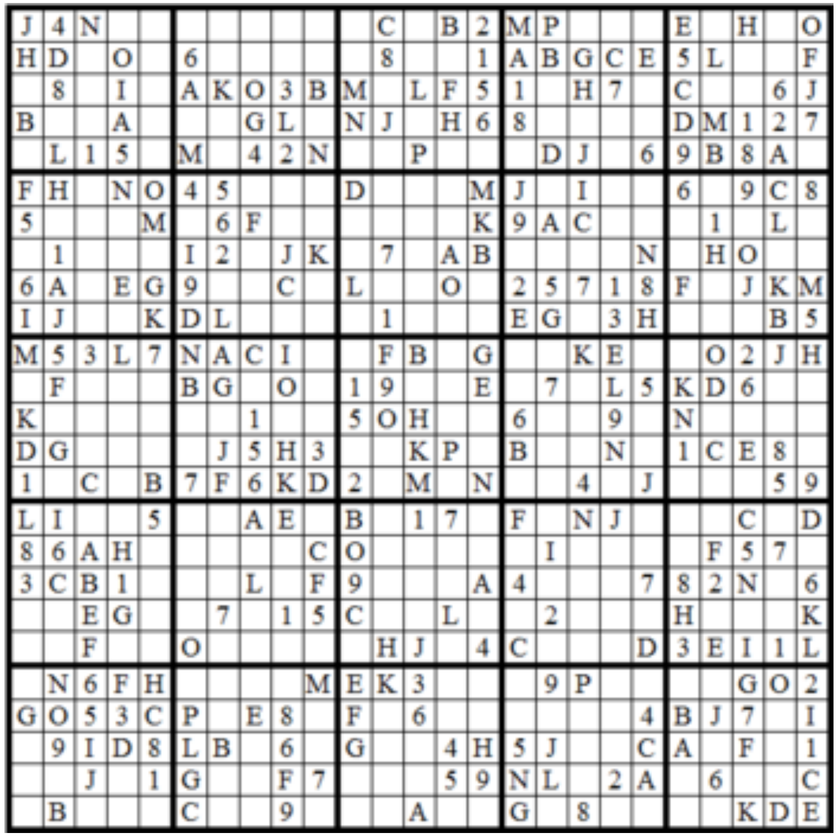
Sudoku Problem

Given a partially filled *n* by *n* sudoku board, determine if there is a solution.



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

	A	8		4							6		E	7	
2				E	A					C	F				3
D	C	4	7									A	6	9	G
		F		5	G					A	D		B		
	G		6		C	A			7	8		4		B	
		9			2	G			A	B			C		
				1		6	4	F	G		3				
			2									3			
			5									B			
				3		F	D	8	4		5				
		C			B	2			3	G			9		
	D		E		6	7			B	1		2		4	
		3		7	1					5	4		G		
G	F	2	A									C	7	5	4
6				D	9					F	C				1
	5	1		8							G		3	E	



J	4	N									C		B	2	M	P				E		H		O
H	D		O		6						8			1	A	B	G	C	E	5	L			F
	8		I		A	K	O	3	B	M		L	F	5	1		H	7		C			6	J
B			A				G	L		N	J		H	6	8					D	M	1	2	7
	L	1	5		M		4	2	N			P				D	J		6	9	B	8	A	
F	H		N	O	4	5				D				M	J		I			6		9	C	8
5				M		6	F							K	9	A	C				1		L	
	1				I	2		J	K		7		A	B					N		H	O		
6	A		E	G	9			C		L			O		2	5	7	1	8	F		J	K	M
I	J			K	D	L					1				E	G		3	H				B	5
M	5	3	L	7	N	A	C	I			F	B		G			K	E			O	2	J	H
	F				B	G		O		1	9			E		7		L	5	K	D	6		
K							1			5	O	H			6			9		N				
D	G					J	5	H	3			K	P		B			N		1	C	E	8	
1		C		B	7	F	6	K	D	2		M		N			4		J				5	9
L	I			5			A	E		B		1	7		F		N	J				C		D
8	6	A	H						C	O						I					F	5	7	
3	C	B	1				L		F	9				A	4				7	8	2	N		6
		E	G			7		1	5	C			L			2				H				K
		F			O						H	J		4	C				D	3	E	I	1	L
	N	6	F	H					M	E	K	3				9	P					G	O	2
G	O	5	3	C	P		E	8		F		6							4	B	J	7		I
	9	I	D	8	L	B		6		G			4	H	5	J			C	A		F		1
		J		1	G			F	7				5	9	N	L		2	A		6			C
	B				C			9				A			G		8					K	D	E

Exponential running time examples

Sudoku Problem

Given a partially filled *n* by *n* sudoku board, determine if there is a solution.

Exhaustive Search (Brute Force Search):

> Try every possible way of filling the empty cells and check if it is valid.

Note: checking if a given solution is correct is **easy**.

In our quest to understand efficient computation,
(and life, the universe, and everything)
we come across:

P vs NP problem

"Can creativity be automated?"

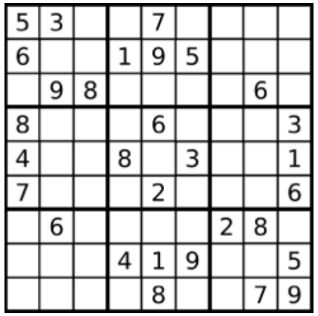
Biggest open problem in all of Computer Science.

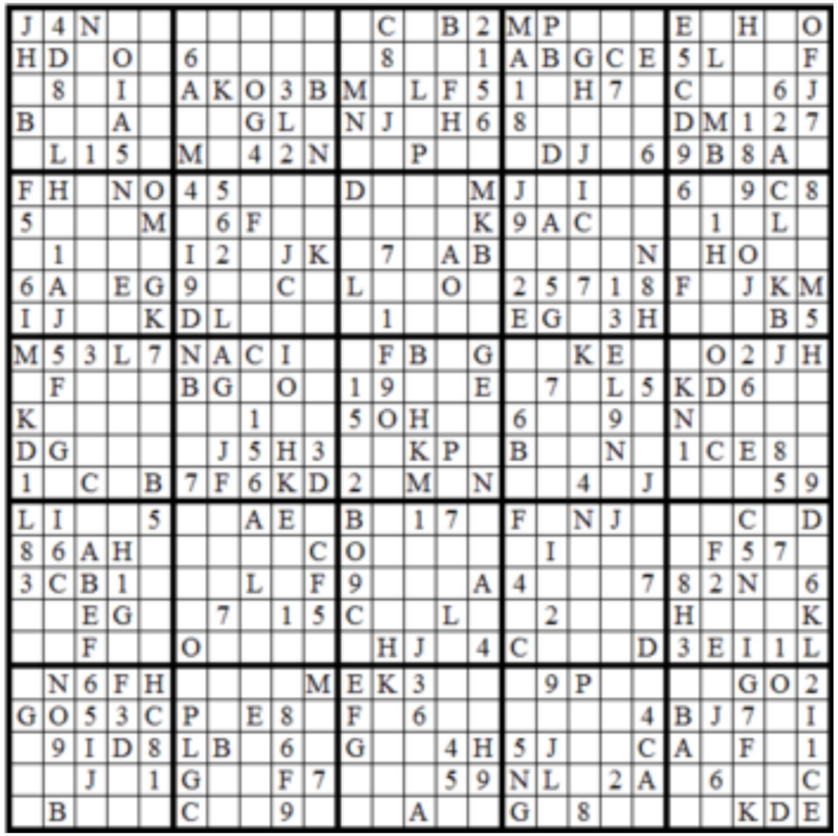
One of the biggest open problems in all of Mathematics.

So what is the P vs NP question?

The P vs NP question is the following:

Can the Sudoku problem be solved in polynomial time?



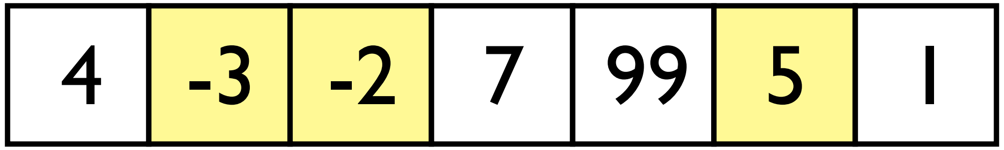


WTF?!

So what is the P vs NP question?

The P vs NP question is the following:

Can the Subset Sum problem be solved in poly-time?

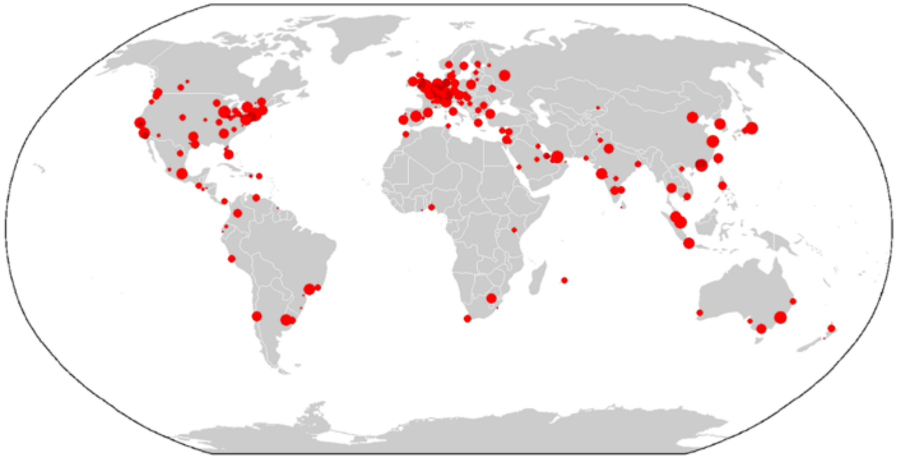


4	-3	-2	7	99	5	1

So what is the P vs NP question?

The P vs NP question is the following:

Can the Traveling Salesperson (TSP) problem be solved in poly-time?



So what is the P vs NP question?

The P vs NP question is the following:

Can the Theorem Proving problem be solved in poly-time?

What the &$%# is going on?!?

Let’s explain from the beginning.

Toolbox of a computer scientist

1. Basic algorithmic techniques

   e.g. greedy algorithms, divide and conquer, dynamic programming, linear programming, semi-definite programming, etc…

2. Basic data structures

   e.g. queues, stacks, hash tables, binary search trees, etc…

3. Identifying and dealing with intractable problems

Toolbox of a computer scientist

3. Identifying and dealing with intractable problems

After decades of research and billions of dollars of funding, no one was able to come up with poly-time algs for:

Theorem Proving, TSP, Subset Sum, Sudoku, Tetris, …

It would be fantastic if we could prove that these **<u>cannot</u>** be solved in poly-time. But…

P

Toolbox of a computer scientist

3. Identifying and dealing with intractable problems

But we are far from accomplishing this.

(maybe these problems are in **P**???)

So what can we do???

Maybe we can try to gather *evidence* that these problems are hard.

Goal:

Find evidence these problems are computationally hard.

Revisiting reductions

A central concept used to compare the "difficulty" of problems.

↓

will differ based on context

Now we are interested in poly-time decidability vs

not poly-time decidability

Want to define: $A \leq B$ (B is at least as hard as A w.r.t. poly-time decidability.)

B poly-time decidable $\implies$ A poly-time decidable

$$B \in \mathsf{P} \implies A \in \mathsf{P}$$

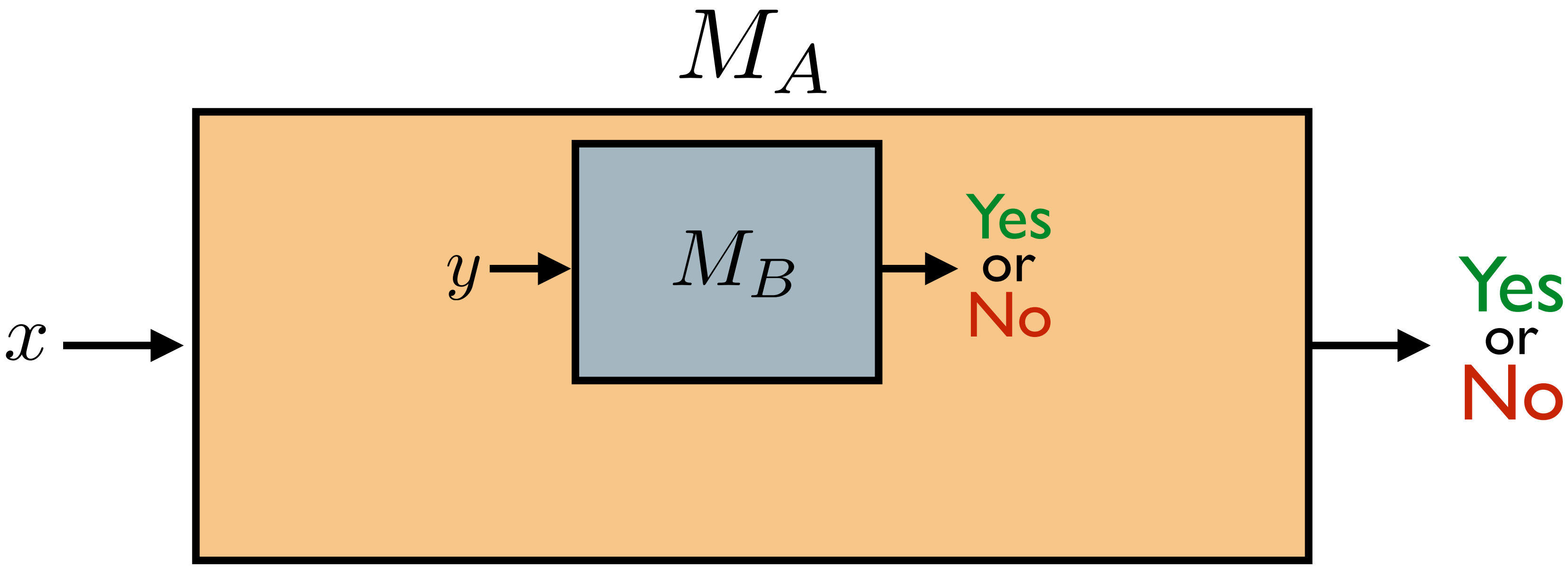
A **not** poly-time decidable $\implies$ B **not** poly-time decidable

$$A \notin \mathsf{P} \implies B \notin \mathsf{P}$$

Revisiting reductions

Notation: $A \leq_T^P B$ (A poly-time reduces to B)

if there is a poly-time machine M_A that decides A using an oracle M_B for B as a black-box subroutine.



B in **P** $\implies$ A in **P**

A not in **P** $\implies$ B not in **P**

Revisiting reductions

```
def $M_B$(…):
    # some code that solves problem B

def $M_A$(…):
    # some code that solves problem A
    # that makes calls to function $M_B$ when needed
```

If M_B poly-time $\Longrightarrow$ M_A poly-time

then we would write $A \leq_T^P B$.

When you want to show $A \leq_T^P B$,
you need to come up with a poly-time M_A.

Revisiting reductions

Example

A:

Given a graph and an integer k, does there exist at least k pairs of vertices connected to each other?

B:

Given a graph and a pair of vertices (s,t),
is s and t connected?

A poly-time reduces to **B**

Revisiting reductions

Example

A:

Given a sequence of integers, and a number k,
is there an increasing subsequence of length at least k?

3, 1, 5, 2, 3, 6, 4, 8

B:

Given two sequences of integers, and a number k,
is there a common subsequence of length at least k?

3, 1, 5, 2, 3, 6, 4, 8

1, 5, 7, 9, 2, 4, 1, 0, 2, 0, 3, 0, 4, 0, 8

A poly-time reduces to **B**

The two sides of reductions

1. Expand the landscape of tractable problems.

If $A \leq_T^P B$ and B is tractable, then A is tractable.

$$B \in \mathbf{P} \implies A \in \mathbf{P}$$

Whenever you are given a new problem to solve:

- check if it is already a problem you know how to solve in disguise.
- check if it can be reduced to a problem you know how to solve.

The two sides of reductions

2. Expand the landscape of intractable problems.

If $A \le_T^P B$ and A is intractable, then B is intractable.

$$A \notin \mathbf{P} \implies B \notin \mathbf{P}$$

But we are pretty lousy at showing a problem is intractable.

Maybe we can still make good use of this…

Gathering evidence for intractability

Suppose we want to gather evidence that $A \notin$ **P**.

If we can show $L \leq_T^P A$ for **<u>many</u>** L

(including some L that we really think should not be in **P**)

then that would be good evidence that $A \notin$ **P**.

Definitions of C-hard and C-complete

Definition: Let **C** be a set of languages containing **P**.

We say that language A is **C**-hard if

for all $L \in$ **C**, $L \leq_T^P A$.

A is at least as hard as every language in **C**.

Definition: Let **C** be a set of languages containing **P**.

We say that language A is **C**-complete if

- A is **C**-hard
- $A \in$ **C**

A is a representative for hardest languages in **C**.

Definitions of C-hard and C-complete

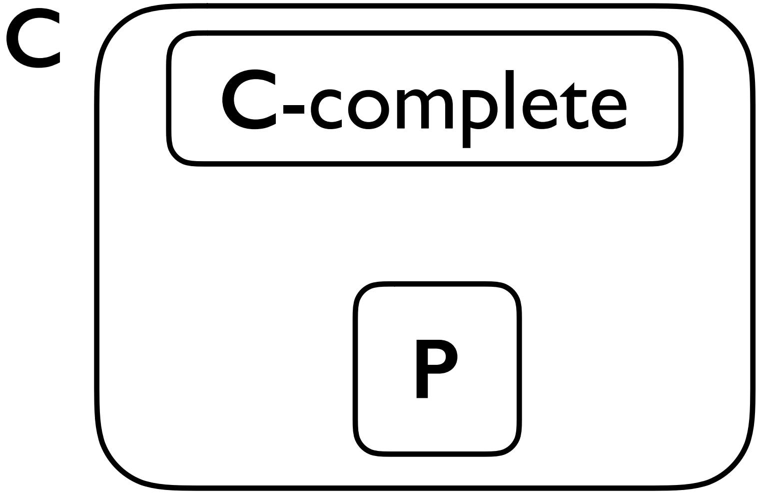
Observation:

Suppose A is **C**-complete.

- If **C** = **P**, then $A \in$ **P**.
- If $A \in$ **P**, then **C** = **P**.

$$\boxed{\mathbf{C} = \mathbf{P} \iff A \in \mathbf{P}}$$

If we believe **C** $\neq$ **P**, then we must believe $A \notin$ **P**.



Recall the goal

Good evidence that A is intractable:

- A is **C**-hard for a really rich set **C**

  (a set **C** such that we believe **C** $\neq$ **P**)

So what is a good choice for **C** ?

(if we want to show TSP, Subset-Sum, Sudoku, etc… are **C**-hard?)

Recall the goal

What if we let **C** be the set of all languages?

Can it be true that TSP is **C**-hard?

What if we let **C** be the set of all languages decidable using Brute Force Search (BFS)?

Can it be true that TSP is **C**-hard?

A complexity class for BFS?

How can we identify the problems solvable using BFS?

What would be a reasonable definition?

What is common about
TSP, Subset-Sum, Theorem Proving Problem, etc…?

Seems hard to find a correct solution
(solution space is too big!)

BUT, easy to verify a given solution.

The complexity class **NP**

Informally:

A language is in **NP** if:

whenever we have a Yes instance,
there is a “*simple*” proof (solution) for this fact.

1. The length of the proof is polynomial in the input size.
2. The proof can be verified/checked in polynomial time.

The complexity class **NP**

Formally:

Definition:

A language A is in **NP** if

- there is a polynomial-time TM V
- a polynomial p

such that for all $x \in \Sigma^*$:

$$x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x,u) = 1$$

If $x \in A$, there is some proof that leads V to accept.

If $x \notin A$, every "proof" leads V to reject.

The complexity class **NP**

Formally:

Definition:

A language A is in **NP** if

- there is a polynomial-time TM V
- a polynomial p

such that for all $x \in \Sigma^*$:

$$x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x,u) = 1$$

The following are synonyms in this context:

proof = solution = certificate

NP: A game between a Prover and a Verifier

Verifier

poly-time
skeptical

Prover

omniscient
untrustworthy

Given some input x (known both to **Verifier** and **Prover**)

Prover wants to convince **Verifier** that $x \in A$.

Prover cooks up a "proof" u and sends it to **Verifier**.

Verifier (in poly-time), should be able to tell
if the proof is legit.

NP: A game between a Prover and a Verifier

Verifier

poly-time
skeptical

Prover

omniscient
untrustworthy

“Completeness”

If $x \in A$, there must be some proof u that convinces the **Verifier**.

“Soundness”

If $x \notin A$, no matter what “proof” **Prover** gives, **Verifier** should detect the lie.

NP: A game between a Prover and a Verifier

Verifier

poly-time
skeptical

Prover

omniscient
untrustworthy

If we have completeness and soundness, then

$$A \in \textbf{NP}.$$

Examples of languages in **NP**

CLIQUE

Input: $\langle G, k \rangle$ where G is a graph and k is a positive int.

Output: Yes iff G contains a clique of size k.

Fact: CLIQUE is in **NP**.

Examples of languages in NP

Proof: We need to show a verifier TM V exists as specified in the definition of **NP**.

```
def $V(x, u)$ :
    - if $x$ is not an encoding $\langle G = (V, E), k \rangle$ of a valid graph G and a positive integer k, REJECT.
    - if $u$ is not an encoding of a set $S \subseteq V$ of size k, REJECT.
    - for each pair of vertices in $S$:
        - if the vertices are not neighbors, REJECT.
    - ACCEPT
```

Examples of languages in **NP**

Proof (continued):

Need to show:

1. if $x \in$ CLIQUE, there is some proof u (of poly-length) that makes V ACCEPT.
2. if $x \notin$ CLIQUE, no matter what u is, V REJECTS.
3. V is polynomial-time.

   (we leave 3 as an exercise)

Examples of languages in **NP**

Proof (continued):

Need to show:

1. if $x \in$ CLIQUE, there is some proof u (of poly-length) that makes V ACCEPT.

if $x \in$ CLIQUE, then $x = \langle G, k \rangle$ is a valid encoding, and G contains a clique of size k.

Then when u is a valid encoding of this clique,
the verifier will accept.

Examples of languages in **NP**

Proof (continued):

Need to show:

2. if $x \notin$ CLIQUE, no matter what u is, V REJECTS.

if $x \notin$ CLIQUE, then there are 2 options:

- x is not a valid encoding $\langle G, k \rangle$.
- x is a valid encoding, but G does not contain a clique of size k.

In either case, V rejects for any u.
(add a couple of lines of justification)

□

This would be the proper way of showing that a language is in **NP**.

However, we usually don't write it this way.

- We assume implicitly that inputs are automatically checked to be of the correct type.
- Instead of starting with the description of V, we start with the description of the expected proof.
- We describe things at a very high level and skip many details.

Examples of languages in **NP**

3COL

Input: $\langle G \rangle$ where G is a graph.

Output: Yes iff G is 3-colorable.

Fact: 3COL is in **NP**.

Examples of languages in **NP**

Proof (sort of):

The proof string is a valid coloring of the vertices with 3 colors.

The verifier goes through each edge one by one and checks that the endpoints are different colors.

If the input graph is 3-colorable, this check will succeed for a valid 3-coloring of the vertices.

If the input graph is not 3-colorable, then no matter what 3-coloring is given, the verifier will be able to find an edge whose endpoints are colored the same.

The verifier is poly-time since going through each edge and checking their colors takes poly-time.

Examples of languages in **NP**

CIRCUIT-SAT

Input: $\langle C \rangle$ where C is a Boolean circuit.

Output: Yes iff C is satisfiable.

Fact: CIRCUIT-SAT is in **NP**.

Exercise

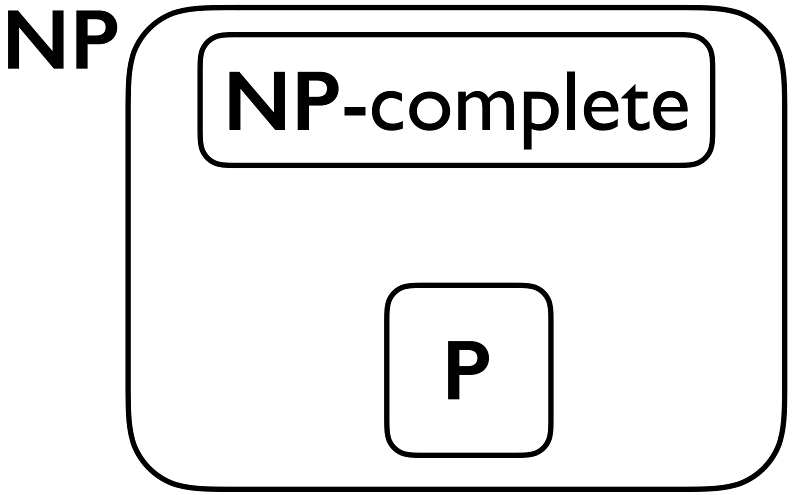
The complexity class **NP**

2 Observations:

1. Every decision problem in **NP** can be solved using BFS.
   - Go through all possible proofs u, and run $V(x, u)$

2. This is a pretty BIG class!

   Contains everything in **P**. (recitation)



People expect **NP** contains much more than **P**.

Coming back to our goal

We wanted to find evidence that
TSP, Subset-Sum, Theorem Proving problem, etc.
are not in **P**.

Could it be true that one of them is **NP**-complete?

Is there **any** language that is **NP**-complete?

Is **NP**-completeness a useful definition?

The Cook-Levin Theorem

Theorem (Cook 1971 - Levin 1973):

SAT is **NP**-complete.

So SAT is in **NP**. (easy)

And for every L in **NP**, L $\leq_T^P$ SAT .

Karp's 21 **NP**-complete problems

1972: "Reducibility Among Combinatorial Problems"

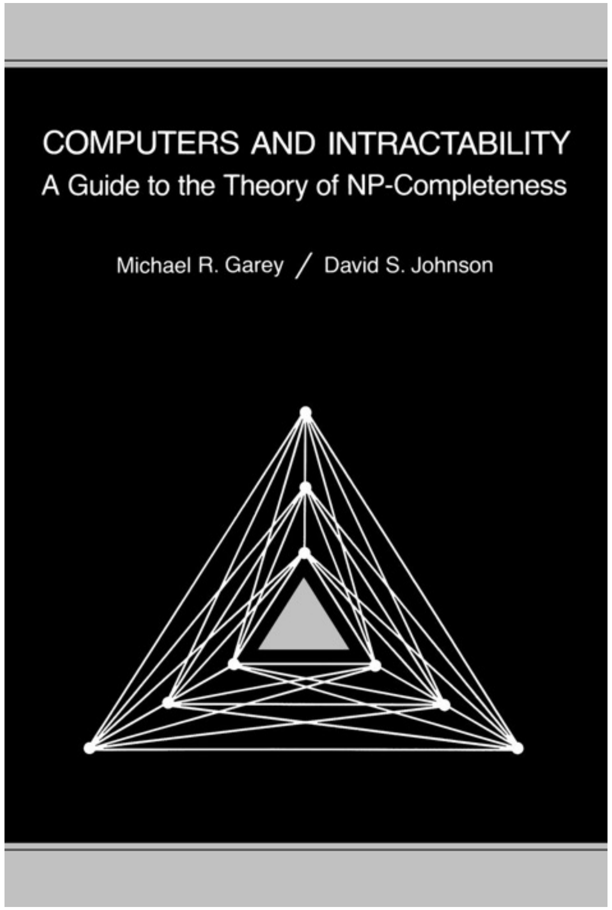
0-1 Integer Programming
Clique
Set Packing
Vertex Cover
Set Covering
Feedback Node Set
Feedback Arc Set
Directed Hamiltonian Cycle
Undirected Hamiltonian Cycle
3SAT

Partition
Clique Cover
Exact Cover
Hitting Set
Knapsack
Steiner Tree
3-Dimensional Matching
Job Sequencing
Max Cut
Chromatic Number

Today

Thousands of problems are known to be **NP**-complete.

(including the languages mentioned in this lecture)



COMPUTERS AND INTRACTABILITY

A Guide to the Theory of NP-Completeness

Michael R. Garey / David S. Johnson

1979

Some other "interesting" examples

Super Mario Bros

Given a Super Mario Bros level, is it completable?

Classic Nintendo Games are (NP-)Hard

Greg Aloupis[*] Erik D. Demaine[†] Alan Guo[†‡]

March 9, 2012

Abstract

We prove NP-hardness results for five of Nintendo's largest video game franchises: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. Our results apply to Super Mario Bros. 1, 3, Lost Levels, and Super Mario World; Donkey Kong Country 1–3; all Legend of Zelda games except Zelda II: The Adventure of Link; all Metroid games; and all Pokémon role-playing games. For Mario and Donkey Kong, we show NP-completeness. In addition, we observe that several games in the Zelda series are PSPACE-complete.

1 Introduction

A series of recent papers have analyzed the computational complexity of playing many different video games [1, 4, 5], yet the most well-known Nintendo games of our youth have yet to be included among these results. In this paper, we consider some of the best-known Nintendo games of all time—Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon—and prove that it is NP-hard to play generalized versions of many games in these series. In particular, our results for Mario apply to the NES games Super Mario Bros., Super Mario Bros.: The Lost Levels, Super Mario Bros. 3, and Super Mario World (developed by Nintendo); our results for Donkey Kong apply to the SNES games Donkey Kong Country 1–3 (developed by Rare Ltd.); our results for Legend of Zelda apply to all Legend of Zelda games (developed by Nintendo) except the side-scrolling Zelda II: The Adventure of Link; our results for Metroid apply to all Metroid games (developed by Nintendo); and our results for Pokémon apply to all Pokémon role-playing games (developed by Game Freak and Creatures Inc.).[1]

Our results are motivated in particular by tool-assisted speed runs for video games. A *speed run* of a game is a play through that aims to achieve a fast completion time, usually performed by a human. In a *tool-assisted* speed run, the player uses special tools, such as emulators, to allow them to slow the game down to a frame-by-frame rate in order to achieve superhuman reflexes and timing. In some sense, tool assistance is not cheating because the rules of the game are still obeyed. The resulting speed runs are impressive to watch, as the paths taken by a tool-assisted player are

---

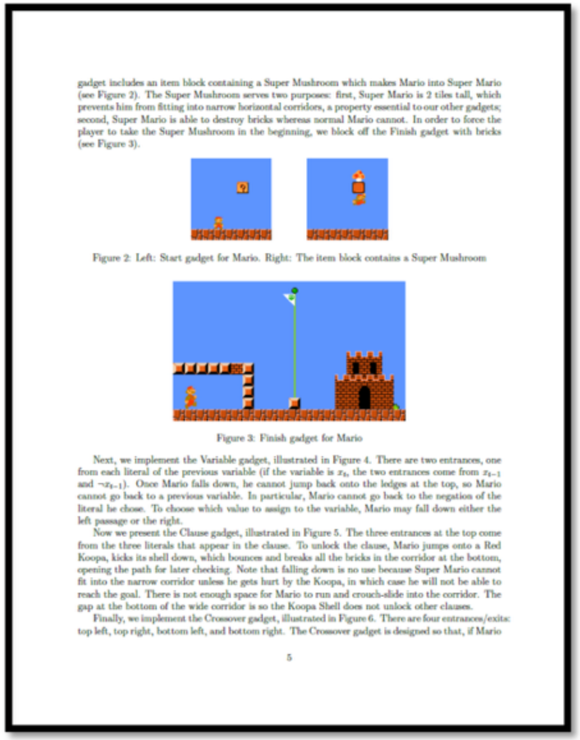
[*] Chargé de Recherches du FNRS, Département d'Informatique, Université Libre de Bruxelles, `aloupis.greg@gmail.com`. Work initiated while at Institute of Information Science, Academia Sinica.

[†] MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, `{edemaine,aguo}@mit.edu`

[‡] Partially supported by NSF grants CCF-0829672, CCF-1065125, and CCF-6922462.

[1] All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Sprites are used here under Fair Use for the educational purpose of illustrating mathematical theorems.

1



gadget includes an item block containing a Super Mushroom which makes Mario into Super Mario (see Figure 2). The Super Mushroom serves two purposes: first, Super Mario is 2 tiles tall, which prevents him from fitting into narrow horizontal corridors, a property essential to our other gadgets; second, Super Mario is able to destroy bricks whereas normal Mario cannot. In order to force the player to take the Super Mushroom in the beginning, we block off the Finish gadget with bricks (see Figure 3).

Figure 2: Left: Start gadget for Mario. Right: The item block contains a Super Mushroom

Figure 3: Finish gadget for Mario

Next, we implement the Variable gadget, illustrated in Figure 4. There are two entrances, one from each literal of the previous variable (if the variable is x_i, the two entrances come from x_{i-1} and $\neg x_{i-1}$). Once Mario falls down, he cannot jump back onto the ledges at the top, so Mario cannot go back to a previous variable. In particular, Mario cannot go back to the negation of the literal he chose. To choose which value to assign to the variable, Mario may fall down either the left passage or the right.

Now we present the Clause gadget, illustrated in Figure 5. The three entrances at the top come from the three literals that appear in the clause. To unlock the clause, Mario jumps onto a Red Koopa, kicks its shell down, which bounces and breaks all the bricks in the corridor at the bottom, opening the path for later checking. Note that falling down is no use because Super Mario cannot fit into the narrow corridor unless he gets hurt by the Koopa, in which case he will not be able to reach the goal. There is not enough space for Mario to run and crouch-slide into the corridor. The gap at the bottom of the wide corridor is so the Koopa Shell does not unlock other clauses.

Finally, we implement the Crossover gadget, illustrated in Figure 6. There are four entrances/exits: top left, top right, bottom left, and bottom right. The Crossover gadget is designed so that, if Mario

5

Tetris

Given a sequence of Tetris pieces, and a number k, can you clear more than k lines?

How do you show a language is **NP**-complete?

Seems like an unbelievably strong statement.

How could one possibly prove such a thing?!?

NP $\leq_T^P$ SAT

Once you have one, things are easier.

If SAT $\leq_T^P$ L, then L is **NP**-hard.

(transitivity of $\leq_T^P$)

How do you show a language is **NP**-complete?

It is similar to showing undecidability.

- we need an initial direct proof that a language is **NP**-hard. (Cook-Levin Theorem)

- to show other languages are **NP**-hard, we use poly-time reductions.

This is the topic of Thursday's lecture.

Good evidence for intractability?

If A is **NP**-hard,
that seems to be good evidence that $A \notin \mathrm{P}$.

(if you believe **P** ≠ **NP**)

But is **P** ≠ **NP**??

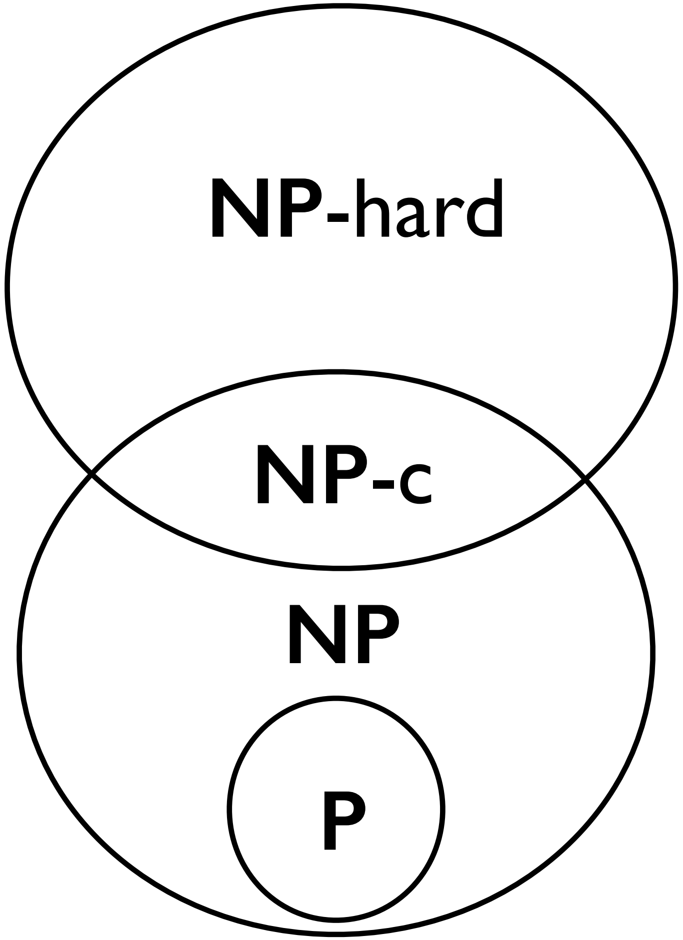
The P vs NP Question

The **P** vs **NP** question

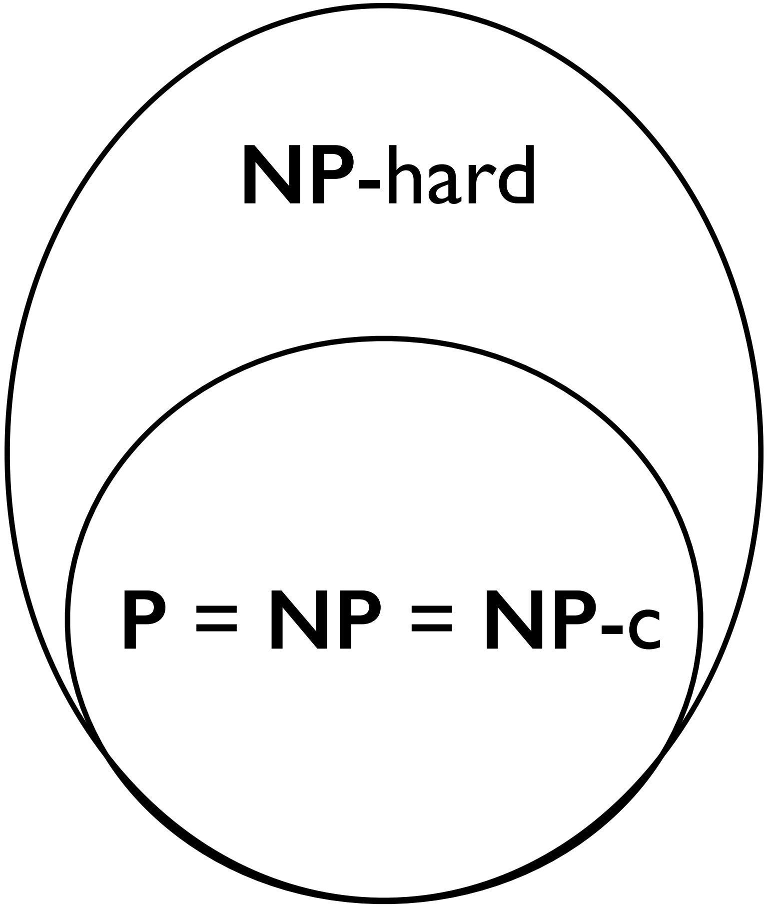
After years of research:

We are pretty confident that this is one of the deepest questions we have ever asked.

The two possible worlds



P ≠ NP



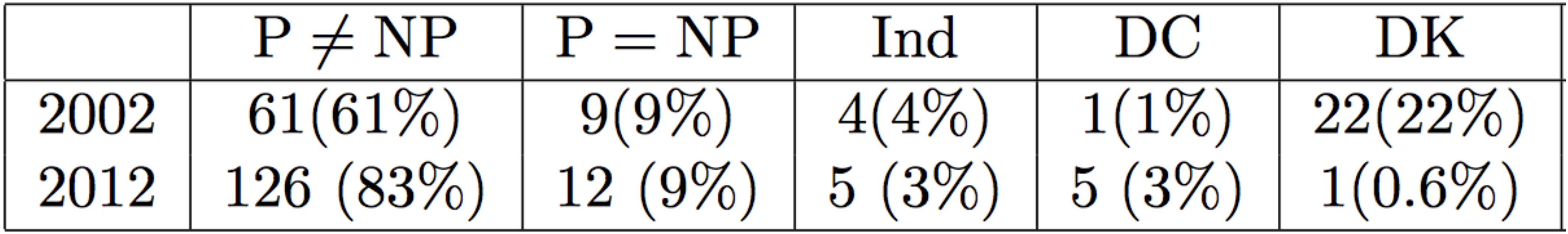
P = NP

What do experts think?

Two polls from **2002** and **2012**

respondents in **2002**: 100

respondents in **2012**: 152



	$P \neq NP$	$P = NP$	Ind	DC	DK
2002	61(61%)	9(9%)	4(4%)	1(1%)	22(22%)
2012	126 (83%)	12 (9%)	5 (3%)	5 (3%)	1(0.6%)

What does NP stand for anyway?

Not Polynomial?

None Polynomial?

No Polynomial?

No Problem?

Nurse Practitioner?

It stands for Nondeterministic Polynomial time.

Languages in NP are the languages decidable in polynomial time by a nondeterministic TM.

DFA ⟷ SFA (actually called NFA)

TM ⟷ NTM

What does NP stand for anyway?

Other contenders for the name of the class:

- Herculean
- Formidable
- Hard-boiled
- PET “possibly exponential time”
  - “provably exponential time”
  - “previously exponential time”

Summary

Summary

- How do you identify intractable problems?
  (problems not in **P**) e.g. SAT, TSP, …

- We are not able to prove they are intractable.
  Can we gather some sort of evidence?

- Poly-time reductions $A \leq_T^P B$ are useful to compare hardness of problems.

- Evidence for intractability of A:
  Show $L \leq_T^P A$, for **<u>all</u>** $L \in$ **C**, for a large class **C**.

- Definitions of **C**-hard, **C**-complete.

- What is a good choice for **C**,
  if we want to show, say, SAT is **C**-hard?

Summary

- The complexity class **NP** (take **C** = **NP**)
- **NP**-hardness, **NP**-completeness
- Cook-Levin Theorem: SAT is **NP**-complete
- Many other languages are **NP**-complete.
- If L is **NP**-hard, is this good evidence it is intractable (i.e., L not in **P**)?
- The **P** vs **NP** question

Next Time

How did Cook-Levin show SAT is **NP**-complete?

And examples of poly-time reductions that show other problems are **NP**-complete.