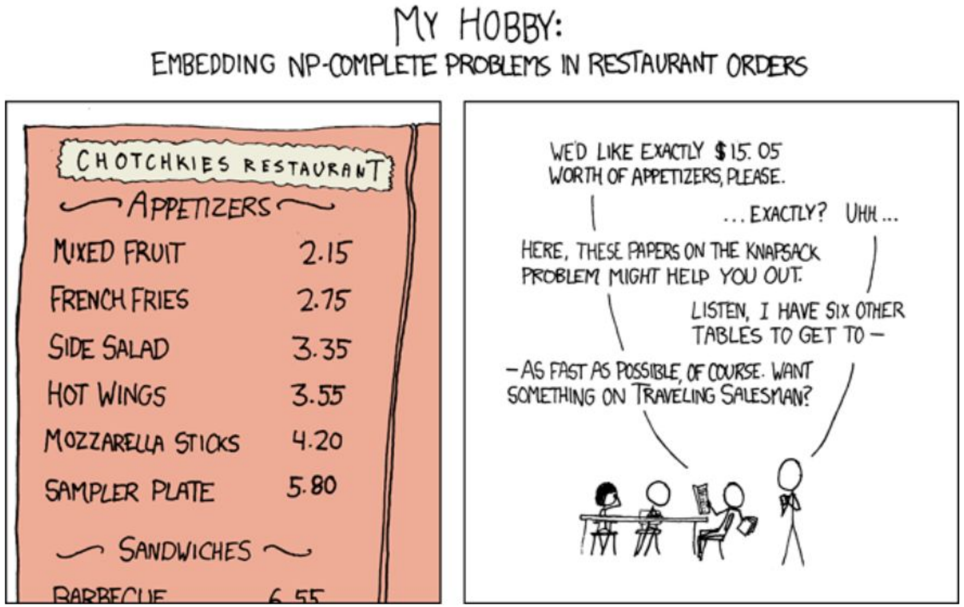

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Lecture 39

Complexity and P = NP (170 Preview)

CS61B, Spring 2024 @ UC Berkeley

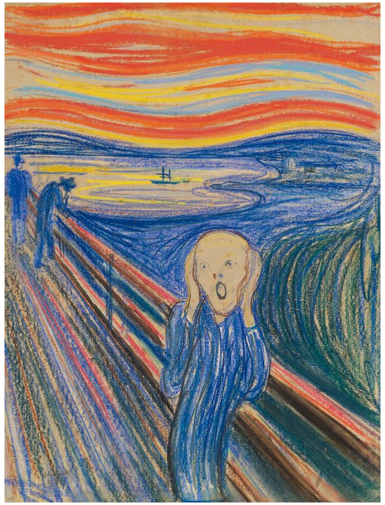
Peyrin Kao and Justin Yokota

Warmup: Reductions Practice

Lecture 39, CS61B, Spring 2024

The Knapsack Problem

You are a thief, and are planning a heist on a museum. The museum has a large number of items, of various weights and monetary values. Your goal is to steal the set of items with the most total value. However, you can only carry up to 10 pounds worth of stuff in your knapsack (and cannot make multiple trips). What items do you steal?



Knapsack Problem: Example

Your knapsack can store up to 10 lbs of material.

Which items do you steal?

Item	Weight (lbs)	Value ($)
Stamp	1	10,000,000
Crown	3	20,000,000
Painting	5	10,000,000
Diamond	2	1,000,000
Pebble	1	1

Knapsack Problem: Example

Your knapsack can store up to 10 lbs of material.

Which items do you steal?

Stamp + Crown + Painting + Pebble:

Weight = 1+3+5+1 <= 10 lbs

Value = 10M+20M+10M+1 = $40,000,001

Item	Weight (lbs)	Value ($)
Stamp	**1**	**10,000,000**
Crown	**3**	**20,000,000**
Painting	**5**	**10,000,000**
Diamond	2	1,000,000
Pebble	**1**	**1**

The Knapsack Problem

The knapsack problem can actually be described as 3 different problems:

1. Given a list of items and a weight limit, return the list of items you should steal (ex. Items, 10 -> [Stamp, Crown, Diamond, Pebble])
2. Given a list of items and a weight limit, return the most value of items you could steal (ex. Items, 10 -> 40,000,001)
3. Given a list of items, a weight limit, and a target value, return True if you can steal that amount of stuff or more, and False if you can't (ex. Items, 10, 40000001 -> True;

   Items, 10, 40000002 -> False)

Item	Weight (lbs)	Value ($)
Stamp	**1**	**10,000,000**
Crown	**3**	**20,000,000**
Painting	**5**	**10,000,000**
Diamond	2	1,000,000
Pebble	**1**	**1**

Stamp + Crown + Painting + Pebble:

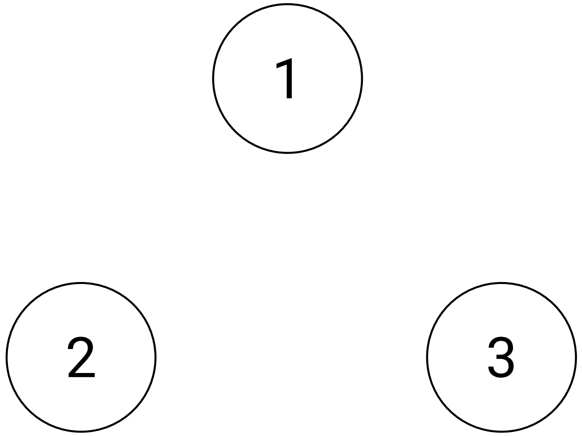
Weight = 1+3+5+1 <= 10 lbs

Value = 10M+20M+10M+1 = $40,000,001

The Knapsack Problem

The knapsack problem can actually be described as 3 different problems:

1. Return the list of items you should steal
2. Return the most value you could steal
3. Return if you can steal at least a target value

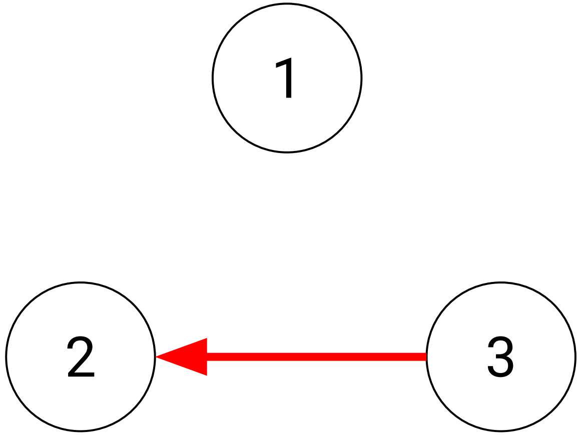


Let's show that these problems are *equivalent* via reductions.

- That is, if we had an oracle that could solve problem 1 in constant time, we could write an algorithm to solve problem 2 in polynomial time, and vice versa

Knapsack: Solving 3 given an oracle to 2

3. Return if you can steal at least a target value
2. Return the most value you could steal

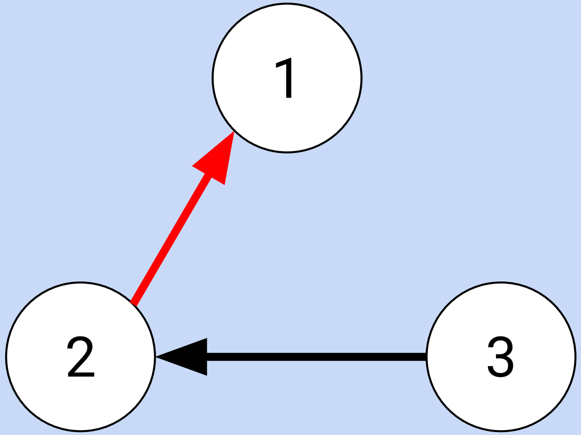


Ask the oracle the most value you could steal

If that amount is greater than or equal to the target, return True. Otherwise return False.

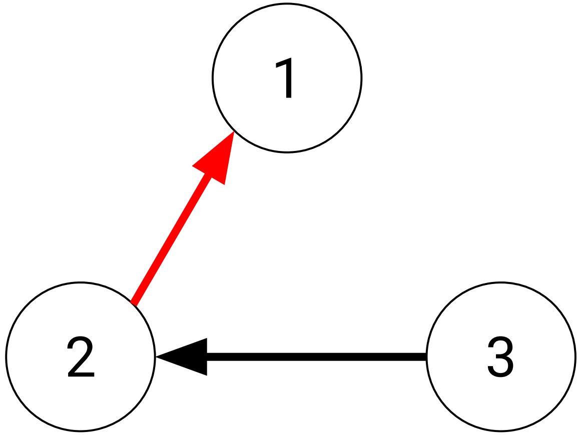
Knapsack: Solving 2 given an oracle to 1

2. Return the most value you could steal
1. Return the list of items you should steal



Knapsack: Solving 2 given an oracle to 1

2. Return the most value you could steal
1. Return the list of items you should steal

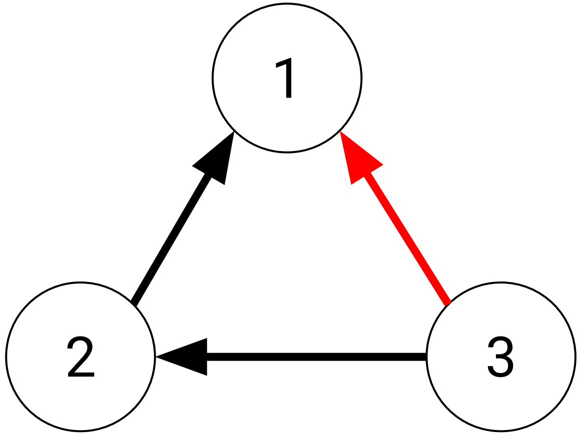


Ask the oracle for the list of items to steal

Return the sum of the values of the items

Knapsack: Solving 3 given an oracle to 1

3. Return if you can steal at least a target value
1. Return the list of items you should steal

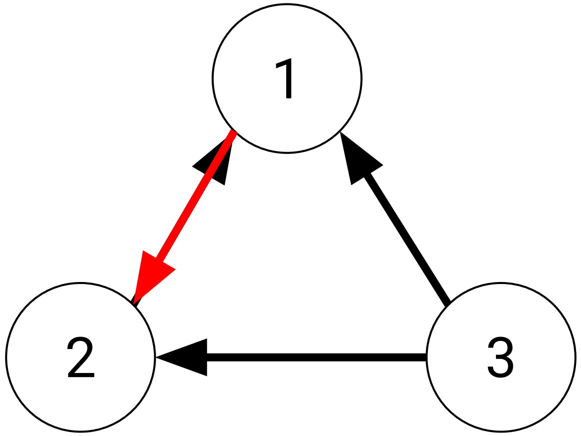


Make a solver to 2 that uses the solver to 1

Then make a solver to 3 using the solver to 2

Knapsack: Solving 1 given an oracle to 2

2. Return the most value you could steal
1. Return the list of items you should steal



(The trickiest one of the set)

Call the oracle on the original list of items, let that value be K

For each item: (Θ(N) total work)

Remove that item from the list, then call the oracle on the new list.

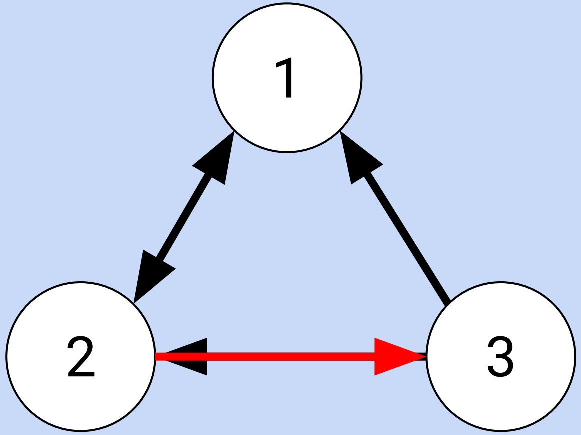
If the oracle returns K, then we can remove the item from the list permanently

If the oracle returns less than K, add the item back to the list

Return the remaining list of items.

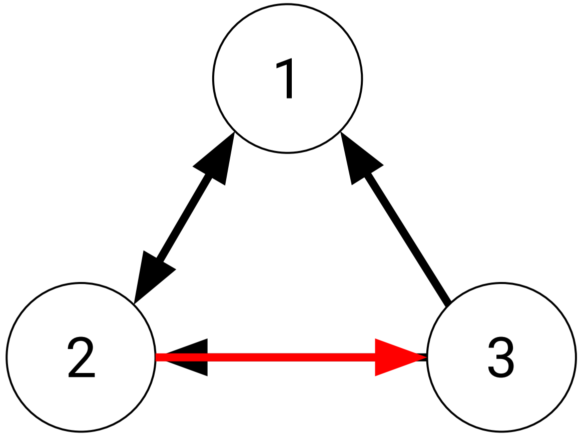
Knapsack: Solving 2 given an oracle to 3

2. Return the most value you could steal
3. Return if you can steal at least a target value



Knapsack: Solving 2 given an oracle to 3

2. Return the most value you could steal
3. Return if you can steal at least a target value



Binary Search:

Compute V, the total value of the items

Ask the oracle if you can steal V/2 value

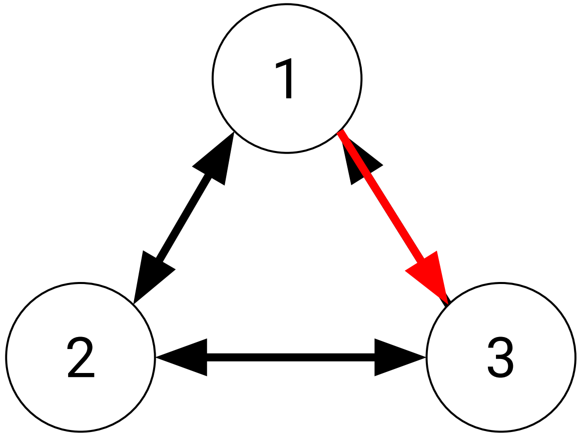
If yes, try 3V/4. If not, try V/4.

Continue until you narrow down the max value.

Max runtime: Θ(log(V))

Knapsack: Solving 1 given an oracle to 3

1. Return the list of items you should steal
3. Return if you can steal at least a target value

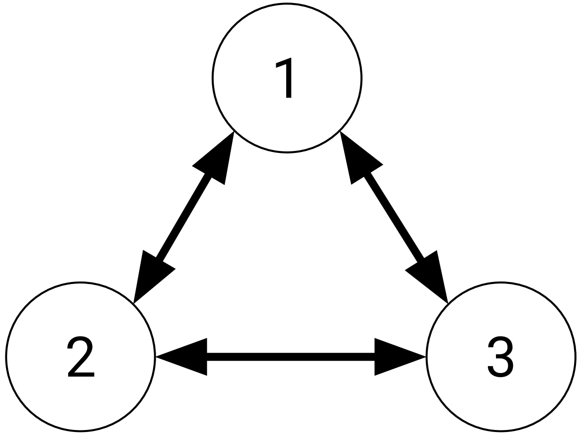


Make a solver to 2 using 3

Then make a solver to 1 using 2

The Knapsack Problem

1. Return the list of items you should steal
2. Return the most value you could steal
3. Return if you can steal at least a target value



If we have a solution to any one of these three problems, we can create a solution to the other two. Thus, we can consider these three problems to be *equivalent under Turing Reduction*.

Decision Problems

1. Return the list of items you should steal: Returns a List
2. Return the most value you could steal: Returns an integer
3. Return if you can steal at least a target value: Returns a boolean

The versions of the Knapsack problem all returned different types.

1's return value contains the most information, while 3 returns the least. In fact, #3 returns the least possible information of any nontrivial function (a True/False answer)

We will call functions that return T/F ***decision problems***. For the rest of this lecture, we'll consider computers that are designed to solve decision problems **only**.

But in general, we can find a reduction from a function problem (ex. What should I steal) to a decision problem (ex. Can you steal at least this much?)

Deterministic Turing Machine vs Nondeterministic Turing Machine

Lecture 39, CS61B, Spring 2024

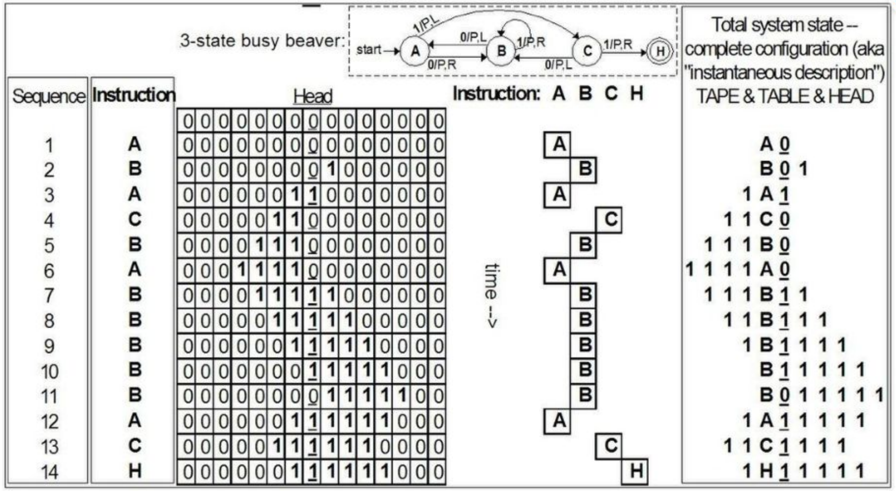
Turing Machines

Formally, any decision problem in theoretical CS is said to run on a **Turing Machine**, which represents a model of **universal computation**; anything that can be solved by a computer program can also be solved by a Turing Machine.

- Incidentally, when we talk about the time and space complexity of a Java program, the formal definition actually uses the equivalent Turing machine for precise definitions of "one unit of time" and "one unit of memory"

One major difference between a Java/Python program and a Turing machine is that the Turing machine has infinite memory

- This is why we can increase input size arbitrarily large when doing asymptotic analysis; won't run into practical bounds



Sequence	Instruction	Head	Total system state -- complete configuration (aka "instantaneous description") TAPE & TABLE & HEAD
		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
1	A	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	A 0
2	B	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0	B 0 1
3	A	0 0 0 0 0 0 1 1 0 0 0 0 0 0 0	1 A 1
4	C	0 0 0 0 0 1 1 0 0 0 0 0 0 0 0	1 1 C 0
5	B	0 0 0 0 1 1 1 0 0 0 0 0 0 0 0	1 1 1 B 0
6	A	0 0 0 1 1 1 1 0 0 0 0 0 0 0 0	1 1 1 1 A 0
7	B	0 0 0 0 1 1 1 1 1 0 0 0 0 0 0	1 1 1 B 1 1
8	B	0 0 0 0 0 1 1 1 1 1 0 0 0 0 0	1 1 B 1 1 1
9	B	0 0 0 0 0 0 1 1 1 1 1 0 0 0 0	1 B 1 1 1 1
10	B	0 0 0 0 0 0 0 1 1 1 1 1 0 0 0	B 1 1 1 1 1
11	B	0 0 0 0 0 0 0 0 1 1 1 1 1 0 0	B 0 1 1 1 1 1
12	A	0 0 0 0 0 0 1 1 1 1 1 1 0 0 0	1 A 1 1 1 1 1
13	C	0 0 0 0 0 1 1 1 1 1 1 0 0 0 0	1 1 C 1 1 1 1
14	H	0 0 0 0 0 0 1 1 1 1 1 1 0 0 0	1 H 1 1 1 1 1

Progress of the computation (state-trajectory) of a 3-state busy beaver

Turing Machines

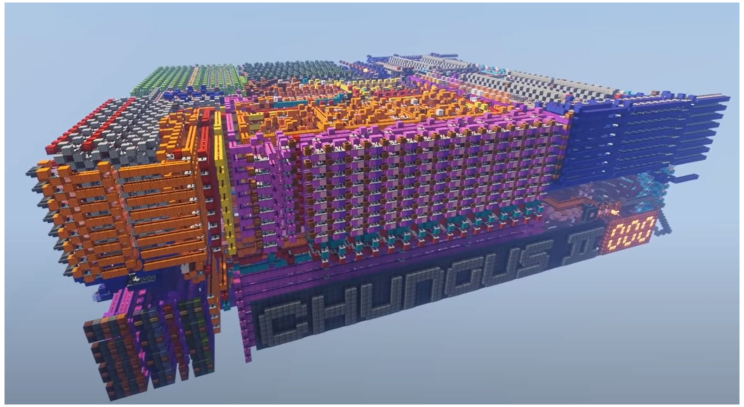
A programming language is said to be **Turing-Complete** if any Turing machine can be simulated with a program written in that language.

- In other words, that language can also compute anything that can be computed

Most programming languages in common use are Turing-Complete (ex. Python, Java, C, C++, Javascript, etc.)

But there are also some "languages" that are Turing-Complete by complete chance, like Minecraft's redstone.

- People have created computers out of redstone, and this is only possible because redstone is Turing Complete.



Deterministic Turing Machines

Because the two are equal, we can informally think of a Turing machine as just any function we could write in Java (or whatever programming language you want) that returns only a bool (we need to restrict to decision problems).

We will call this a **deterministic Turing Machine** (A Turing machine with no randomness involved), or a **DTM**.

Further, we will define the set **P** as the set of all decision problems that can be solved with a deterministic Turing Machine in **polynomial time**.

Examples of problems in P:

Is this array of length N sorted?

Does there exist a spanning tree of this graph (with N nodes) smaller than X weight?

Is this number (that's N bits long) prime? (Solved in 2002!!!)

Nondeterministic Turing Machines

A **Nondeterministic Turing Machine** or **NTM** is a Turing Machine with one additional operation that can be performed: guessing.

Informally, we allow an operation `int guess(int min, int max)` that returns a number between `min` and `max`.

If ANY pattern of guesses ends up causing us to return True, then the nondeterministic Turing Machine returns True.

If ALL guess patterns return False, then the nondeterministic Turing Machine returns False.

The set **NP** is the set of problems that can be solved in **polynomial time** with a NTM

Problems in NP

Lecture 39, CS61B, Spring 2024

The problem of Sudoku

A sudoku is a logic puzzle like the one on the right

The goal is to write the numbers 1-9 in each cell such that:

- Each row has the numbers 1-9
- Each column has the numbers 1-9
- Each 3x3 cell has the numbers 1-9

Decision problem version: Given a partially filled grid like the one to the right, does at least one solution exist?

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

Solving Sudoku with a NTM

Here's how a NTM might solve Sudoku:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

Step 1 creates 9^{47} distinct "universes". If even one of these universes returns True in step 2, the NTM will return True. If all the universes return False, the NTM will return False.

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

Solving Sudoku with a NTM

Universe 1: All 1s

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns False

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

Solving Sudoku with a NTM

Universe 2:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns False

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

Solving Sudoku with a NTM

Universe 3:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns False

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

Solving Sudoku with a NTM

Universe 385585354550914402442800888282404424608060642:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

What does this universe return?

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

Solving Sudoku with a NTM

Universe 385585354550914402442800888282404424608060642:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns False

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

Solving Sudoku with a NTM

Universe 385585354550914402442800888282404424608060643:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

What does this universe return?

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

Solving Sudoku with a NTM

Universe 385585354550914402442800888282404424608060643:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns True!

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

Solving Sudoku with a NTM

Universe 706965049015104706497203195837614914543357369:

Step 1:

For each of the 47 blank cells, make a guess as to the right value in each cell

Step 2:

Check if all the conditions of a valid Sudoku solution holds. If they hold, return True. If not, return False.

This universe returns False

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

Solving Sudoku with a NTM

In universe 385585354550914402442800888282404424608060643, we returned True, so our NTM would return True.

What's the runtime (when run on generalized Sudoku of size n^2 by n^2)? (Note that because we are running a NTM, all the universes happened simultaneously; our runtime is the universe that took the most time to return True or False)

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

Showing that Sudoku is in NP

In universe 3855853545509144024428008882824044246080606433, we returned True, so our NTM would return True.

What's the runtime (when run on generalized Sudoku of size n^2 by n^2)?

It takes $\Theta(n^4)$ time to do step 1 (one step per square)

You can check if a set of n^2 items contain all the numbers 1 to n^2 in $\Theta(n^2)$ using a set. We do this $3n^2$ times (once for each row, column, and cell), so it takes $\Theta(n^4)$ time to do step 2

Therefore, the total runtime is $\Theta(n^4)$, and Sudoku is in NP.

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

Is the Knapsack Problem in NP?

Is Knapsack 3 in NP?

3. Given a list of N items, a weight limit, and a target value, return True if you can steal that amount of stuff or more, and False if you can't (ex. Items, 10, 40000001 -> True;

Items, 10, 40000002 -> False)

Item	Weight (lbs)	Value ($)
Stamp	1	10,000,000
Crown	3	20,000,000
Painting	5	10,000,000
Diamond	2	1,000,000
Pebble	1	1

Is the Knapsack Problem in NP?

Is Knapsack 3 in NP?

3. Given a list of N items, a weight limit, and a target value, return True if you can steal that amount of stuff or more, and False if you can't (ex. Items, 10, 40000001 -> True;

Items, 10, 40000002 -> False)

Item	Weight (lbs)	Value ($)
Stamp	1	10,000,000
Crown	3	20,000,000
Painting	5	10,000,000
Diamond	2	1,000,000
Pebble	1	1

Yes! Strategy:

Step 1: Guess a set of items to steal (creates 2^N universes)
Step 2: Verify that the set solves the problem: If that set of items has less weight than our weight limit (takes Θ(N) time to compute) and the total value is greater than the target (takes Θ(N) time to compute), return True. Otherwise, return False.

NP == Polynomial-Time Verifiable

In general, solving a problem with a NTM boils down to those two steps:

1. Generate (nondeterministically) a random solution to the problem
2. Verify (deterministically) if that solution actually solves the problem

Nontrivial fact: Any problem that can be solved by an NTM can also be solved in the above procedure (with the same runtime)

- Short justification: Can "move" all the guesses to the start of the program and save those guess values in memory somewhere

Because of this, NP is also defined as **the set of problems whose solution can be verified in polynomial time by a DTM**

- Given a proposed solution, can you check if that solution actually solves the problem efficiently?

Contrast to P, which is the set of problems whose solutions can be **generated** in polynomial time

More NP problems

Here are a few examples of problems in NP (after turning them into their respective decision problem):

- Any problem in P
- Knapsack
- Sudoku
- Most logic puzzles (Kenken, Minesweeper, Tetris, Rubik's Cube)
- Longest Path (Given a graph, find the longest path that doesn't reuse edges)
- Independent Set (Given a graph, find the largest set of vertices that have no adjacencies)
- Prime Factorization
- Encryption (Decrypt an encrypted bitstream)
- Compression (Optimally compress a bitstream, such that it can be uncompressed in polynomial time)

Problems NOT in NP

Here are a few examples of decision problems not currently known to be in NP:

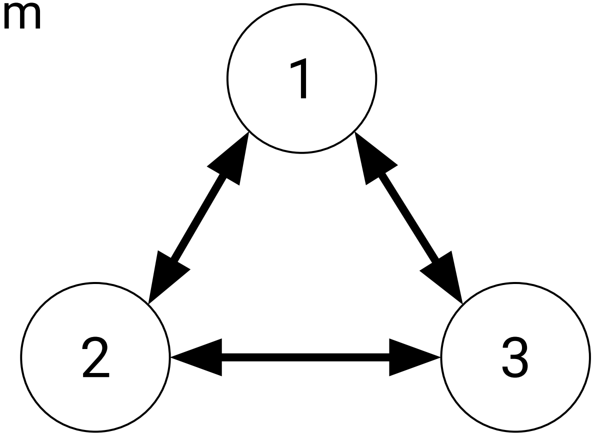
- Chess (Given a chess position, determine which side is winning, or if it's a draw assuming perfect play)
- Many other games, including Go
- Given a program and input, does the program terminate? (Halting Problem, known to be undecidable)
- Compression *without* an upper limit on runtime (reduces to the Halting Problem)

NP-Complete Problems

Lecture 39, CS61B, Spring 2024

NP Hardness

Earlier, we showed that 3 versions of the Knapsack problem were equivalent under Turing reductions.



If A reduced to B and B reduced to C, then A can always reduce to C.

Intuitively: if A reduces to B, then B is "at least as hard" as A to solve. If B is harder than A and C is harder than B, then C is harder than A.

In general, it's not true that if A reduces to B, then B reduces to A.

Ex. All problems reduce to the SOLVE decision problem, which receives as input a decision problem and an input, and returns whether the decision problem would return True or False on that input.

We will call a problem **NP-Hard** if every problem in NP reduces to that problem (or in other words, that problem is at least as hard as every problem in NP)

The Satisfiability Question (SAT)

The SAT problem is defined as follows: You're given a function of boolean variables (ex. "`x && y`", "`x && (!x)`"). Is it possible to assign values to all the variables, such that the boolean expression evaluates to True?

Ex. "`x && y`" can return True if we set x = True, y = True. So SAT should return True.

Ex. "`x && (!x)`" can never return True; both x = True and x = False yield False overall. So SAT should return False.

SAT is in NP (guess an assignment of all variables and see if it returns True)

Totally Shocking Fact

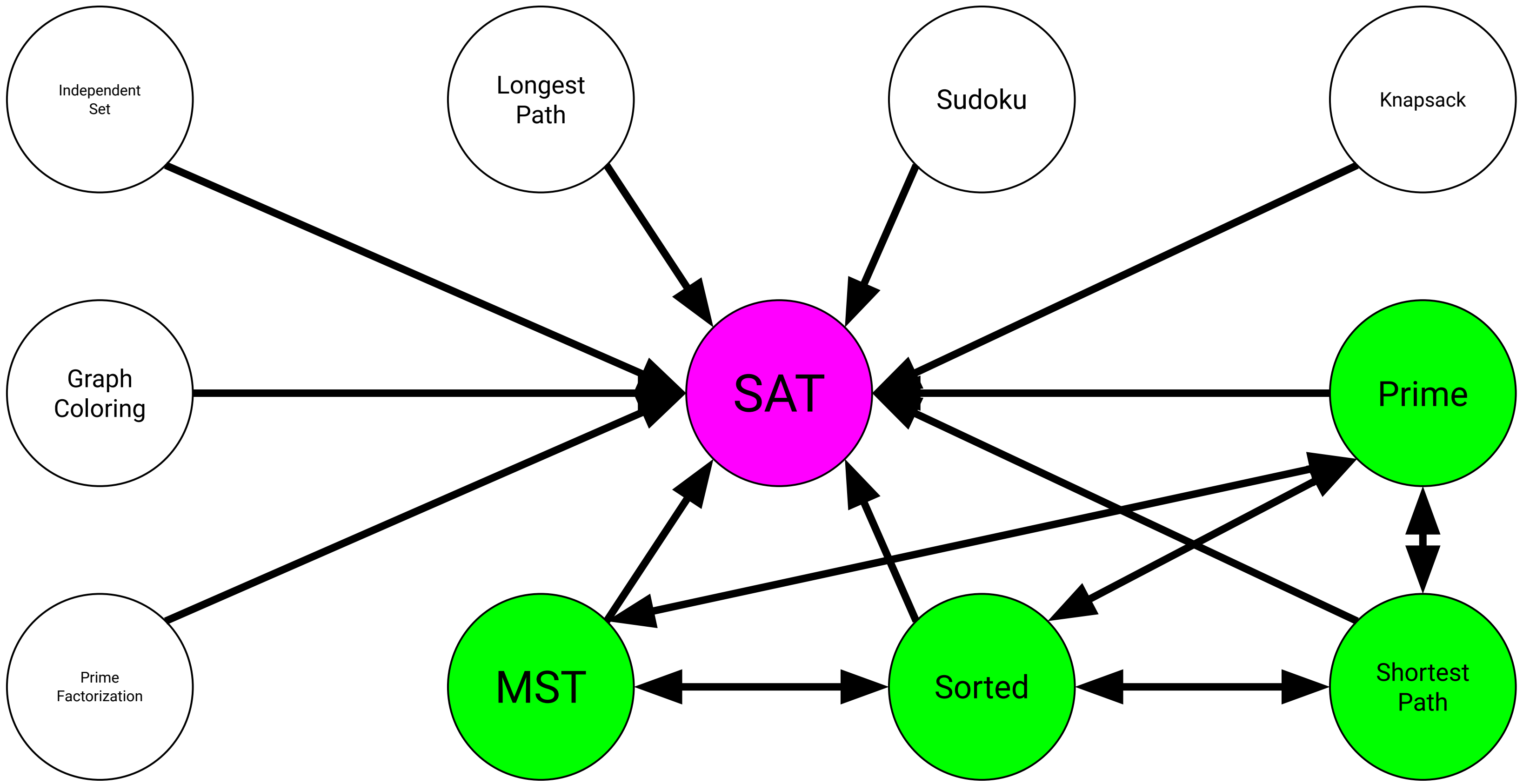
SAT is NP-Hard

In other words, **any decision problem that can be solved by a NTM** can be transformed into a SAT problem in polynomial time.

In addition, SAT is in NP. We call a problem that's both in NP and NP-Hard an NP-Complete problem. Any NP-Complete problem is the hardest problem in NP.

This result is by Cook (1971) and Levin (1973). See Cook-Levin Theorem for more.

The Reductions Graph



Pink problems are NP-complete

Green problems are in P

Even More Shocking Fact

Amazingly, SAT also can reduce to other problems in NP:

SAT reduces to 3-SAT (SAT with some restriction on the boolean formula)

SAT reduces to Independent Set

3-SAT reduces to Graph Coloring

Graph Coloring reduces to Exact Cover

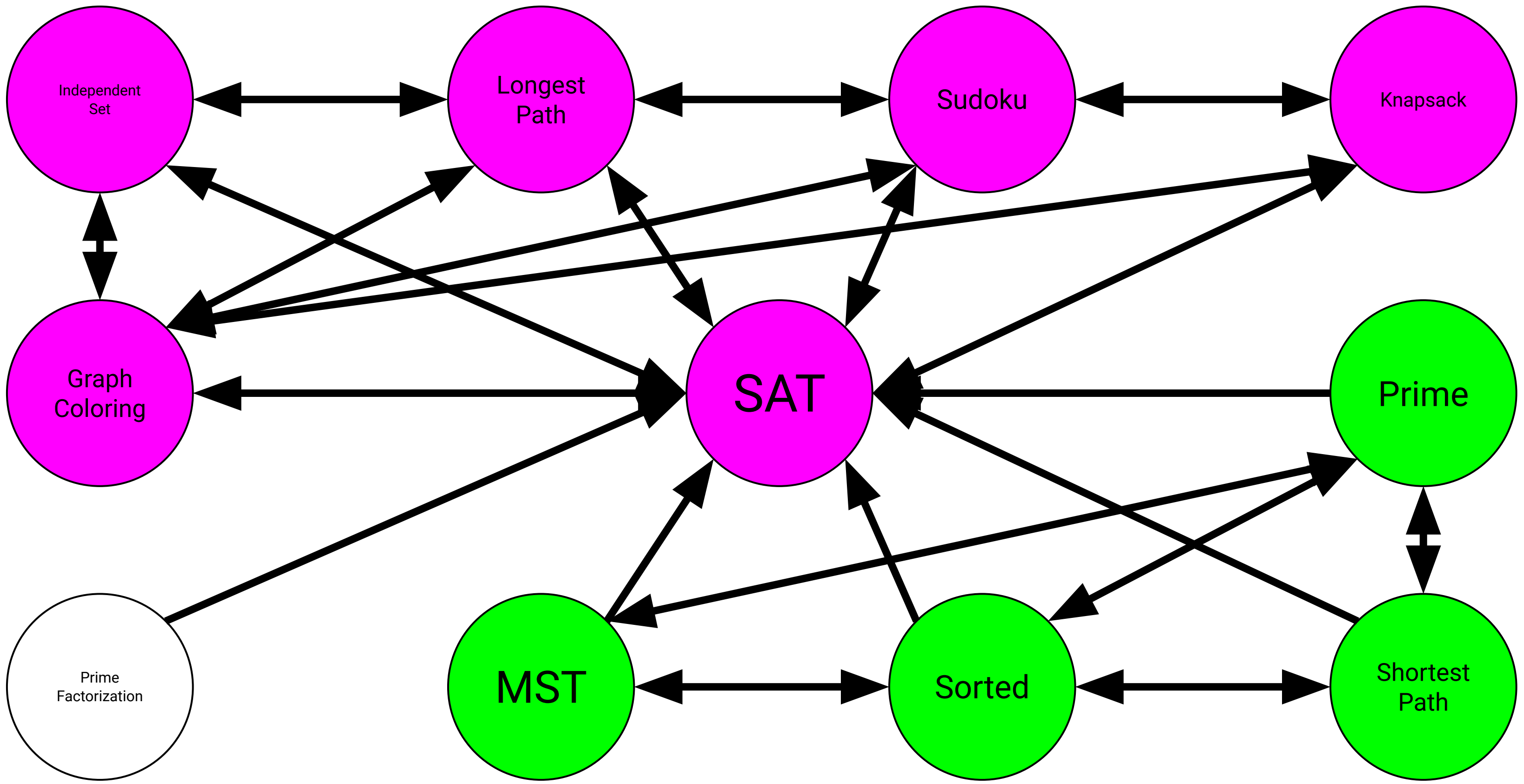
Exact Cover reduces to Knapsack

Independent Set reduces to Vertex Cover reduces to Hamilton Circuit reduces to Longest Path

A complex sequence of problems shows that 3-SAT reduces to Sudoku

Prime Factorization is NOT currently known to be NP-complete.

The Reductions Graph



Pink problems are NP-complete

Green problems are in P

The set of NP-Complete problems

Because of this, any two NP-Complete problems are equivalent under Turing reduction.

- In other words, the Knapsack problem is actually the same as solving a Sudoku!

There are tens of thousands of problems now known to be NP-Complete; many of these problems were independently discovered in completely different fields, well before a proof of NP-Completeness was discovered.

If even one of these problems has a reduction to a problem in P (or a polynomial time solution was discovered for it), then **every other NP problem will also be solved in polynomial time**.

At this point, no one has found a polynomial-time reduction of any of these problems, but no one has proven that no such reduction exists.

This is the **P=NP problem**: Find a Turing reduction from an NP problem to P (P = NP), or prove that no reduction exists (P != NP).

P = NP?

Lecture 39, CS61B, Spring 2024

P = NP?

Consensus Opinion (Bill Gasarch Poll, 2019 poll)

- 89%: P ≠ NP (109 respondents)
- 11%: P = NP (15 respondents)
- ~3% mentioned in comments an alternative: P = NP is provably impossible to prove OR disprove

Why is opinion generally negative?

- Someone would have proved it by now.
    - "The only supporting arguments I can offer are the failure of all efforts to place specific NP-complete problems in P by constructing polynomial-time algorithms." - Dick Karp
- Creation of a correct solution seems philosophically more difficult than verifying a solution.

P = NP?

Consensus Opinion (Bill Gasarch Poll, 2019 poll)

- 89%: P ≠ NP (109 respondents)
- 11%: P = NP (15 respondents)
- ~3% mentioned in comments an alternative: P = NP is provably impossible to prove OR disprove

Why do some people think P=NP is still possible?

- NP is surprisingly close to P.
  - We have algorithms that solve the Knapsack problem in polynomial time:
    - If N is defined as the total weight of the items, instead of the number of items
      - But only if the weights and values of items are integers, not rational numbers
    - If we want to find a solution within x% of optimal, for any x>0
  - Modern SAT solvers work in polynomial time in *almost all* randomly generated cases, and only exhibit exponential time in cases tailor-made to the algorithm

What happens if you prove P=NP?

If P = NP is proven (even if we find an algorithm that takes $\Theta(n^{1000000})$) time:

- Every problem in NP collapses into P
- All modern cryptography breaks
- A fundamental assumption made by 89% of CS theorists is broken, and further improvements are likely

If P != NP is proven:

- An entire new branch of theory will likely be developed off those methods
    - There are currently a LOT of unsolved problems in complexity hierarchy
- Despite hundreds of people working on this problem, there's been basically no progress on solving this in the past decade

The Millenium Problems

In 2000, the Clay Mathematics Institute set up $1,000,000 prizes for the solution (proof, disproof, or proof of independence from ZFC) of each of seven problems.

Millenium Prize Problems.

- Hodge conjecture
- Poincare conjecture (solved in 2002!)
- Riemann hypothesis
- Yang-Mills existence and mass gap
- Navier-Stokes existence and smoothness
- Birch and Swinnerton-dyer conjecture
- P=NP

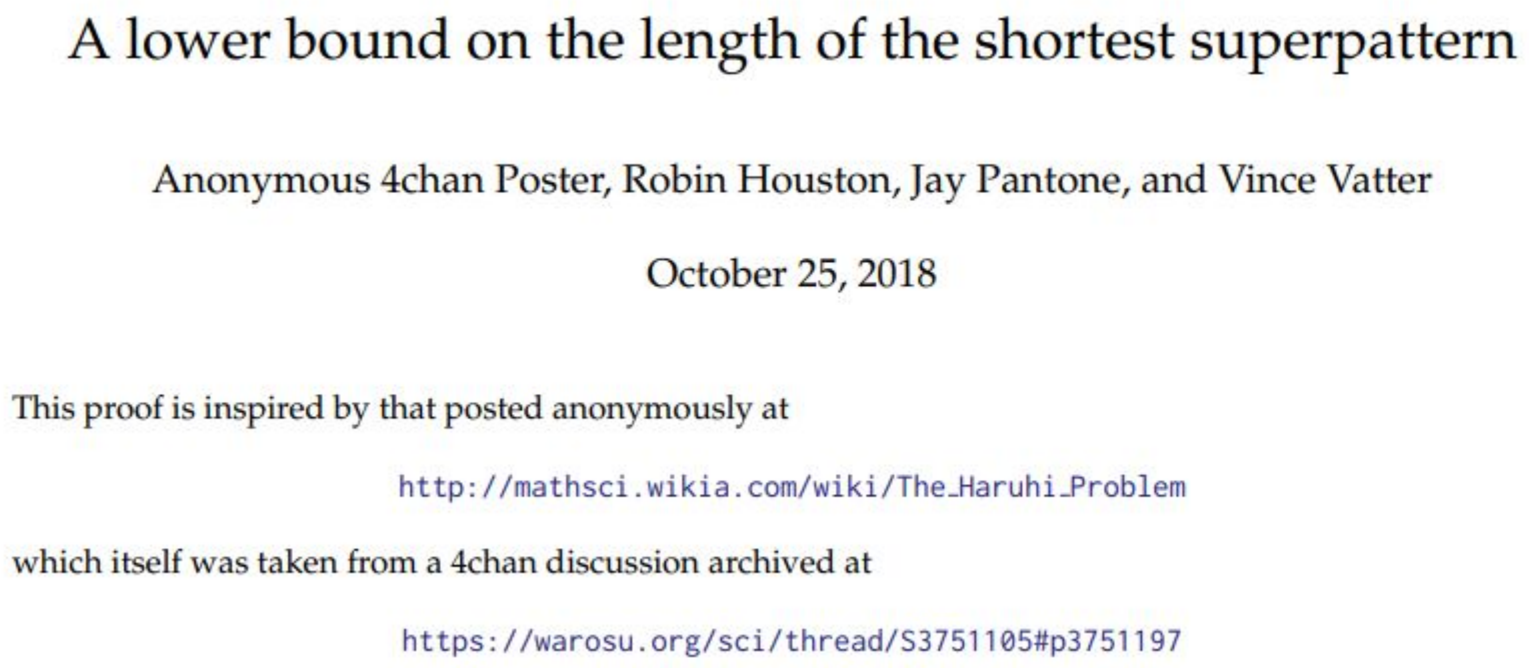
Just because a problem hasn't been solved doesn't mean you can't find a solution

Despite all I've said, you shouldn't be discouraged from finding a solution. Amateur theorists find massive breakthroughs in CS and math surprisingly often.

Example: Some fans of the anime "The Melancholy of Haruhi Suzumiya" ended up solving a major unsolved problem while trying to determine how many ways they could watch the series. The proof was written on 4chan in 2011, discovered by a mathematician in 2018, and published in 2021.

Sometimes, the best way to do CS is to just play around with stuff and see what happens.

Just because a problem hasn't been solved doesn't mean you can't find a solution



A lower bound on the length of the shortest superpattern

Anonymous 4chan Poster, Robin Houston, Jay Pantone, and Vince Vatter

October 25, 2018

This proof is inspired by that posted anonymously at

`http://mathsci.wikia.com/wiki/The_Haruhi_Problem`

which itself was taken from a 4chan discussion archived at

`https://warosu.org/sci/thread/S3751105#p3751197`