Limits of Computation

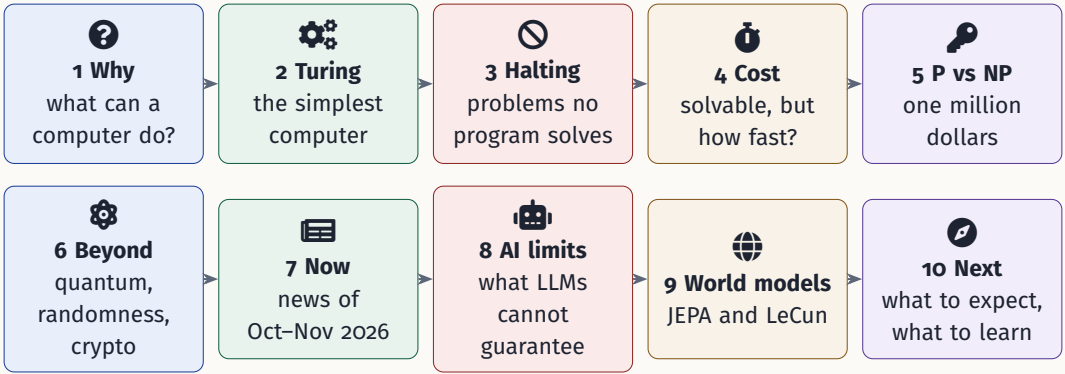
What computers cannot do, and what that means for AI

Dr. Quang-Vinh Dang
Senior Lecturer

Version of 23 November 2026

British University Vietnam
Introduction to Computing Science, BSc Data Science and AI, Year 1

The story of this lecture



Each part: a **question**, a **picture**, a small **demo** you can run, and one **take-away**. All code on the slides is real and was run with Python 3.14.

Two questions that sit under all of computing

Question 1: computability

Is there *any* program, however clever, that solves the problem for every input?

If not, no amount of money, data, GPUs or AI changes that.

Question 2: complexity

If a program exists, can it finish before the sun burns out?

Solvable in principle is not the same as solvable in practice.

Third question, for 2026: what do these limits say about *AI systems* that write code, prove theorems and claim discoveries?

A traffic light for problems

Easy for computers

- sort a million names
- shortest route on a map
- search the web
- check a password hash
- multiply two big numbers
- play a perfect game of noughts and crosses

polynomial time, class P

Solvable but explosive

- timetable for a whole university
- best tour through 60 cities
- pack parcels perfectly
- fold a protein exactly
- break a good password by guessing
- solve any Sudoku of any size

best known: exponential time

Impossible, provably

- will this program ever stop?
- is this program free of bugs?
- is this file a virus (always right)?
- do two programs do the same thing?
- does this code ever print “hello”?

undecidable: no algorithm exists

Hook: five lines, nobody knows if they always stop

```python
# Does this loop always stop? Nobody knows.
def steps(n):
    c = 0
    while n != 1:
        n = n//2 if n % 2 == 0 else 3*n + 1
        c += 1
    return c

for n in (6, 27, 97, 871):
    print(f"n={n:4d} stops after {steps(n):3d} steps")
best = max(range(1, 100000), key=steps)
print("slowest below 100000:", best,
      steps(best), "steps")
```

```
n=   6 stops after   8 steps
n=  27 stops after 111 steps
n=  97 stops after 118 steps
n= 871 stops after 178 steps
slowest below 100000: 77031 350 steps
```

real output, Python 3.14.6

Rule. Take any whole number n. If even, halve it. If odd, triple it and add 1. Repeat until you reach 1.

Checked. Every start up to about 2^{68} reaches 1 (computer searches).

Unknown. No proof that *every* start stops. This is the *Collatz conjecture*, open since 1937.

Open problem. Check bound (about 2^{68}): D. Barina, J. Supercomputing 2021.

Why should a first-year data-science student care?



Software

Why no tool finds all your bugs, and why tests, types and proofs each catch only some.



Data and AI

Training, clustering, scheduling and planning are often NP-hard. Knowing this tells you to reach for heuristics, not exact answers.

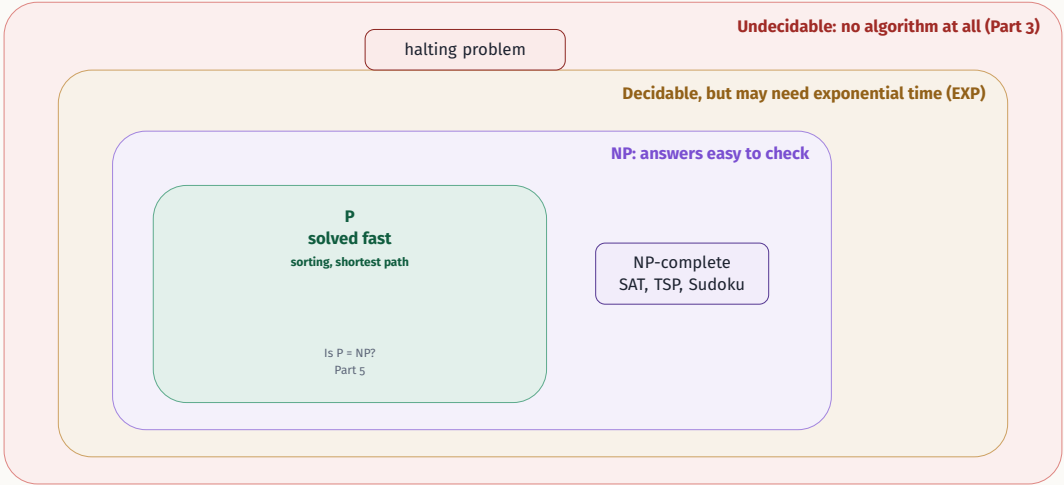


Security

Passwords, HTTPS and banking are safe only because some problems are hard. If that changes, the internet changes.

Career. Hype cycles come and go. Limits stay. People who know them judge claims (“AI solved X”) faster and better.

The map we will fill in



Warm-up poll: which of these can a computer always answer?

1. Is a given number prime?
2. Does a given program contain the word `while`?
3. Does a given program ever stop?
4. What is the shortest tour of 25 given cities?
5. Is this Sudoku solvable?
6. Does a given program contain a bug?

Think first, then check at the end of the lecture.

Three different things can go wrong: no algorithm at all, an algorithm that is too slow, or a question that is not precise enough.

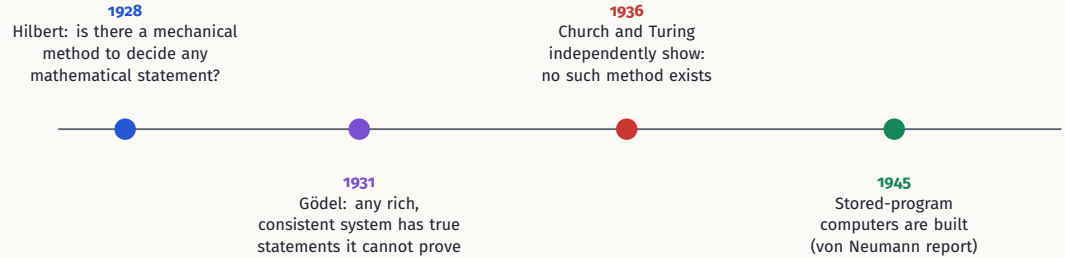
Answers: 1 yes, fast. 2 yes, trivial. 3 **no**. 4 yes but slow (exponential best known). 5 yes, fast for 9×9. 6 **no** (depends on what "bug" means; Rice).

Part 2

Turing machines

The simplest computer that can still compute everything

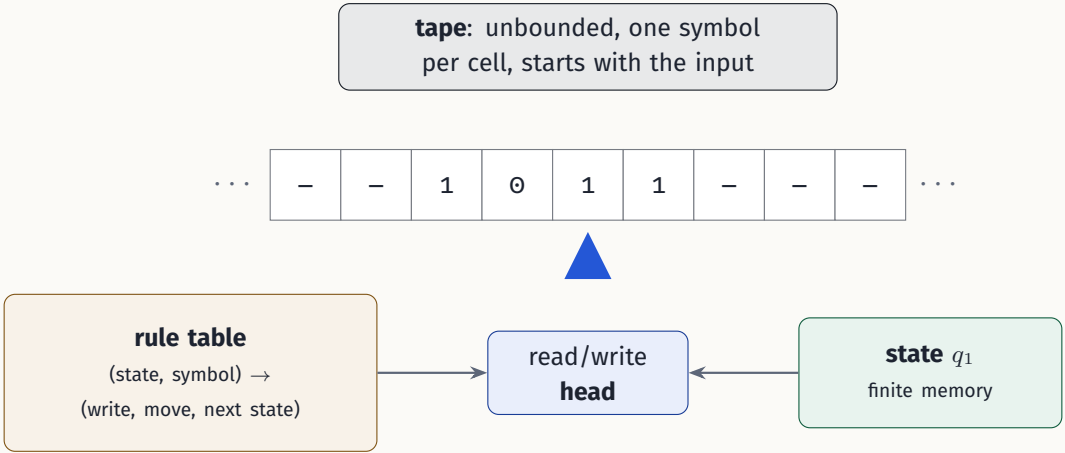
1936: the question behind every computer



In 1936 the word *computer* meant a **person** doing calculations with pencil and paper. Turing asked: what is such a person really doing, and what can they never do?

Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem”, Proc. London Math. Soc., 1936. Church, Amer. J. Math., 1936.

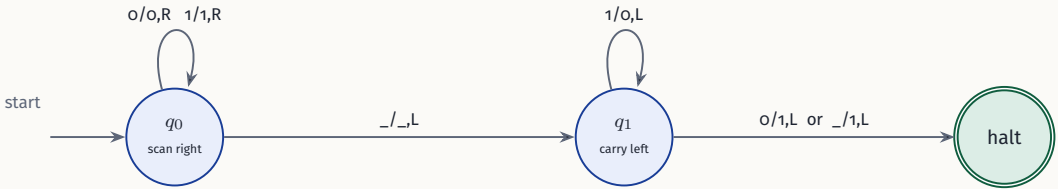
Anatomy of a Turing machine



Each step: read the cell under the head, **write** a symbol, **move** left or right, switch to a **new state**.

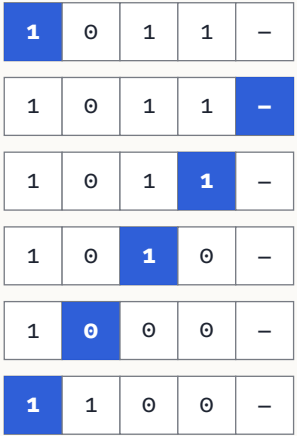
Halt when the state is `halt`.

Example 1: add one in binary, as a state diagram



Arrow label `read/write,move`. Idea: run to the right end, then turn every trailing `1` into `0` until you find a `0` (or blank) to turn into `1`.

Example 1 in motion: 1011 + 1 = 1100



step 0: q_0, input 11

step 4: q_0 has reached the blank

step 5: q_1 turns around

step 6: 1+1 = 0, carry

step 7: 1+1 = 0, carry

step 8: 0 becomes 1, **halt**: 12

Blue cell = head. Eight steps for four bits. These are the real first and last states of the simulator on the next slide.

A Turing machine simulator in 16 lines of Python

```
# A tiny Turing machine simulator.
def run(rules, tape, state="q0", blank="_",
        show=False, limit=500):
    t = dict(enumerate(tape)); h = 0; n = 0
    while state not in ("halt", "reject"):
        s = t.get(h, blank)
        w, d, state = rules[(state, s)]
        t[h] = w; h += 1 if d == "R" else -1
        n += 1
        if show:
            print(n, state, "".join(
                t.get(i, blank) for i in
                range(min(t), max(t) + 1)))
        if n > limit: return None, n, "limit"
    out = "".join(t[i] for i in sorted(t))
    return out.strip(blank), n, state
```

The tape is a dictionary: position → symbol. Unlimited, blank by default.

The rules are a dictionary too: `(state, symbol)` → `(write, move, next)`.

The loop is the whole machine: look up, write, move, switch state. Nothing else.

Run it: binary increment, real output

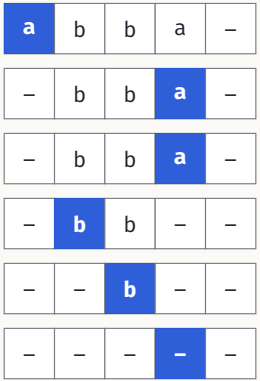
```
# Binary increment: go right, then carry left.
inc = {
 ("q0","0"):("0","R","q0"),
 ("q0","1"):("1","R","q0"),
 ("q0","_"):("_","L","q1"),
 ("q1","1"):("0","L","q1"),
 ("q1","0"):("1","L","halt"),
 ("q1","_"):("1","L","halt")}
```

```
increment 1011 (=11):
1 q0 1011
2 q0 1011
3 q0 1011
4 q0 1011
5 q1 1011_
6 q1 1010_
7 q1 1000_
8 halt 1100_
('1100', 8, 'halt')
0 -> 1
111 -> 1000
1011 -> 1100
```

real output, Python 3.14.6

The simulator prints the tape after each step (without the head marker). It matches the hand trace on the previous slide: 1011 → 1100. **Warning**: the machine never “knows” it is adding. It only follows rules.

Example 2: is this word a palindrome?



abba: remember the first letter, erase it

run to the right end, step back

last letter equals the remembered one? erase it

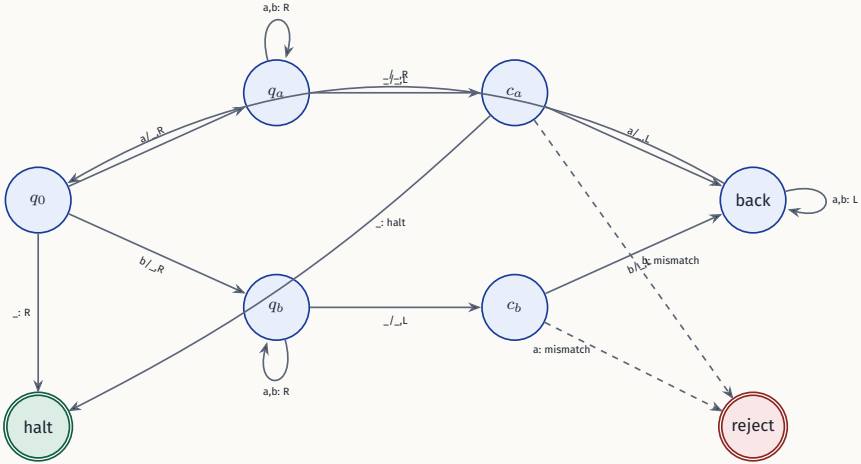
run back to the left end

repeat on the shorter word: b, b

empty tape: **halt, accept**

The **state** is the machine's memory: "I erased an `a`" or "I erased a `b`". Mismatch means **reject**. A finite number of states handles words of any length.

Palindrome machine: the state diagram



Palindrome machine: run it

```
# Palindrome over {a,b}: erase ends, compare.
pal = {
 ("q0","a"):("_","R","qa"),
 ("q0","b"):("_","R","qb"),
 ("q0","_"):("_","R","halt"),
 ("qa","a"):("a","R","qa"),
 ("qa","b"):("b","R","qa"),
 ("qa","_"):("_","L","ca"),
```

… 18 rules in all, in `code/tm.py`

```
'abba' halt steps 15
'abab' reject steps 6
'aba' halt steps 10
'' halt steps 1
```

real output, Python 3.14.6

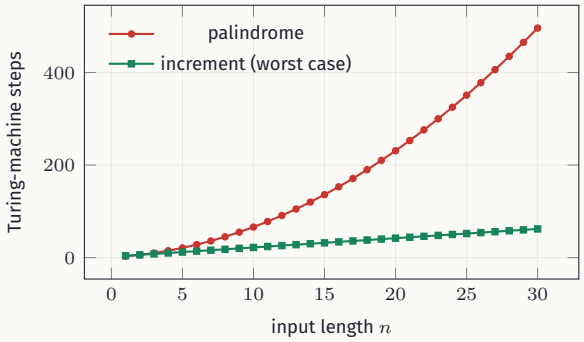
abba: accepted in 15 steps.
aba: accepted in 10.

abab: rejected after 6 steps. The first mismatch ends it.

An **empty word** is a palindrome: halt in 1 step.

Yes, a laptop does this in one line: `w == w[::-1]`. Turing machines are not fast. They are *simple enough to reason about*.

Counting steps: the first look at cost



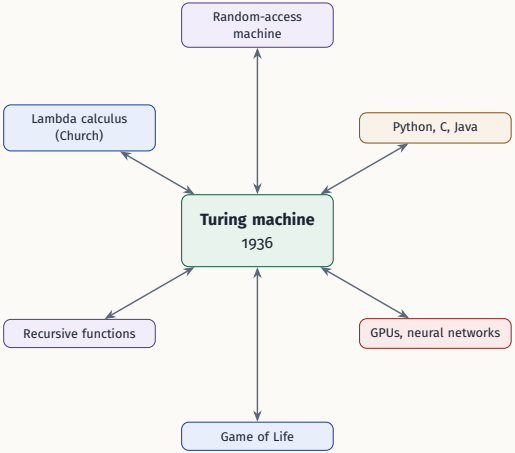
len	palindrome steps	increment steps
2	6	6
4	15	10
8	45	18
16	153	34
30	496	62

Doubling n doubles the increment cost (linear) but roughly **quadruples** the palindrome cost: 153 to 496 steps when n goes 16 to 30. The head runs back and forth, about $n^2/2$ steps. Same task, different machine, different cost.

The Church–Turing thesis

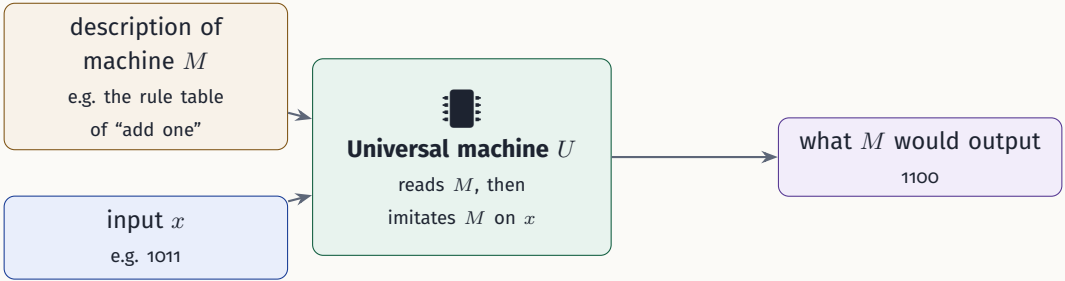
Thesis. Anything that can be computed by a step-by-step mechanical procedure can be computed by a Turing machine.

- A **thesis**, not a theorem: "mechanical procedure" is informal.
- Every serious formalisation turned out equivalent; no counter-example in 90 years.
- **Careful:** it concerns *what* is computable, not *how fast*. The *extended* thesis (polynomial equivalence) is a stronger claim, which quantum computing challenges (Part 6).



Each model can simulate the others: all compute the same functions.

The universal machine: programs are data



One fixed machine runs *any* program given as input. That is the idea of the **stored-program computer**: your laptop is a universal machine, an app is its input.

And the trick that will break things next: a **program can be fed to a program**, including itself. A simulator like `run(rules, tape)` on the slide before is exactly this.

Check yourself: trace a machine

state	sees	write, move	next
s	1	0, R	s
s	0	1, R	s
s	_	_, R	halt

1. Start in `s` on the left of `1101`. What is on the tape at `halt`?
2. What does this machine do in one phrase?
3. How many steps for an input of length n?
4. Change one rule so that it never halts. What did you create?

Hint for question 4.
Make the machine move right forever on blanks:

```
(s, _) -> (_, R, s)
```

This is the smallest possible **infinite loop**, and the star of the next part.

Try it: edit the `inc` table in `code/tm.py` and run `python3.14 tm.py`. Answers: 0010, flip every bit, $n + 1$.

Part 3

The halting problem

Questions about programs that no program can answer

First hint: there are more problems than programs

Programs are finite strings. List them by length: “a”, “b”, …, “aa”, “ab”, …. Every program appears at some position: they can be **counted** (countably many).

Yes/no problems on numbers are infinite 0/1 sequences, one answer per number. Cantor (1891): such sequences *cannot* be listed. There are **uncountably many**.

So most problems have no program. Undecidable problems are not rare curiosities. They are the *typical* case.

	answers for numbers 1, 2, 3, …				
problem 1	0	1	1	0	1
problem 2	1	1	0	0	0
problem 3	0	0	1	1	1
problem 4	1	0	1	0	1

flipped diagonal	1	0	0	1

The flipped diagonal differs from problem k at position k, so it is on *no* row of the list. This same trick will return in a moment.

The halting problem

HALT.
Input: a program P and an input x.
Question: does P eventually stop when run on x?

Three programs, three situations:

```
print("hi")              stops
while True: pass         loops
collatz(n)               nobody knows
```

Why it is natural. Every bug hunter wants it: "will my program freeze on this input?"

Why it is hard. To run P and see is not enough: after one hour you cannot tell *slow* from *never*.

Turing's theorem (1936). No program decides HALT for all (P, x).

The proof, step 1: suppose someone built a perfect checker



This is a **proof by contradiction**. We do not claim H exists. We assume it for a moment and show that the assumption destroys itself.

Step 2: build a mischievous program D from H

D(P):

1. Ask H: "does P halt when given *its own text* as input?"

2. If H says **halts**: loop forever.
If H says **loops**: stop immediately.

D does the opposite of what H predicts about P run on P.

D is a perfectly ordinary program: a call to H, an `if`, and a loop. If H exists, D exists.

Feeding a program its own text is legal: programs are strings (the universal machine slide).

Step 3: run D on itself



Either answer is wrong,
so **H cannot exist**.

The same proof as a table: the diagonal

run program P_i on input x_j (inputs are programs too)

	x_1	x_2	x_3	x_4	x_5
P_1	**H**	L	H	H	L
P_2	L	**L**	H	L	H
P_3	H	H	**H**	L	L
P_4	L	H	L	**H**	H
P_5	H	L	L	L	**H**
D	**L**	**H**	**L**	**L**	**L**

H = halts, L = loops

- Each row is a program, each column an input; a cell says what really happens.
- **D** is the **flipped diagonal**: it does the opposite of P_i on x_i.
- So D differs from *every* row P_i at column i. D is therefore not in the list, yet the list was supposed to contain *all* programs.

The thought experiment in code

```python
# The diagonal argument, in code.
def halts(f, x):          # claimed perfect checker
    ...                   # (any implementation)

def para(f):              # built from halts
    if halts(f, f):       # "f(f) stops"?
        while True: pass  # then loop forever
    return                # else stop at once

# Whatever halts(para, para) answers, para
# does the opposite. Try both answers:
for says in (True, False):
    really = not says     # para flips the verdict
    print(f"says halts={says!s:5}, truth="
          f"{really!s:5} ->",
          "right" if says == really else "WRONG")
```

```
says halts=True , truth=False -> WRONG
says halts=False, truth=True  -> WRONG
```

real output, Python 3.14.6

We cannot code the perfect `halts`. We *can* show that every candidate fails: whatever `halts(para, para)` answers, `para` does the opposite, so the checker is wrong. The same construction defeats *any* candidate, however sophisticated, including an AI model.

Natural objections, and why they fail

“Just wait longer.”

No finite wait is enough: a program may stop at step 10^{100}, or never.

“Use a timeout.”

Fine, but that answers “halts within T steps”, a different, decidable question.

“Use a bigger machine.”

The proof never used speed or memory. Any computer, any future, same result.

“Train an AI on millions of programs.”

It will often be right. It is still a program, so D can be built against it.

“D is a silly program.”

Silly but legal. The theorem says *every* checker fails somewhere, not that useful ones do not exist.

“Then maths is broken.”

No: it is a precise *map of limits*, proved with the same rigour as anything else.

What the theorem does *not* say

Many specific programs can be analysed. A program with no loops always stops. A program with one clear `for` over a list stops.

Specific proofs are possible. We prove that quicksort stops. We cannot do it with *one method for all programs*.

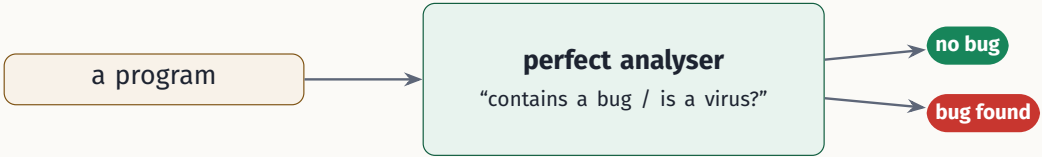
Real machines are finite. With only k bits of memory, a program has at most 2^k states, so HALT is decidable in principle, though not within the age of the universe.

The theorem says

There is no *single* algorithm that, for *all* programs and inputs, always halts with the correct yes/no.

Any tool must therefore (a) sometimes say “don’t know”, (b) sometimes be wrong, or (c) sometimes run forever.

Consequence 1: no perfect bug detector, no perfect antivirus



Hide a bug (or virus) behind “`if H says this program halts: do the bad thing`”.
A perfect analyser would then solve HALT.

- Fred Cohen proved in 1987 that perfect virus detection is undecidable. Real scanners use signatures and heuristics: some **false alarms**, some **misses**.
- Practical tools aim at one side: tests find real bugs (no false alarms, miss some); static analysers warn about possible bugs (catch more, some false alarms).

F. Cohen, “Computer viruses: theory and experiments”, Computers & Security 6(1), 1987.

Rice's theorem, in plain words

Rice (1953). Every non-trivial question about what a program *does* (its behaviour, not its text) is undecidable.

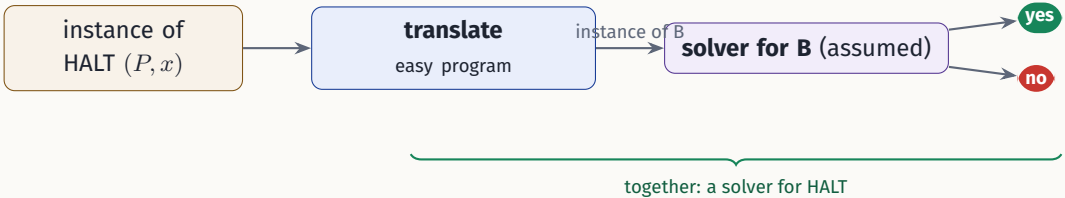
"Non-trivial": some programs have the property, some do not. Behaviour means the function the program computes.

Test. Could two programs with identical behaviour, but different text, give different answers? If yes, it is a question about *text* (decidable). If no, it is about *behaviour* (undecidable).

Text: decidable	Behaviour: not
has more than 100 lines?	ever prints "hello"?
uses the word `while`?	always returns a square number?
has a syntax error?	ever crashes?
imports `os`?	same as that other program?
	is a virus?

H. G. Rice, "Classes of recursively enumerable sets and their decision problems", Trans. Amer. Math. Soc. 74, 1953.

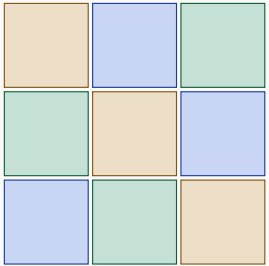
How to show a new problem is undecidable: reduce HALT to it



If B were decidable, the chain would decide HALT. HALT is not decidable, so **B is not decidable either**. Same pattern as "solving B solves hard thing A" that you will meet again for NP-completeness, but with *computability* instead of *speed*.

More undecidable problems, some surprising

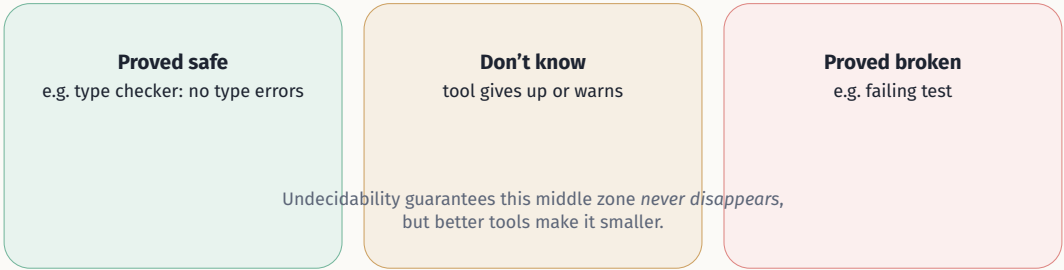
- **Hilbert's tenth problem**: does a polynomial equation with integer coefficients have an integer solution? (Matiyasevich 1970, building on Davis, Putnam, Robinson.)
- **Tiling**: can given tile types cover the infinite plane? (Berger 1966, Wang tiles.)
- **Post correspondence problem** (Post 1946): can a list of domino pairs spell the same word on top and bottom?
- **Word problem for groups** (Novikov 1955, Boone 1958).
- **Program equivalence**, "is this program ever wrong", optimal compression (*Kolmogorov complexity*).



Wang tiles: colours on edges must match.
A finite tile set, an infinite question.

Matiyasevich, Hilbert's Tenth Problem, MIT Press 1993. Berger, Memoirs AMS 66, 1966. Post, Bull. AMS 52, 1946. Novikov 1955; Boone 1958.

In practice: tools that say “yes”, “no” or “unknown”



Compilers reject programs they cannot prove safe (Rust’s borrow checker): may refuse some good programs.

Static analysis flags “possible” null pointers: some false alarms.

Formal verification proves chosen properties, with human help, for chosen programs.

Part 3 in one slide, and check yourself

1. HALT is undecidable: the diagonal argument.

2. Rice: all non-trivial behaviour questions are undecidable.

3. Tools must accept “unknown” or some mistakes.

4. But: decidable $\neq$ practical. Next question: how *fast*?

Check yourself

1. Why does the timeout version of HALT become decidable?
2. In the diagonal table, what are the entries of D in terms of the diagonal?
3. Classify: “has a loop”, “prints 7”, “never crashes”.
4. Could an AI model that reads the source decide HALT on all programs? Why not?

Part 4

Decidable, but how fast?

Big-O revisited, polynomial versus exponential, and why it matters

Solvable in principle is not solvable in practice

Guess a password made of lowercase letters, by trying every one.

The program is trivial, and it *always* finishes. Decidable. So what is the problem?

Assume 10^9 guesses per second (a fast GPU rig).

length	guesses 26^n	time
8	2.1×10^{11}	3.5 minutes
12	9.5×10^{16}	3 years
16	4.4×10^{22}	1.4 million years
20	2.0×10^{28}	6×10^{11} years

Age of the universe: about 1.4×10^{10} years. Each extra letter multiplies the work by 26. That is **exponential growth**.

Big-O refresher: read the loops

$\mathcal{O}(n)$

```
for x in data:
  total += x
```

one pass; double the data, double the time.

$\mathcal{O}(n^2)$

```
for a in data:
  for b in data:
    compare(a, b)
```

double the data, 4× the time.

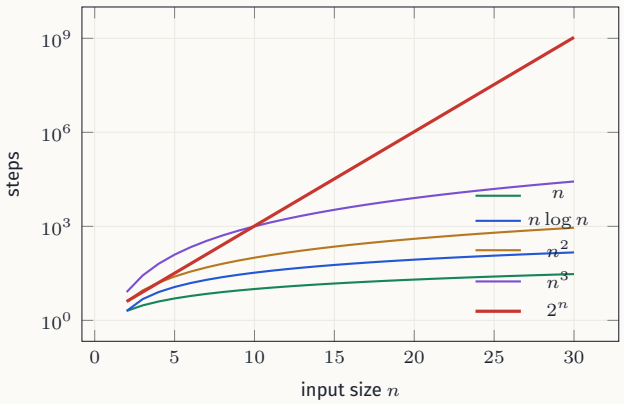
$\mathcal{O}(2^n)$

```
for subset in all_subsets:
  check(subset)
```

add *one* item, double the time.

Also on the ladder: $\mathcal{O}(\log n)$ (binary search), $\mathcal{O}(n \log n)$ (good sorting), $\mathcal{O}(n^3)$ (naive matrix multiplication), $\mathcal{O}(n!)$ (all orderings).

Growth rates, drawn (computed by code, log scale)



On a log axis a polynomial bends *down* (it is a power of n) while an exponential stays a **straight, steep line**: constant multiplier per step.

Data file `out/growth.dat` is written by `code/growth.py`.

How long would it take? (real output, 10^9 operations per second)

```
  n        n^2        n^3          2^n             n!
 10     1e-07 s    1e-06 s      1e-06 s        0.004 s
 20     4e-07 s    8e-06 s      0.001 s         77.2 yr
 30     9e-07 s    3e-05 s       1.07 s    8.42e+15 yr
 50     3e-06 s  0.0001 s         13 day    9.66e+47 yr
 70     5e-06 s  0.0003 s  3.75e+04 yr     3.8e+83 yr
(1e9 operations per second)
```

real output, Python 3.14.6

```python
# Growth table + data for the plots.
import math
rate = 1e9   # operations per second

def fmt(s):
    for u, k in (("yr", 3.15e7), ("day", 86400),
                 ("h", 3600), ("s", 1)):
        if s >= k: return f"{s/k:.3g} {u}"
    return f"{s:.1g} s"
```

Polynomial: 70 items is no problem, even n^3 takes a third of a millisecond.

Exponential: at $n = 50$, thirteen *days*; at $n = 70$, 37,500 *years*.

Factorial is worse: 20 items already take 77 years. This is the cost of "try every ordering".

Does a faster computer rescue us?

Question. Today you can solve size n in one hour. You buy a computer *1000 times faster*. What size can you solve in one hour?

cost	new size	change
n	$1000\,n$	thousand times bigger
n^2	$31.6\,n$	32 times bigger
n^3	$10\,n$	10 times bigger
2^n	$n + 10$	**ten more items**

$2^{n'} = 1000 \cdot 2^n \Rightarrow n' = n + \log_2 1000 \approx n + 10$.

Moore's law cannot fix exponential cost. Hardware gains are multipliers; exponential growth eats multipliers for breakfast.

Better *algorithms* change the curve itself. That is where the big wins in computing have come from.

The class P: efficiently solvable

P = decision problems solvable in time $\mathcal{O}(n^k)$ for some fixed k (*polynomial time*), on a Turing machine or any reasonable computer.

Why polynomials? They are **closed** under composition: using a polynomial routine polynomially many times is still polynomial. And the class does not depend on the machine model (for ordinary computers).

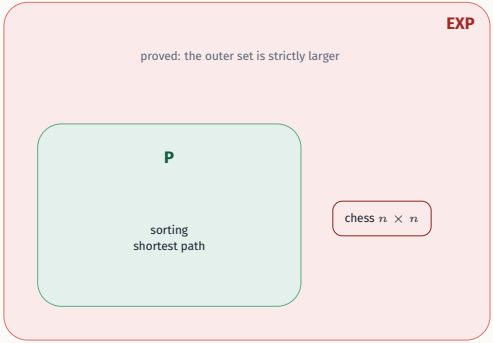
Problem	in P because
Sorting	$\mathcal{O}(n \log n)$
Shortest path	Dijkstra
Is n prime?	AKS (2002)
Matching	Edmonds 1965
Linear programming	Khachiyan 1979
2-SAT, 2-colouring	linear time

EXP: provably needs exponential time

EXP = problems solvable in time 2^{n^k}.

Generalised games. Who wins from a given position on an $n \times n$ chess board? EXP-complete (Fraenkel and Lichtenstein 1981). Same for generalised Go with Japanese ko rules (Robson 1983).

Time hierarchy theorem (Hartmanis and Stearns 1965): more time lets you solve strictly more problems. Hence **P** $\subsetneq$ **EXP**: some problems have *no* fast algorithm. This is proved.



Fraenkel and Lichtenstein, J. Combin. Theory A 31, 1981. Robson, Proc. ICALP 1983. Hartmanis and Stearns, Trans. AMS 117, 1965.

Look-alike problems: one easy, one (apparently) hard

Easy (in P)	Seems hard (NP-complete)	What changed
Shortest path between two cities	Longest simple path	a “min” became a “max”
Visit every *street* once (Euler tour)	Visit every *junction* once (Hamilton tour)	edges become vertices
2-colour a map	3-colour a map	2 becomes 3
2-SAT: clauses with 2 literals	3-SAT: clauses with 3 literals	2 becomes 3
Split items into two piles of equal count	Split items into two piles of equal *weight* (partition)	count becomes weight
Minimum spanning tree	Travelling salesman	“connect” becomes “tour”

Tiny changes in a problem can move it across the line. No one has a simple rule that tells which side a new problem is on.

Demo: brute force, timed (real measurements)

```
def subset_sum(nums, target):
    for r in range(len(nums) + 1):
        for c in itertools.combinations(nums, r):
            if sum(c) == target: return c
```

```
        t = [0, *p, 0]
        best = min(best, sum(d[a][b]
                   for a, b in zip(t, t[1:])))
    return best

def clock(f, *a):
```

n	subset-sum	SAT	cities	TSP
8	0.0000 s	0.0001 s	5	0.0000 s
10	0.0001 s	0.0004 s	6	0.0001 s
12	0.0003 s	0.0018 s	7	0.0004 s
14	0.0012 s	0.0078 s	8	0.0035 s
16	0.0052 s	0.0283 s	9	0.0288 s
18	0.0231 s	0.1172 s	10	0.2953 s
20	0.0976 s	0.4642 s		
22	0.4420 s			

real output, Python 3.14.6

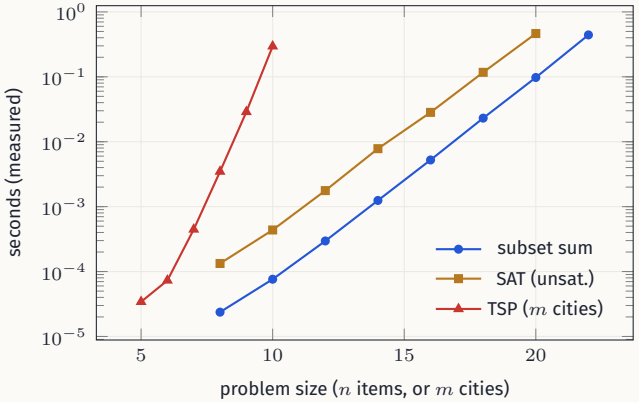
Subset sum: do some of these numbers add up to a target? Try all subsets: 2^n.

SAT: is there a true/false assignment making a formula true? Here a formula that is *unsatisfiable*, so every one of the 2^n assignments must be tried.

TSP: shortest tour through m cities by trying all $(m-1)!$ orders.

Look at the columns: every $+2$ items multiplies the time by about 4. That is 2^n in action.

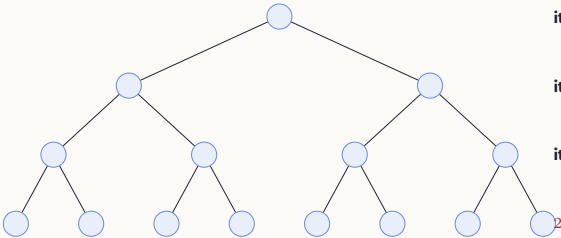
The same data as a plot (log scale)



Near-straight lines on a log axis: **exponential** (or factorial) growth. TSP climbs the steepest: from 9 to 10 cities the time grows tenfold.

Extrapolate: SAT at $n = 40$ would take about $2^{20} \times 0.45$ s ≈ 5 days in this naive Python; at $n = 60$, about 16,000 years.

Where does 2^n come from?



item 1: in or out?

item 2: in or out?

item 3: in or out?

$2 \times 2 \times 2 = 8$ complete choices

Each yes/no decision doubles the number of candidate answers. With n decisions: 2^n. Brute force walks every leaf. The hope behind clever algorithms is to **prune** whole branches without visiting them.

Part 5

P versus NP

Easy to check, hard to find? The million-dollar question

Finding is hard. Checking is easy.

A hard Sudoku may take you an hour. **A finished one** takes a minute to check.

Jigsaw, maze, crossword: same story. Solving needs search; verifying needs a scan.

Big question: is verifying really easier than finding, or are we just not clever enough?

```
check a given answer: True (1 microseconds)
search all 2,704,156 subsets: 1.1 s
```

real output, Python 3.14.6

```python
# Finding is hard, checking is easy.
import random, time, itertools
random.seed(3)
N = 24
nums = [random.randrange(10**6, 10**7)
        for _ in range(N)]
ans = random.sample(range(N), N // 2)
target = sum(nums[i] for i in ans)

def check(sub):              # polynomial time
    return sum(nums[i] for i in sub) == target

t = time.perf_counter(); ok = check(ans)
us = (time.perf_counter() - t) * 1e6
print("check a given answer:", ok,
      f"({us:.0f} microseconds)")

target += 1                  # make it unsolvable
t = time.perf_counter()
for c in itertools.combinations(range(N), 12):
    check(c)
print("search all 2,704,156 subsets:",
      f"{time.perf_counter() - t:.1f} s")
```

NP: problems whose answers are quick to check

NP = decision problems where a **yes** answer has a short *certificate* (a proof) that can be checked in polynomial time.

Certificate: the solved grid, the tour, the variable assignment.

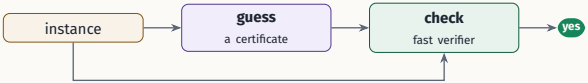
Verifier: an ordinary polynomial-time program.

Note: only the *yes* side needs a short proof. “No” answers can be harder to certify.

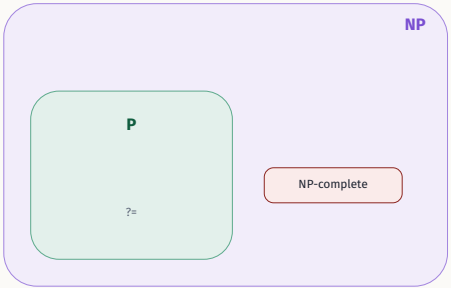
Common mistake

NP does **not** mean “not polynomial”. It means *nondeterministic polynomial*: a machine that could *guess* the certificate, then check it quickly.

Guess and check, and where P sits



P ⊆ NP: if you can *solve* quickly, you can check quickly (just solve again). The open question is whether the reverse holds.



if P ≠ NP (the usual belief)

Many everyday puzzles are in NP

Problem	Short certificate	Check
Sudoku (any size)	the filled grid	rows, columns, boxes
Subset sum	the chosen numbers	add them up
Travelling salesman (“tour $\leq L$?”)	the tour	add the edge lengths
Graph colouring with k colours	a colour for each node	compare neighbours
SAT	a true/false value per variable	evaluate every clause
Timetabling	the timetable	scan for clashes
Factoring (“factor below k?”)	the factor	divide
Protein folding (simple models)	a shape	compute its energy

Each certificate is *short* (polynomial size) and each check is *fast*. Whether each problem has a fast *solver* is exactly the question.

SAT: the most famous problem in the story

SAT. Given a Boolean formula in the form "AND of ORs", is there a way to assign *true*/*false* to the variables that makes the whole formula true?

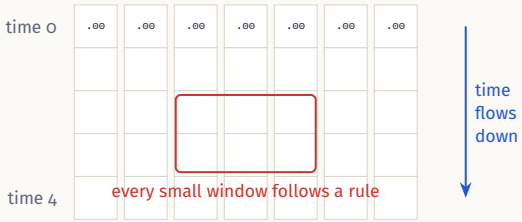
Example.

$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$

Each bracket is a *clause*. Satisfiable: 2 of the 8 assignments work (right). With n variables there are 2^n assignments to try.

x	y	z	formula
F	F	F	**F**
F	F	T	**F**
F	T	F	**T**
F	T	T	**F**
T	F	F	**F**
T	F	T	**T**
T	T	F	**F**
T	T	T	**F**

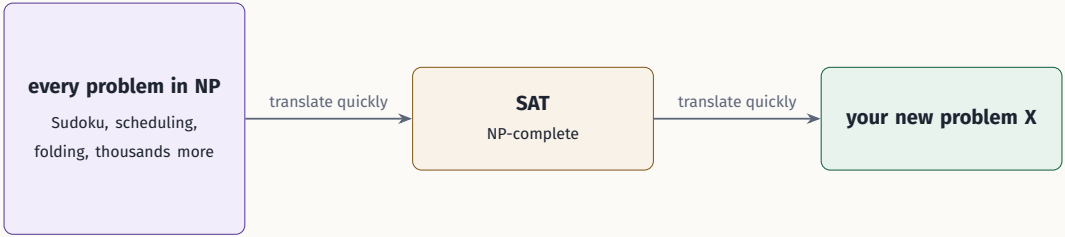
Why SAT is special: any computation is a formula (Cook–Levin)



- Record a Turing machine's run as a grid: row = time, column = tape cell.
- "Row 0 holds the input, each row follows from the previous by the rules, the last row says accept" is a **giant AND of tiny local conditions**: a SAT formula, polynomial in size.
- **Cook 1971, Levin 1973**: every NP problem turns into SAT this way. SAT is **NP-complete**: at least as hard as everything in NP.

S. Cook, "The complexity of theorem-proving procedures", STOC 1971. L. Levin, Problems of Information Transmission 9, 1973.

NP-complete: the hardest problems in NP



So a fast algorithm for X would give a fast algorithm for SAT,
hence for *everything* in NP.

A **reduction** turns any instance of problem A into an instance of problem B with the *same answer*, in polynomial time. If B is easy, A is easy. Contrapositive: if A is hard, B is hard.

Reduction by example 1: Sudoku as a SAT problem

1	–	–	–
–	–	3	–
–	4	–	–
–	–	–	2

a 4×4 mini Sudoku

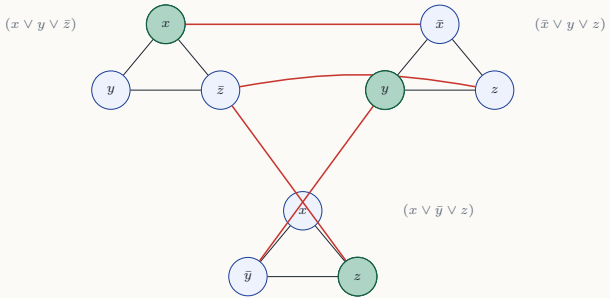
Variables. $v_{r,c,d}$ = "cell (r, c) holds digit d". A 9×9 grid has $9 \cdot 9 \cdot 9 = 729$ of them.

Clauses. Each cell has *some* digit: $(v_{r,c,1} \vee \cdots \vee v_{r,c,9})$. Never two in one cell: $(\neg v_{r,c,d} \vee \neg v_{r,c,d'})$.

Rules. Each digit appears in each row, column and box. **Givens**: one-literal clauses $(v_{1,1,1})$.

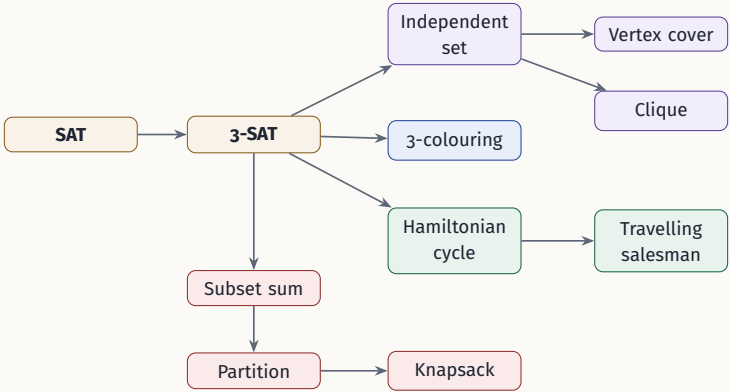
Satisfying assignment = solved Sudoku. A SAT solver solves the puzzle without knowing any Sudoku tricks.

Reduction by example 2: 3-SAT to a graph problem



- One **triangle per clause**; its three nodes are the literals.
- **Red edges** join contradictory literals (x and $\bar{x}$).
- Pick *one node per triangle* with no red edge between picks (an **independent set** of size 3).
- Green picks say: x, y, z all true. It satisfies every clause. Formula satisfiable $\iff$ such a set exists.

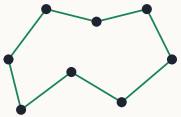
Karp (1972): a web of reductions



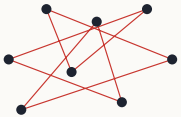
Richard Karp showed in 1972 that 21 natural problems are all NP-complete, each via a reduction from an earlier one. Today thousands are known: scheduling, packing, circuit layout, cutting stock, … A fast algorithm for *any one* of them solves *all*.

R. Karp, “Reducibility among combinatorial problems”, in Complexity of Computer Computations, 1972. Garey and Johnson, Computers and Intractability, 1979.

Travelling salesman: how many tours?



tour of length 18.4 (optimal)



a random tour: 36.9

cities	tours $(n-1)!/2$
8	2,520
10	181,440
15	4.4×10^{10}
20	6.1×10^{16}
25	3.1×10^{23}

Lengths computed by code. For 8 cities, 2,520 tours are easy to enumerate. 25 cities: more than the number of seconds in 10^{15} years at a billion tours per second.

Worst-case hard is not the same as hard in practice

TSP: the Concorde solver has found *provably optimal* tours for instances with tens of thousands of cities (85,900 for a chip-design instance).

Sudoku: 9×9 is trivial for a backtracking program in milliseconds. The $n^2 \times n^2$ generalisation is NP-complete (Yato and Seta 2003).

Lesson: NP-complete means “no known algorithm is fast on *all* inputs”, not “every instance is hopeless”.

Real instances have structure: geography, symmetry, many constraints that prune the search quickly.

Adversarial instances exist for every algorithm, and that is what the worst-case theory is about.

Applegate, Bixby, Chvátal, Cook, The Traveling Salesman Problem: A Computational Study, Princeton UP 2006; Concorde: math.uwaterloo.ca/tsp. Yato and Seta, IEICE Trans. Fund. E86-A(5), 2003.

What if P = NP?

Optimisation becomes routine

best schedules, routes, chip layouts, protein designs, found fast.

Most cryptography breaks

finding a key from a ciphertext would be as easy as checking one.

Search for proofs gets easy

if a proof of length n exists, find it in polynomial time. Gödel asked this in a 1956 letter to von Neumann.

Learning becomes easy

finding the simplest model that fits the data would be fast.

Creativity is cheaper

the gap between *appreciating* a solution and *finding* it would vanish, a point Aaronson often stresses.

But

an algorithm may have n^{100} steps. Proving P=NP is not the same as getting a useful algorithm.

Aaronson, "P $\stackrel{?}{=}$ NP", in Open Problems in Mathematics, Springer 2016. Gödel's letter to von Neumann, 1956.

Why most experts believe P $\neq$ NP

Decades of failure. Thousands of NP-complete problems, attacked by the best minds for 50 years, no polynomial algorithm for any.

Asymmetry of experience. Appreciating a solution is much easier than producing it. P = NP would say that gap is an illusion.

Consequences look too good. Automatic proofs, perfect scheduling, no secure encryption.

2019 poll of complexity theorists



Opinion is not proof. Experts have been surprised before (e.g. primality in P, 2002; the claimed ETH refutation, Part 7).

W. Gasarch, “The Third P =? NP Poll”, SIGACT News 50(1), 2019 (reported figures: 88% P $\neq$ NP).

Why is it so hard to prove? Known barriers

Relativization

Baker, Gill, Solovay (1975): there are imaginary "oracle worlds" where P=NP and others where P≠NP. So any proof that works equally in all worlds fails.

Natural proofs

Razborov and Rudich (1994): a broad family of circuit-lower-bound techniques cannot work, if strong pseudo-random generators exist.

Algebrization

Aaronson and Wigderson (2008): a further family of algebraic techniques is also blocked.

Any real proof needs a genuinely new idea. This is one reason a claim of "P $\neq$ NP proved" appearing on the internet is almost always wrong (see the claims-checking slide in Part 7).

A million dollars

Clay Mathematics Institute, 2000: seven Millennium Prize Problems, $1,000,000 each. **P versus NP** is one. The problem is posed as: if it is easy to check that a solution is correct, is it also easy to solve?

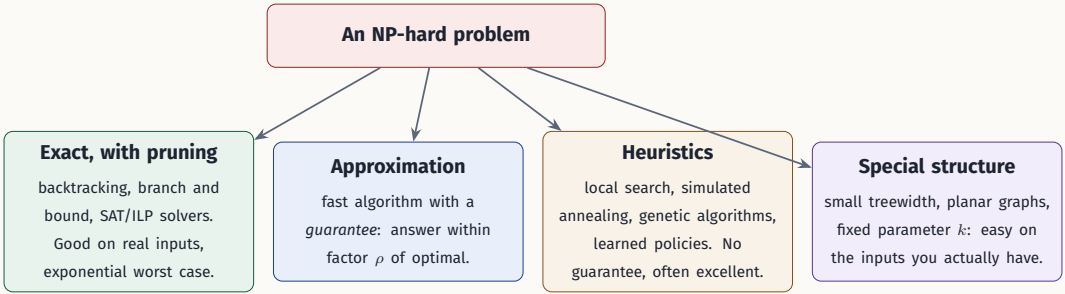
Rules: publication in a qualifying journal, then two years of general acceptance by the mathematical community, before the prize committee considers it.

Status: only one of the seven (the Poincaré conjecture, by Perelman) has been awarded, and he declined the prize.

- P versus NP
- Riemann hypothesis
- Navier–Stokes
- Yang–Mills
- Hodge
- Birch–Swinnerton-Dyer
- Poincaré (solved)

claymath.org/millennium-problems and /millennium/p-vs-np (problem statement). The Riemann hypothesis claim of Nov 2026 is discussed in Part 7.

Intractable in practice: the working engineer's toolbox



Knowing a problem is NP-hard does *not* mean give up. It means: stop looking for a perfect fast algorithm, and choose which **compromise** you accept.

Approximation: a guarantee instead of perfection

pick an uncovered edge, take both ends: 6 nodes



optimal cover: 3 nodes



The greedy rule is never worse than twice optimal: each picked edge needs at least one of its ends in *any* cover. Here it is exactly $2\times$.

Metric TSP: Christofides (1976) gives tours at most $1.5\times$ optimal. In 2021, Karlin, Klein and Oveis Gharan improved the factor by a minuscule amount after 45 years.

Some problems resist approximation too: under standard assumptions, clique cannot be approximated well in polynomial time.

NP-hard problems differ a lot in *how* hard they are to approximate.

Christofides, CMU Tech. Report 388, 1976. Karlin, Klein, Oveis Gharan, “A (slightly) improved approximation algorithm for metric TSP”, STOC 2021.

SAT solvers: a tiny one, with real output

```python
def sat_by(a, l):
    return a.get(abs(l)) == (l > 0)

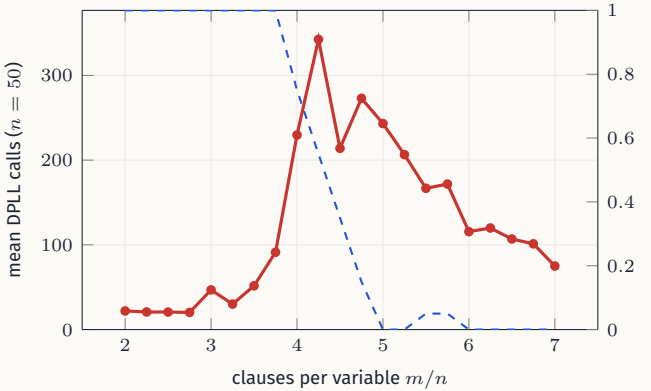
def dpll(clauses, a, st):
    st["calls"] += 1
    a = dict(a)
    changed = True
    while changed:                      # unit rule
        changed = False
        for cl in clauses:
            if any(sat_by(a, l) for l in cl):
                continue
            free = [l for l in cl if abs(l) not in a]
            if not free: return None  # conflict
            if len(free) == 1:
                a[abs(free[0])] = free[0] > 0
                changed = True
    left = [c for c in clauses
            if not any(sat_by(a, l) for l in c)]
    if not left: return a
    v = next(abs(l) for l in left[0]
             if abs(l) not in a)
    for val in (True, False):           # branch
        r = dpll(clauses, {**a, v: val}, st)
        if r: return r
    return None
```

```
tiny example: {1: True, 3: True, 2: False}
n   m/n  sat?  DPLL calls   2^n
20  3.0  yes           6   1e+06
20  4.3  yes          19   1e+06
20  6.0  no           27   1e+06
40  3.0  yes          15   1e+12
40  4.3  yes         129   1e+12
40  6.0  no           47   1e+12
60  3.0  yes          17   1e+18
60  4.3  yes         198   1e+18
60  6.0  no          283   1e+18
```

real output, Python 3.14.6

Idea (DPLL): choose a variable, try true, then false; apply the *unit rule* (a clause with one free literal forces it); abandon a branch at the first conflict. 40 variables: $2^{40} \approx 10^{12}$ assignments, but about 100 calls.

Random 3-SAT has a phase transition near 4.27 clauses per variable



Few clauses: almost always satisfiable and easy (many solutions). Many clauses: almost always unsatisfiable, and a conflict appears fast. In between, near $m/n \approx 4.27$, the answer flips and the search is **hardest**.

Data: 20 random formulas per ratio, `code/satphase.py`. Threshold: Mitchell, Selman and Levesque, AAAI 1992 (empirical).

Part 5 in one slide, and check yourself

1. NP = quick to check. P = quick to solve. $P \subseteq NP$.

2. SAT is NP-complete; reductions spread hardness to thousands of problems.

3. P = NP is open ($1M). Most experts bet $P \neq NP$.

4. In practice: solvers, heuristics and approximation work wonders on real inputs.

Check yourself

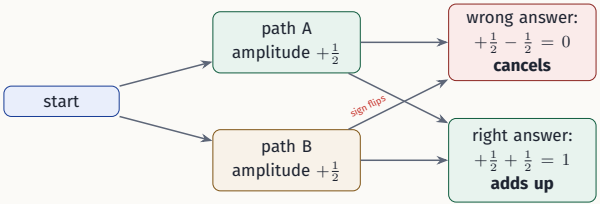
1. Is a problem in P also in NP? Why?
2. Give a certificate for 3-colouring a graph, and say how to check it.
3. Does “NP-hard” mean “impossible in practice”? Explain.
4. If someone finds a polynomial algorithm for TSP, what else follows?

Part 6

Beyond: quantum, randomness, cryptography

What new kinds of machines change, and what they do not

Quantum computing: interference, not “all answers at once”



A qubit holds *amplitudes* (positive or negative) for 0 and 1. An algorithm arranges the paths so wrong answers cancel and the right one adds up.

Myth: “it tries every answer at once and picks the best.” Measuring gives *one* random outcome.

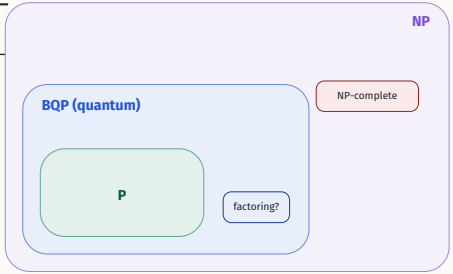
Reality: only problems with special structure get a speed-up.

And: it computes nothing a Turing machine cannot. Computability is untouched; only speed may change.

What quantum computers change, and what they do not

Algorithm	Task	Speed-up
Shor 1994	factoring, discrete log	exponential (to polynomial)
Grover 1996	unstructured search of N items	square root: $N \to \sqrt{N}$
BBBV 1997	brute-force search for NP-complete problems	no better than square root, in the black-box model

Grover on SAT: $2^n \to 2^{n/2}$. Still **exponential**.

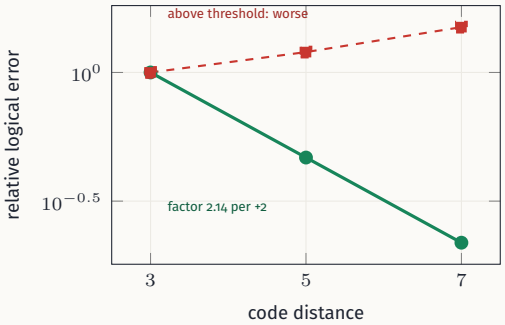


Widely believed: NP-complete problems lie outside BQP. Factoring is in BQP but is *not* known to be NP-complete. Unproven.

Shor, SIAM J. Comput. 26, 1997 (FOCS 1994). Grover, STOC 1996. Bennett, Bernstein, Brassard, Vazirani, SIAM J. Comput. 26, 1997.

Where quantum hardware stands (reported milestones)

illustrative “below threshold”: bigger code, fewer errors



Green: ratios from the reported factor 2.14 (arXiv:2408.13687).

Red: hypothetical contrast.

Error correction works in principle. Google’s surface-code memories: a larger code gave a *lower* logical error rate (factor about 2.14 per step in distance). Nature 2025.

Still early. Breaking real RSA needs thousands of *logical* qubits, each built from many noisy physical ones.

Standards moved already. NIST published post-quantum encryption and signature standards in Aug 2024 (FIPS 203, 204, 205): “harvest now, decrypt later”.

Acharya et al., “Quantum error correction below the surface code threshold”, arXiv:2408.13687 (Nature 638, 2025). NIST, FIPS 203/204/205, 13 Aug 2024. We make no claims about hardware timelines.

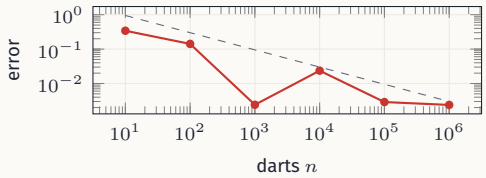
Randomness: flipping coins can help

```python
# Coin flips as a tool: estimate pi by darts.
import random
random.seed(5)
print("darts      estimate   error")
with open("../out/mc.dat", "w") as f:
    f.write("n err\n")
    for k in range(1, 7):
        n = 10 ** k
        hit = sum(random.random() ** 2 +
                  random.random() ** 2 < 1
                  for _ in range(n))
        est = 4 * hit / n
        err = abs(est - 3.141592653589793)
        f.write(f"{n} {err:.6g}\n")
        print(f"{n:<9d} {est:.5f}   {err:.5f}")
```

```
darts      estimate   error
10         2.80000   0.34159
100        3.00000   0.14159
1000       3.14400   0.00241
10000      3.11800   0.02359
100000     3.14448   0.00289
1000000    3.14398   0.00238
```

real output, Python 3.14.6

error shrinks about like $1/\sqrt{n}$ (dashed)



Primality: the Miller–Rabin test answers quickly, with error below 4^{-k} after k rounds (deterministic AKS: 2002).

Belief: P = BPP (Impagliazzo and Wigderson 1997, under a hardness assumption). Randomness is not believed to crack NP.

Miller, JCSS 13, 1976; Rabin, J. Number Theory 12, 1980; Agrawal, Kayal, Saxena, Annals of Math. 160, 2004; Impagliazzo and Wigderson, STOC 1997.

Cryptography is built on hard problems

One-way function: easy to compute, hard to invert. Multiply two primes: easy. Factor the product: no known fast method.

RSA (1977) relies on factoring. Your browser's HTTPS lock relies on such assumptions.

Caution: factoring is not known to be NP-complete, and crypto needs hardness on *typical* instances, a stronger demand than worst-case.

```
 7 digits  multiply   0.5 us   factor    0.000 s
11 digits  multiply   0.5 us   factor    0.005 s
15 digits  multiply   0.2 us   factor    0.503 s
17 digits  multiply   0.4 us   factor    5.027 s
```

real output, Python 3.14.6

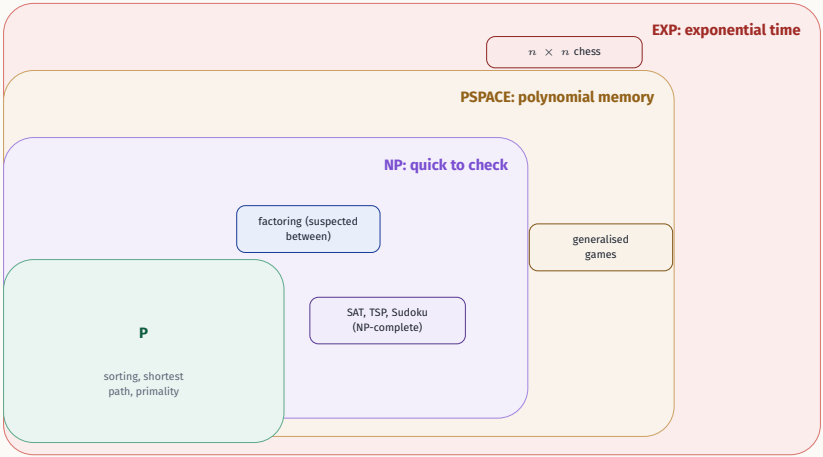
```python
# Multiplying is easy. Factoring is not (yet).
import time

def factor(n):                          # trial division
    d = 2
    while d * d <= n:
        if n % d == 0: return d
        d += 1
    return n
```

Trial division: going from 15 to 17 digits multiplies the time by about 9. Real RSA keys have 600+ digits and attackers use far better algorithms, yet the gap stays astronomical.

Rivest, Shamir, Adleman, Commun. ACM 21(2), 1978.

The complexity zoo, as a map



Known: P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP and **P** $\neq$ **EXP**. Unknown: which of the inclusions in between are strict (most believed). Outside the whole picture: the undecidable.

Part 7

What is happening now

Two big claims from October and November 2026, read with care

First, how to read a headline in science



Each step removes a different kind of error: *typos and false proofs* (3, 4), *wrong problem statements* (4), *hype* (5, 6). Skipping steps is how wrong claims spread.

In this part, each claim is placed on this ladder using sources shown on the slide. We state **only what the sources support** and mark the rest as **claim**.

What does “ETH” mean here?

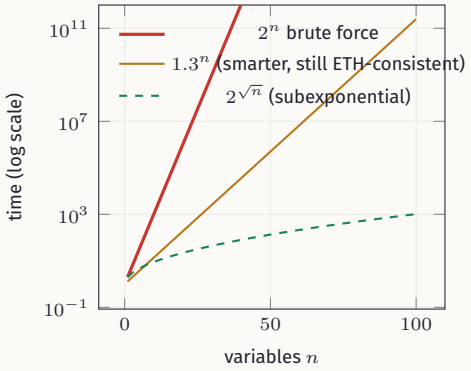
Exponential Time Hypothesis (ETH), Impagliazzo and Paturi 1999: some constant $s > 0$ makes 3-SAT on n variables need time 2^{sn}. No *subexponential* $2^{o(n)}$ algorithm.

Strong ETH: k-SAT needs time close to 2^n as k grows.

Why it matters: hundreds of papers prove “no algorithm faster than X unless ETH fails”. ETH $\Rightarrow$ P $\neq$ NP, not the reverse.

Not ETH Zurich (the university) or ETH (Ethereum): here ETH is the hypothesis.

illustrative shapes of running times



ETH says 3-SAT can never drop to the green dashed shape.

Source: en.wikipedia.org/wiki/Exponential_time_hypothesis (accessed 23 Nov 2026); Impagliazzo and Paturi, J. Comput. Syst. Sci. 62(2), 2001.

October 2026: a claimed refutation of ETH

OpenAI (26 Oct 2026): “Surprisingly, our system has recently proved that the ETH is false, a result that was subsequently verified in Lean.” Found 16 Oct, formalised in Lean 17 Oct.

Release: external “shepherds” led by Impagliazzo, Paturi and Williams; OpenAI will publish the AI outputs, and the group asked that all raw data be public.

Also claimed: Anthropic announced that one of its models independently disproved ETH, with details on GitHub.

Status, 23 Nov 2026

CLAIMED, under review. Not peer reviewed.



steps 1–3 done by the lab; step 4 under way

We did not read the 67-page PDF ourselves; we rely on OpenAI’s page and the expert commentary on the next slide.

OpenAI, “Working with academic communities to build scientific understanding”, 26 Oct 2026: openai.com/index/building-scientific-understanding-with-academic-communities. Preprint: github.com/openai/math (preprints/The-Exponential-Time-Hypothesis-is-False-October-25-2026). Anthropic claim: Wikipedia (ETH) and Fortnow, next slide.

What independent voices say

Lance Fortnow (28 Oct): has *not* verified the proof himself, but writes "it is safe to assume now that the exponential-time hypothesis is toast". Notes it concerns k-SAT, not all of NP.

Dias and Lima (arXiv, 31 Oct): the claimed bounds beat brute force only at astronomically large n: no practical speed-up.

MIT Hardness Group (arXiv, 2 Nov): most ETH-based lower bounds survive with slightly adjusted exponents ("ETH is dead, long live SH").

Reviewers' caution (reported): their involvement "should not be taken as certification of the claim".

Sources disagree on details: the exact running-time bound is reported differently, and the Lean file proves a slightly weaker bound than the paper. We quote no formula.

Fortnow, blog.computationalcomplexity.org/2026/10/the-exponential-time-refutation.html (28 Oct 2026). Dias and Lima, arXiv:2611.00517. MIT Hardness Group et al., arXiv:2611.02650 (v2, 16 Nov). Reviewers' statement: Dataconomy, 4 Nov 2026 (secondary).

What it would, and would not, change

If it stands

3-SAT and k-SAT have subexponential algorithms. Textbooks and papers that say “assuming ETH” need new wording.

It does not mean

P = NP; broken encryption; solved TSP or Subset Sum (reductions to 3-SAT blow up the number of variables); fast SAT solvers tomorrow.

Scientific lesson

a “widely believed” hypothesis can fall. A hypothesis is a bet, not a theorem. Complexity theory keeps changing.

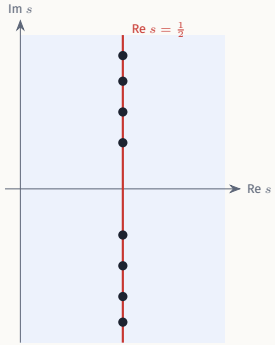
Compare the ladder: we now have a claim with machine-checked formalisation and serious expert engagement. We *do not yet* have independent, completed community verification or peer review.

Scope statements: Fortnow (28 Oct 2026); Dias and Lima (arXiv:2611.00517); MIT Hardness Group (arXiv:2611.02650).

The Riemann hypothesis: what is it?

Primes (2, 3, 5, 7, …) look random, but their count up to x follows a smooth law. How tight is that law? Riemann (1859) tied the error to the zeros of the *zeta function* $\zeta(s)$.

Hypothesis: every non-trivial zero has real part exactly $\frac{1}{2}$. Checked by computer for trillions of zeros, never proved. It is one of the seven Millennium problems.



zeros (dots) conjectured to lie on the red line

Riemann, 1859; Clay Mathematics Institute, Millennium Prize Problems (claymath.org). Illustration schematic, not to scale; dot positions are not actual zeros.

November 2026: a claimed proof of the Riemann hypothesis

Claim (20 Nov 2026). Levent Alpöge of Anthropic posted a four-page note: “I believe I have, working with Claude, discovered a proof of the Riemann hypothesis.” Only a Lean version was released; no full human-readable argument yet.

Reaction (AFP). Brian Conrey, a leading specialist, called it “a monumental moment in human history” but had seen no manuscript.

Lean communities (21 Nov, as reported). The Lean FRO and Mathlib Initiative say the library matches Mathlib’s definition of the hypothesis and state “full confidence” it is correct.

Status, 23 Nov 2026

CLAIMED; machine-checked per the Lean community; not human-vetted or accepted.

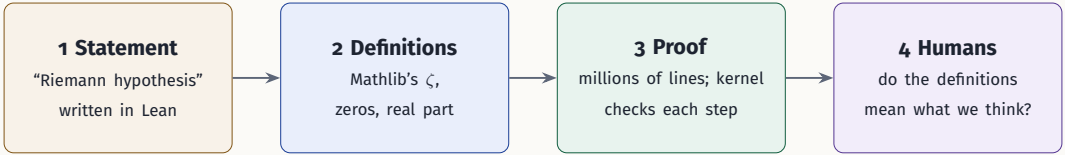


steps 1–3 reported; steps 4–6 pending

Clay prize rules need journal publication plus two years.

AFP via Phys.org, 20 Nov 2026 (phys.org/news/2026-11-anthropic-claude-math-greatest-riddle.html). Lean FRO and Mathlib Initiative statement, 21 Nov 2026, as reported by OfficeChai (secondary); we could not open the primary post (Lean FRO on X, status/2124159121645801562).

What a Lean certificate does, and does not, guarantee



Machine-checked: the proof (3) follows from the stated axioms and definitions. That removes the main risk of a *wrong proof*, which is a real risk for arguments said to run to 250,000 pages if written out (as reported).

Not guaranteed: that (1) and (2) say what we intend. Statement-matching must be checked by people and tools (the Lean statement describes such checks). Also not given: *understanding*.

Claims to check: where each claim stands on 23 Nov 2026

	ETH refuted (OpenAI, Anthropic)	Riemann hypothesis proved
Who and when	OpenAI 26 Oct; an Anthropic claim followed (reported)	Alpöge (Anthropic) with Claude, 20 Nov
Primary source seen	OpenAI's announcement page (yes); the preprint PDF (not read)	AFP report quoting the note (yes); Lean statement (secondary only)
Machine check	by the labs, Lean (a weaker bound than the headline)	Lean community statement, 21 Nov (reported)
Expert review	under way; reviewers say "not certification"	just starting; experts have not seen a human-readable proof
Peer review	none yet	none; Clay rules need 2 years
Our label	**CLAIMED, serious, unconfirmed**	**CLAIMED, formally checked, unconfirmed**

We therefore **do not** say "ETH is false" or "the Riemann hypothesis is solved" as facts. We say what was *claimed*, by whom, and what checks exist.

A checklist for evaluating a claim like this

1 Primary source

Is there a paper or announcement you can open, not just a news story? Who wrote it? Be careful with copies and AI-generated summaries.

2 Formal check

Is there a machine-checked proof? Does the checked statement match the claim? Who confirmed that?

3 Independent replication

Have people with no stake rebuilt or re-run it? Do named specialists engage, with names and reasoning?

4 Peer review and time

Is it in a journal? Prize rules, such as the Clay Institute's two years, exist because checking takes time.

5 Incentives

Who gains from the headline? AFP noted the Riemann announcement came a day after Anthropic's record stock market debut.

6 Consistency

Do sources agree on basic facts? Here they differ on bounds and dates. Mismatches are a warning sign.

Timing remark: AFP via Phys.org, 20 Nov 2026. The Clay rule is stated in the same report and on claymath.org.

Timeline of the claims (from the sources on the previous slides)



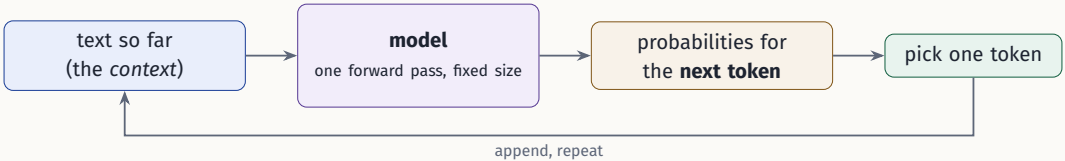
Honest summary on 23 Nov 2026: **two major claims, both by AI labs, both with formal components, neither yet confirmed by the community**. Re-check the primary sources before citing this slide.

Part 8

The limits of large language models

What complexity and computability say about today's AI, and what the evidence shows

What a language model does, as a loop



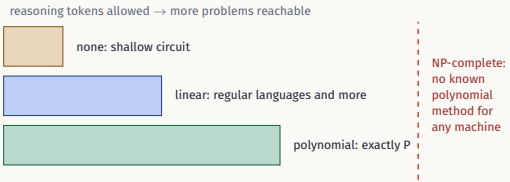
Strength: trained on huge text and code corpora, fluent, broad, good at pattern completion and at writing programs.

Structure to remember: each token costs the *same* amount of computation, and nothing in the loop checks whether the output is true.

Fixed computation per token, and what “thinking” tokens add

No intermediate steps: the answer comes after a fixed number of layers. A *shallow* computation, however wide.

Chain of thought: each written token is extra computation. Merrill and Sabharwal (2023): more steps, more power; logarithmic steps add little; *polynomially many* steps give, in their model, exactly P.



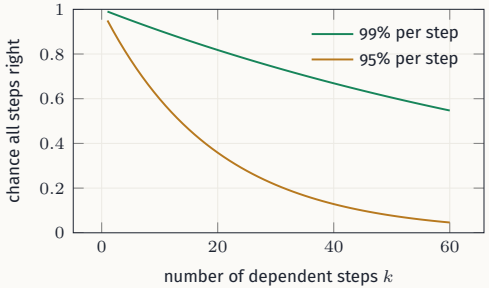
Intuition only: more reasoning tokens means more computation, but still polynomial per answer.

Merrill and Sabharwal, “The expressive power of transformers with chain of thought”, arXiv:2310.07923 (2023, rev. 2024). Idealised theoretical model; not a measurement of any product.

Compositional depth: errors multiply

Faith and Fate (Dziri et al., 2023) studied multi-digit multiplication, logic-grid puzzles and a dynamic-programming problem. Accuracy fell as the size of the underlying computation graph grew.

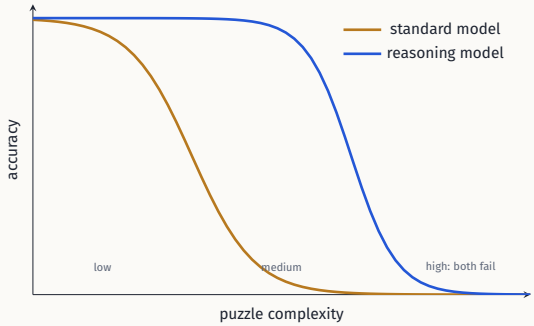
Their reading: the models often reduce multi-step reasoning to matching patterns seen in training, rather than learning a systematic procedure, and errors in early steps propagate.



Simple arithmetic (p^k), *illustrative*: assumes independent errors. Long chains need very high per-step reliability, or checks along the way.

Dziri et al., "Faith and Fate: Limits of Transformers on Compositionality", arXiv:2305.18654 (NeurIPS 2023). Plot is a textbook calculation, not data from the paper.

Reasoning models: strong evidence, and a live debate



Schematic of the three regimes reported by Shojaee et al.

(*illustrative*: curves drawn by hand, not their data).

"The Illusion of Thinking" (Apple, 2025): on puzzles with adjustable difficulty, *complete accuracy collapse* beyond a threshold; reasoning effort even declined near it.

Published reply (Lawsen, 2025): many failures reflect output-token limits and impossible puzzle instances. Asked for a short program instead of a move list, models did well in preliminary tests.

Lesson: ask for a *program* and run it. Computation moves from tokens to an exact tool.

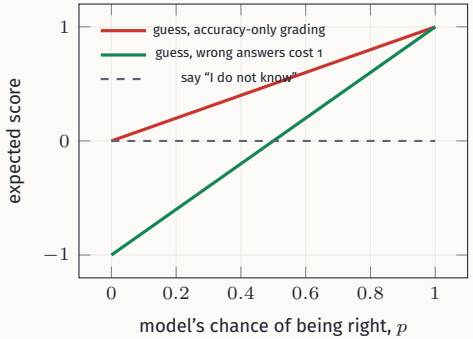
Shojaee et al., arXiv:2506.06941 (NeurIPS 2025). A. Lawsen, arXiv:2506.09250 (2025). Preprint-level debate; later work may change the picture.

Hallucination: plausible, confident, wrong

Kalai, Nachum, Vempala, Zhang (2025): models hallucinate because training and common benchmarks reward guessing over admitting uncertainty; errors arise even in simple settings where true and false statements are hard to tell apart.

Link to Part 3: a perfect truth-checker for arbitrary claims about programs would decide undecidable questions. From any source, we cannot have it.

illustrative the incentive, by arithmetic



With accuracy-only grading, guessing never loses. If wrong answers cost, abstaining wins below $p = 0.5$. The paper proposes changing how benchmarks score answers.

Kalai, Nachum, Vempala, Zhang, “Why Language Models Hallucinate”, arXiv:2509.04664 (4 Sep 2025). The plot is a textbook calculation of the incentive, not data from the paper.

Can an AI solve the halting problem? Can it solve NP-hard problems?

Halting (and Rice)

An AI model, run as a program, is a program. The diagonal construction D can be built against any fixed checker, including a neural one. So **no model can be right on every program**. It can be right on most programs people actually write.

NP-hard problems

A fast, always-correct model for SAT or TSP would be a polynomial algorithm, which would be a breakthrough in complexity theory. Models can propose *good* solutions; optimality and speed on all instances are not promised.

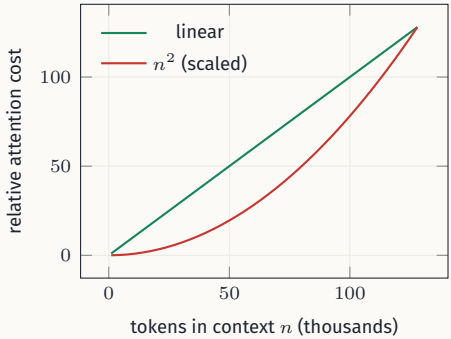
The limits of Parts 3–5 apply to **any** computing system. What AI changes is the *typical-case* performance, not the *worst-case* map.

Context limits and cost

Context window: a model reads only a finite number of tokens at once. Older text must be summarised or dropped.

Cost of attention. In the original transformer, self-attention compares every token with every other: about n^2 operations for n tokens per layer (Vaswani et al., 2017).

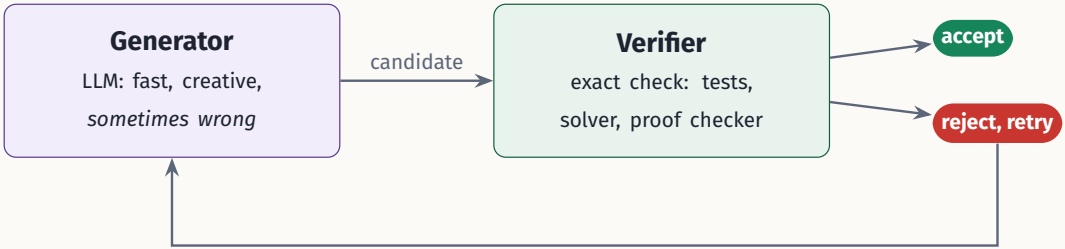
Reasoning costs tokens. More "thinking" tokens means more time and money per answer.



Schematic, *illustrative*. Many engineering tricks reduce the cost; they do not remove the trade-off between context length and compute.

Vaswani et al., "Attention is all you need", arXiv:1706.03762 (2017): self-attention layer complexity $O(n^2 \cdot d)$.

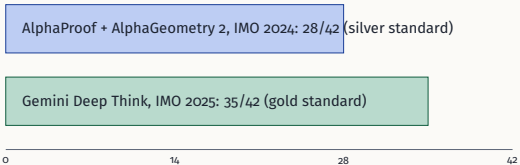
Generate and verify: the NP idea in modern AI



This mirrors NP: **guessing** may be hard, **checking** is cheap and exact. The system is only as trustworthy as the *verifier* and the statement it checks.

Examples: code with unit tests; a SAT solver confirming a plan; a proof assistant (Lean) checking a proof. Outputs that cannot be verified, such as essays, remain opinion.

Proof assistants and AI-assisted mathematics



Different systems: AlphaProof writes Lean proofs; the 2025 system worked in natural language.

AlphaProof (Google DeepMind): reinforcement learning to write proofs in Lean. Methodology in *Nature*, 12 Nov 2025.

Gemini Deep Think (IMO 2025): five of six problems solved. The IMO confirmed the answers are correct; its review does not validate the system.

Pattern: a model proposes, a formal system *certifies*. Limit: a certificate does not say we stated the right theorem.

Google DeepMind: “AI achieves silver-medal standard…” (25 Jul 2024); Hubert et al., “Olympiad-level formal mathematical reasoning with reinforcement learning”, Nature, 12 Nov 2025; DeepMind blog on Gemini Deep Think at IMO 2025 (21 Jul 2025).

Scorecard: what LLMs do well, and where to be careful

Strong evidence of ability	Known limits (with sources in this part)
Writing and explaining code, drafts, summaries	No built-in guarantee of correctness; hallucination (Kalai et al.)
Competition-level maths when paired with checkers (AlphaProof, IMO 2025)	Long exact computations drift; accuracy can collapse on large instances (Dziri; Shojaee, disputed by Lawsen)
Proposing candidates for humans or solvers to verify	Fixed compute per token; reasoning bounded by token budget (Merrill and Sabharwal)
Using tools: run code, search, call solvers	Finite context; cost grows with length ($\approx n^2$ attention)
Helping formalise and find proofs in Lean	Cannot beat undecidability or worst-case NP-hardness, like any program

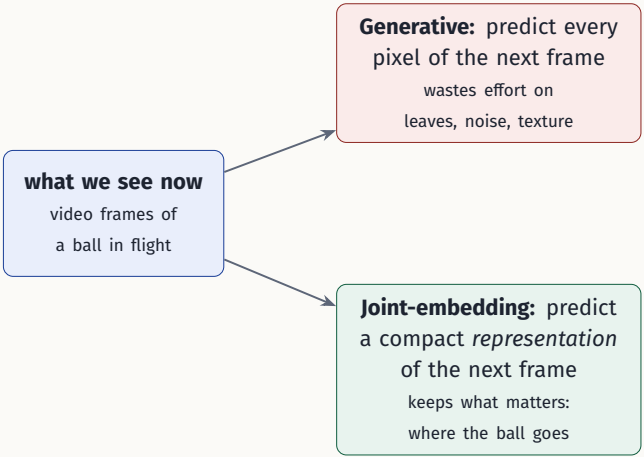
A sound habit: **trust** $\propto$ **verifiability**. The more cheaply and exactly an output can be checked, the more safely you can use AI to produce it.

Part 9

World models and JEPA

An alternative to next-token prediction, and what the evidence says so far

Two ways to predict what happens next



This is the central idea of **JEPA**, the Joint Embedding Predictive Architecture proposed by Yann LeCun: predict in *representation space*, not in pixel or token space.

LeCun, "A Path Towards Autonomous Machine Intelligence", position paper, OpenReview, 2022.

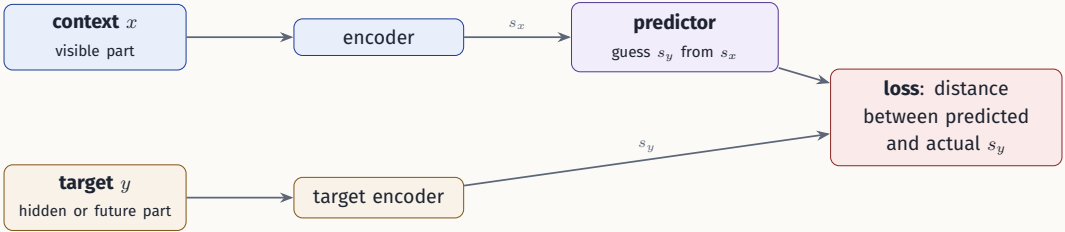
Autoregressive LLM versus JEPA

	Autoregressive LLM	JEPA-style world model
Predicts	the next token, from a finite vocabulary	the representation of a missing or future part of the input
Loss measured in	token probabilities	embedding space (distance between representations)
Output	text (or images, tokens) sampled step by step	no raw output needed; representations are used by other modules
Strength	fluent, flexible, huge text corpora	learning from video; ignoring unpredictable detail
Claimed weakness	errors compound over long outputs; limited grounding in the physical world	training can *collapse* without care; no generator, so less directly usable
Status	deployed at global scale	research systems; early products

The first four rows follow LeCun's published framing (position paper and I-JEPA / V-JEPA papers).

The last row is our summary of the situation in Nov 2026.

The JEPA recipe, drawn



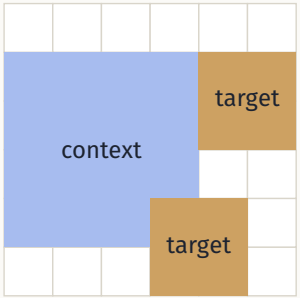
Collapse risk. If both encoders output a constant, the loss is zero and nothing is learned. JEPAs add safeguards (slowly updated target encoder, stop-gradients, regularisers) to prevent it.

Why representations? The target encoder can drop unpredictable detail, so the predictor only forecasts the *predictable* structure of the world.

I-JEPA (2023): predict image blocks in representation space

Idea. From one block of an image, predict the *representations* of several other blocks of the same image. No hand-made augmentations, no pixel reconstruction.

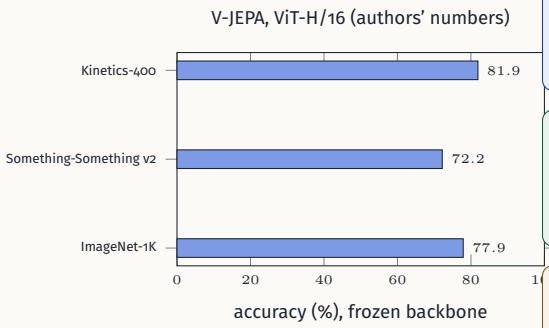
Findings (Assran et al.). The masking strategy matters most: targets must be large enough to be semantic, and the context informative and spread out. A ViT-Huge/14 trained on ImageNet in under three days on 16 A100 GPUs performed strongly on classification, counting and depth tasks.



schematic of block masking

Assran, Duval, Misra, Bojanowski, Vincent, Rabbat, LeCun, Ballas, “Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture”, arXiv:2301.08243 (Jan 2023).

V-JEPA (2024) and V-JEPA 2 (2025): from video to robots



V-JEPA (Bardes et al., Feb 2024): feature prediction as the *only* objective, on 2 million public videos; no text, no negatives, no reconstruction.

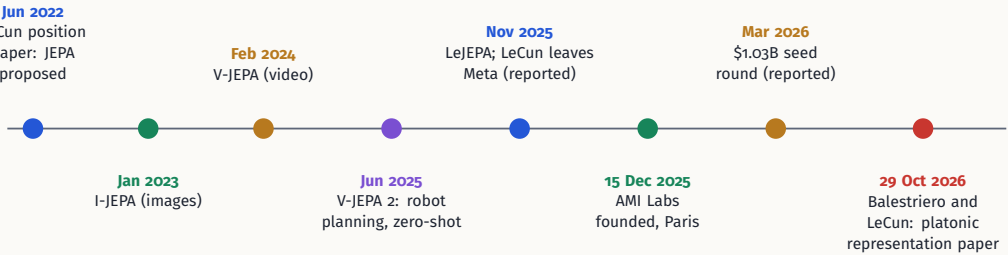
V-JEPA 2 (Assran et al., Jun 2025): over 1 million hours of video and images; 77.3% top-1 on Something-Something v2; 39.7 recall@5 on Epic-Kitchens-100.

V-JEPA 2-AC: post-trained on under 62 hours of robot video; used *zero-shot* for pick-and-place on Franka arms in two labs.

Reading these numbers: they come from the authors' papers. Whether JEPA beats alternatives overall is what the debate is about.

Bardes et al., arXiv:2404.08471 (2024); Assran et al., "V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning", arXiv:2506.09985 (11 Jun 2025).

Timeline: the JEPA line of work (all dates from the papers and reports cited)



Research papers (blue-green) are the evidence; company news is context. Preprints marked 2026 have not yet been independently replicated, as far as we could find.

arXiv:2301.08243, 2404.08471, 2506.09985, 2511.08544, 2610.39302; Wikipedia, AMI Labs (citing NYT, WSJ, Wired, 10 Mar 2026).

Newer theory: LeJEPA and the “platonic” paper

LeJEPA (Balestriero and LeCun, Nov 2025): a theory of JEPAs and a training objective, SIGReg, that pushes embeddings toward an isotropic Gaussian. One main hyperparameter; about 50 lines of code; 79% ImageNet-1k linear accuracy with a frozen ViT-H/14, over 60 architectures tried.

“Proving and Improving the Platonic Representation Hypothesis with JEPAs” (Balestriero and LeCun, 29 Oct 2026): proves that JEPAs trained on different modalities end up with equivalent representations up to a rotation; reports that a 109M-parameter text JEPA aligned with image encoders better than language models up to 72B parameters.

Status: one arXiv preprint (steps 1–2 of the Part 7 ladder). Results are the authors’ own; no independent replication known to us.

Balestriero and LeCun, “LeJEPA”, arXiv:2511.08544 (Nov 2025). Balestriero and LeCun, arXiv:2610.39302 (29 Oct 2026). We read the arXiv abstracts; not the full papers.

LeCun's critique of LLMs and his new lab

The critique (as reported). LeCun left Meta in November 2025, arguing that the leading AI companies were heading toward a dead end by focusing mainly on large language models. His alternative: world models that understand the physical world, keep memory, reason and plan.

Counter-view. Many labs report strong results from scaling LLMs with reasoning and tools (Part 8). Whether LLMs alone reach general intelligence is an open empirical question.

AMI Labs (Advanced Machine Intelligence): Paris-based, founded 15 Dec 2025. Executive chairman: LeCun; CEO Alexandre LeBrun; Chief Science Officer Saining Xie.

Funding (reported): a $1.03 billion seed round in March 2026. Stated focus: manufacturing, biomedicine, robotics. No products listed; none expected for about five years.

Wikipedia, "Advanced Machine Intelligence Labs" (last edited 19 Nov 2026), citing NYT, WSJ, Le Figaro and Wired of 10 Mar 2026 and company records; several citations on that page are broken, so treat funding details as reported, not confirmed.

Other world-model efforts, and a fair summary

Other approaches. *Dreamer* (Hafner et al., *Nature* 2025) learns a world model in latent space and plans inside it, mastering many control tasks with one setup. Reports list further players working on world models, including World Labs, Google DeepMind, OpenAI and Meta.

What we cannot say. We did not find peer-reviewed evidence, by Nov 2026, that JEPA-based systems outperform the best LLM-based systems at general reasoning or at deployed products.

What is clear. Predicting in representation space is an active, productive research line with open code and papers; V-JEPA 2 shows robot planning from a small amount of robot data.

Limits still apply. Planning inside a world model is a *search* problem: in general NP-hard or worse. A world model does not repeal Parts 3–5. It may give better heuristics.

Hafner et al., "Mastering diverse control tasks through world models", Nature 640, 2025 (DreamerV3). Competitor list: Wikipedia, AMI Labs page (cites Maddyness, Mar 2026). Not used: blog posts and aggregator pages about further 2026 JEPA releases that we could not trace to a paper.

Part 10

What to expect

Honest scenarios, open problems, and what to learn

Three honest scenarios (our judgement, not a forecast)

A. Powerful assistants plus verifiers

AI proposes, exact tools check. Programmers, scientists and analysts work with it daily. Biggest gains where checking is cheap (code, formal maths, optimisation).

Signpost: more claims arrive with machine-checked artefacts.

B. A formalisation wave

important proofs and critical software increasingly come with proof-assistant certificates. Human work shifts toward stating the right theorems and understanding.

Signpost: Lean/Coq checks become routine for journals and standards.

C. Surprise and diversification

scaling slows in some areas; world models, hybrids or new ideas take over there. Or a theory surprise, such as the ETH claim, reshapes what we assume.

Signpost: independent benchmarks and replications disagree with lab claims.

None of these changes the maps of Parts 3–5: halting stays undecidable and NP-complete problems stay (apparently) hard unless a proof says otherwise. What can change fast is *how much of the typical case* machines handle.

Open problems that are still open (and a few that moved)

Problem	Why it matters	Status, 23 Nov 2026
P versus NP	efficient search, security, AI	open; Clay prize
Is P = BPP?	is randomness needed?	open; believed equal
Can factoring be done fast classically?	RSA and its replacements	open
Is NP inside BQP?	what quantum can do	open; believed no
Exponential Time Hypothesis	basis of fine-grained lower bounds	**claimed refuted (Part 7)**
Riemann hypothesis	primes, number theory	**claimed proved (Part 7)**
Collatz conjecture	simple loops, hard to analyse	open
Do world models beat LLMs?	future of AI systems	open, empirical

Open problems are where the field is alive. Notice how many are “is X really hard?”. Hardness is, for now, mostly an *assumption*.

What to learn to stay relevant

Algorithms and cost

read a program's cost, spot the exponential hiding in a loop, know when to use a solver.

Verification

tests, types, invariants, proofs; how to check AI output cheaply and exactly.

Logic and probability

reason about claims, errors, and uncertainty; read a proof sketch.

Programming with tools

Python, SQL, version control; use AI as a collaborator you supervise.

Source criticism

the claims checklist from Part 7: primary source, replication, incentives.

Domain and communication

understand a field deeply; explain results to people who must act on them.

Recap: ten sentences

1. A Turing machine (tape, head, states, rules) captures what “computing” means; every known computer can be simulated by one.
2. Church–Turing thesis: the informal idea of “mechanical computation” equals what Turing machines compute.
3. A universal machine runs any program given as data; programs can be fed to programs, including themselves.
4. Halting problem: no program decides, for every program and input, whether it stops (diagonal argument).
5. Rice: every non-trivial question about a program’s behaviour is undecidable; real tools say “unknown” or accept errors.
6. Cost = steps as a function of input size; polynomial is feasible, exponential defeats faster hardware.
7. P: quick to solve. NP: quick to check. $P \subseteq NP$; whether they are equal is open ($1M).
8. NP-complete problems (SAT, TSP, Sudoku, …) are all equivalent via reductions; solvers and heuristics work well on real instances.
9. Quantum computers do not change computability and are not believed to solve NP-complete problems; cryptography rests on assumed hardness.
10. AI systems are programs: they inherit these limits. Verify what matters, and rate claims by evidence, not by headlines.

Review questions, part 1: concepts

1. Name the four parts of a Turing machine and say what each does.
2. Why is the Church–Turing thesis a *thesis* and not a theorem?
3. Explain, in three sentences, the diagonal argument for the halting problem.
4. Give two examples of undecidable questions that matter to a software company.
5. What is Rice's theorem, and why does it stop a perfect antivirus?
6. Why does a 1000× faster computer not rescue an exponential algorithm?
7. Define P and NP. Why is NP *not* "non-polynomial"?
8. What is a reduction? Why does reducing A to B show B is at least as hard as A?
9. Give three reasons experts believe P $\neq$ NP.
10. Why can quantum computers not simply "try all answers at once"?

Review questions, part 2: AI, claims and calculations

11. A news article says “AI proves famous conjecture”. List five questions you ask before believing it.

12. What does a Lean-checked proof guarantee, and what does it *not*?

13. Why can no AI model decide halting for all programs? Why can it still be useful on most?

14. Explain “generate and verify” with an example from your own coding.

15. What is the central idea of JEPA, and what is representation collapse?

16. A solver needs 2^n μs. How long for $n = 40$? For $n = 60$?

17. A model is 97% right per step. Chance a 20-step chain is fully right (independent errors)?

18. A tour problem has 12 cities. How many distinct tours $(n-1)!/2$?

19. Sketch a Turing machine that decides whether a binary number is even.

20. Classify: *sorting*, *SAT*, *halting*, $n \times n$ *chess*. (P, NP-complete, undecidable, EXP-complete)

Answers 16–18: 13 days; 36,000 years; 0.54; 19,958,400.

Hands-on: five things to try this week

1. Run `python3.14 tm.py`. Change one rule of `inc` so it *subtracts* one. Test it on 1, 10, 1000.

2. Write a Turing machine for “is the binary number even?” and count its steps.

3. Run `brute.py`. Predict the time for 24 items before running. How far off were you?

4. Encode a 4×4 Sudoku as clauses and feed it to `sat.py`.

5. Apply the claims checklist to one news item.

Files (all Python 3.14)

```
tm.py        Turing machine
tmcost.py    step counts
halting.py   diagonal demo
collatz.py   open loop
growth.py    growth table
verify.py    check vs find
brute.py     brute-force timing
sat.py       tiny SAT solver
satphase.py  phase transition
factor.py    trial division
run_all.sh   regenerate outputs
```

Where to read next

Books

- M. Sipser, *Introduction to the Theory of Computation*, 3rd ed., 2012.
- S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, CUP 2009.
- C. Petzold, *The Annotated Turing*, Wiley 2008.
- S. Aaronson, *Quantum Computing since Democritus*, CUP 2013.
- M. Garey and D. Johnson, *Computers and Intractability*, Freeman 1979.

Original papers and sources used in this deck

- Turing 1936; Cook 1971; Karp 1972; Shor 1994; Grover 1996.
- OpenAI, 26 Oct 2026 (ETH); Fortnow blog, 28 Oct 2026.
- arXiv:2611.00517; 2611.02650; 2310.07923; 2305.18654; 2506.06941; 2506.09250; 2509.04664.
- arXiv:2301.08243; 2404.08471; 2506.09985; 2511.08544; 2610.39302; 2408.13687.
- AFP via Phys.org, 20 Nov 2026 (Riemann claim).

Sources for the claims in Parts 7–9 (checked 23 Nov 2026)

Item	Source (URL)
OpenAI ETH announcement	openai.com/index/building-scientific-understanding-with-academic-communities/
OpenAI ETH preprint (not read)	github.com/openai/math, folder “The-Exponential-Time-Hypothesis-is-False-October-25-2026”
Expert comment, Fortnow	blog.computationalcomplexity.org/2026/10/the-exponential-time-refutation.html
Critique, Dias and Lima	arxiv.org/abs/2611.00517
MIT Hardness Group	arxiv.org/abs/2611.02650
ETH background and claim summary	en.wikipedia.org/wiki/Exponential_time_hypothesis
Riemann claim, AFP	phys.org/news/2026-11-anthropic-claude-math-greatest-riddle.html
Lean certification (secondary)	officechai.com/ai/ (“Lean Community Certifies Levent Alpöge’s Claude-Assisted Proof”, 21 Nov 2026)
AMI Labs	en.wikipedia.org/wiki/Advanced_Machine_Intelligence_Labs
JEPA and LLM papers	arxiv.org/abs/ + numbers on the previous slide