CS251

Great Ideas in *Theoretical* Computer Science

Time Complexity

And the Power of Algorithms

2 Main Questions in Theory of Computation

Computability of a problem:

Is there an algorithm to solve it?

Complexity of a problem:

Is there an *efficient* algorithm to solve it?

- time
- space (memory)
- randomness
- quantum resources

Computational Complexity (Practical Computability)

Simulations (e.g. of physical or biological systems)

- tremendous applications in science, engineering, medicine,…

Optimization problems

- arise in essentially every industry

Social good

- finding efficient ways of helping others

Artificial intelligence

Security, privacy, cryptography

- applications of computationally hard problems

list goes on

.

.

.

Computational Complexity (Practical Computability)

- How do we define computational complexity?

- What is the right level of abstraction to use?

- How do we analyze complexity?

- What are some interesting problems to study?

- What can we do to better understand
  the complexity of problems?

⋮

Kurt Friedrich Gödel (1906-1978)

One of the most important logicians in history.

John von Neumann (1903-1957)

Contents [hide]

- Mathematical formulation of **quantum mechanics**.

- Founded the field of **game theory** in mathematics.

- Created some of the first **general-purpose computers**.

Gödel's letter to von Neumann (1956)

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max\_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

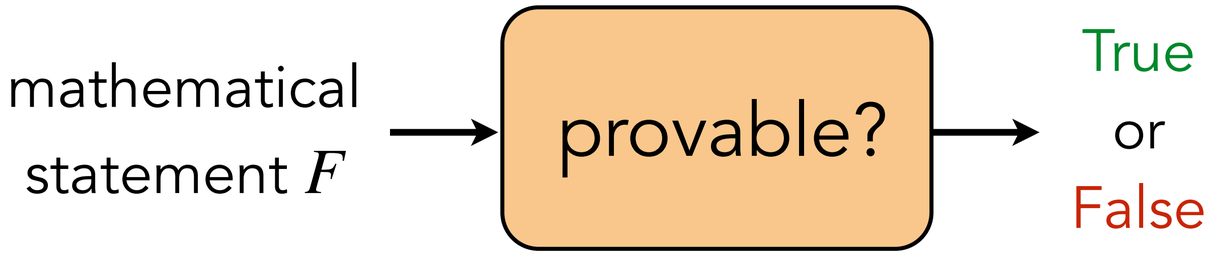
Gödel's letter to von Neumann (1956)

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

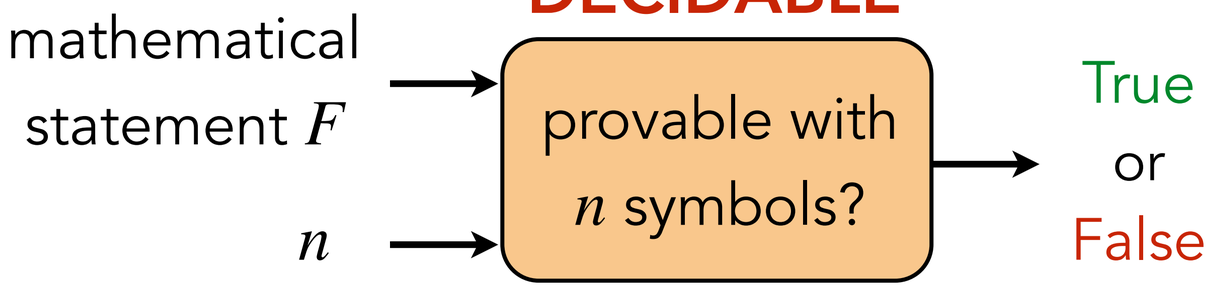
Gödel's letter to von Neumann (1956)

ENTSCHEIDUNGSPROBLEM

UNDECIDABLE



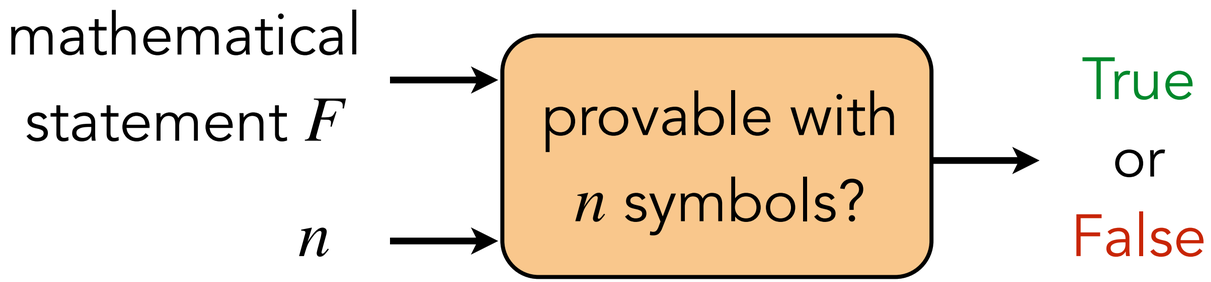
DECIDABLE



Gödel's letter to von Neumann (1956)

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

Gödel's letter to von Neumann (1956)



$\Psi(F, n) =$ the number of steps required for input (F, n)

$$\varphi(n) = \max_{F} \Psi(F, n)$$

a worst-case notion of running time

? How fast does $\varphi(n)$ grow for an optimal machine?

asymptotic analysis

Gödel's letter to von Neumann (1956)

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

Gödel's letter to von Neumann (1956)

One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

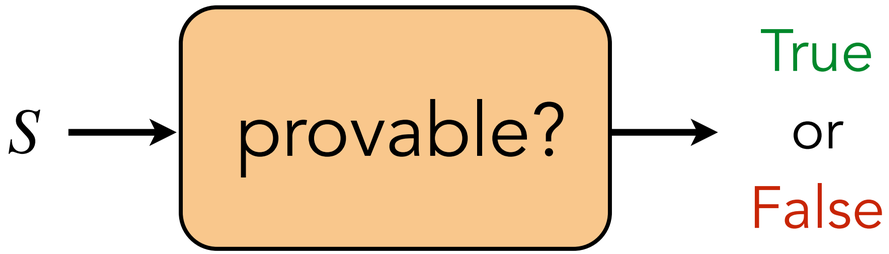
Gödel's letter to von Neumann (1956)

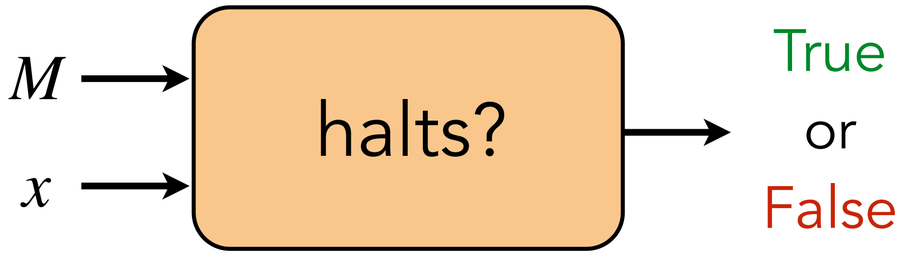
One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max\_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly.

This is the first formalization of the **P** vs **NP** problem.

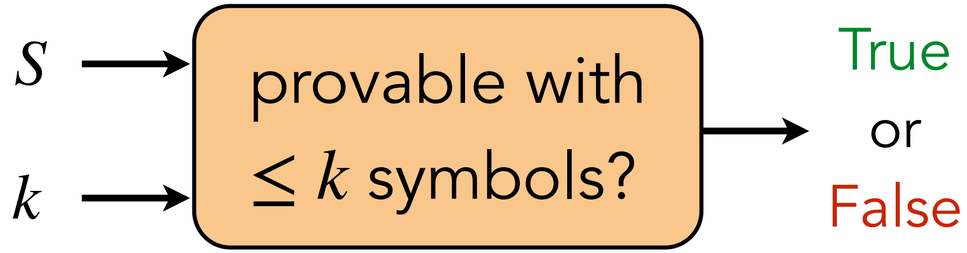
Is undecidability a real issue?

Yes, the following (and many more) is uncomputable:





But these are computable:



But are they **practically computable**?

What really matters is **computational complexity.**

Part 1 of CS251:

Understand the divide between
computable and uncomputable.

Part 2 of CS251:

Understand the divide between
practically computable and practically uncomputable.

What is the meaning of:

"The (asymptotic) running time complexity of algorithm A is $O(n^2)$."

Great Ideas in Complexity Analysis

The (worst-case) **running time of an algorithm** A is a function

$$T_A(n) = \max_{\substack{\text{inputs } x \\ \text{of length } n}} \#\text{ steps } A(x) \text{ takes.}$$

- **Worst-case analysis.**
- **Asymptotic analysis and the Big-O notation.**
- **Polynomial-time.**

- **Worst-case analysis.**

Why worst-case?

We are not dogmatic about it.

Can study “average-case” (random inputs)

Can try to look at “typical” instances.

…

BUT worst-case analysis has its advantages:

- An ironclad guarantee.
- Hard to define “typical” instances.
- Random instances are often not representative.
- Often much easier to analyze.

Great Ideas in Complexity Analysis

The **running time of an algorithm** A is a function

$$T_A(n) = \max_{\substack{\text{inputs } x \\ \text{of length } n}} \#\text{ steps } A(x) \text{ takes.}$$

- **Worst-case analysis.**
- **Asymptotic analysis and the Big-O notation.**
- **Polynomial-time.**

- **Asymptotic analysis and the Big-O notation.**

$$T(n) = \frac{1}{2}n^2 + \frac{3}{2}n + 1.$$

Analogous to “too many significant digits”.

“Sweet spot” of Big-O

- coarse enough to suppress details like *programming language*, *compiler*, *architecture*,…

- sharp enough to make comparisons between different *algorithmic approaches*.

Great Ideas in Complexity Analysis

The **running time of an algorithm** A is a function

$$T_A(n) = \max_{\substack{\text{inputs } x \\ \text{of length } n}} \#\text{ steps } A(x) \text{ takes.}$$

- **Worst-case analysis.**
- **Asymptotic analysis and the Big-O notation.**
- **Polynomial-time.**

- **Polynomial-time.** $O(n^k)$ for some constant k.

  In practice:

$O(n)$	Awesome! Like really awesome!
$O(n \log n)$	Great!
$O(n^2)$	Kind of efficient.
$O(n^3)$	Barely efficient. (???)
$O(n^5)$	Would not call it efficient.
$O(n^{10})$	Definitely not efficient!
$O(n^{100})$	WTF?

- **Polynomial-time.**

In theory:	Polynomial time	Efficient.
	Otherwise	Not efficient.

- Poly-time is not meant to mean “efficient in practice”

- It means “You have done something extraordinarily better than brute force search.”

- Poly-time: mathematical insight into a problem’s structure.

- If you show, say Factoring Problem, has running time $O(n^{100})$, it will be the best result in CS history.

- **Polynomial-time.**

  In theory: Polynomial time — Efficient.

  Otherwise — Not efficient.

  - Robust to notions:
    elementary step, what model we use,
    reasonable encoding of input, implementation details.

  - Nice closure property: Plug in a poly-time alg. into another poly-time alg. —> poly-time

  - Big exponents don't really arise.

  - If it does arise, usually can be brought down.

- **Polynomial-time.**

  In theory: Polynomial time — Efficient.

  Otherwise — Not efficient.

- **Summary**: Poly-time vs not poly-time
  is a qualitative difference, not a quantitative one.

poll.cs251.com

What is the running time in terms of the input length?

```
def isPrime(N):
    if (N < 2):
        return False
    for factor in range(2, N):
        if (N % factor == 0):
            return False
    return True
```

Poll Answer

Algorithms on numbers involve **BIG** numbers.

361850278866613110698659328152149711045574302116926035853677593202076268610172378462348732698071029701288743560214819642328577822956716750213930654736953943653222082116941587830769649826310589717739181525033220266350650989268038319483927388150543242207717912183888281996148408052302196889866637200606252650131096492647520509000398417612205871116456794655904497168360442407699634271830465447980211682970134907741400904763482906718227439612036981423070996643455133414637616824423860107889741058131271306226214208636008224651510961018

This is still small! Imagine having thousands of digits.

Poll Answer

$B =$ 569303002052399999347964290462191172509856702055625810276625148723403109442 9

$B \approx 5.7 \times 10^{75}$ (5.7 quattorvigintillion)

$\text{len}(B) = 251$

An algorithm repeating B times is **practically uncomputable**.

$\text{len}(B) =$ # bits to write B

$\text{len}(B) \approx \log_2 B$

Poll Answer

```
def isPrime(N):
    if (N < 2):
        return False
    for factor in range(2, N):
        if (N % factor == 0):
            return False
    return True
```

iterations: $\approx N$

$N = 2^{\log_2 N} = 2^{\text{len}(N)} = 2^n$

exponential in input length

CS251 computational model for complexity analysis

The model matters if you are interested in more refined complexity analysis.

CS251 computational model for complexity analysis

The Random-Access Machine (RAM) model

Good combination of reality & simplicity.

+ , - , / , *, <, >, etc.	e.g. 15*251	takes 1 step for "small numbers"
memory access	e.g. A[94]	takes 1 step

Small number: Bounded by a polynomial in **input length**.

Large number: Not small.

Arithmetic operations/comparisons take 1 step
only if the numbers are small.

Example: Are the numbers small or large?

Small number: Bounded by a polynomial in **input length**.

Large number: Not small.

```
def foo(int B):
    return B + B
```

```
def bar(string s):
    i = 0
    while (i < len(s)):
        print(s[i])
        i = i + 1
```

What is the complexity of addition, multiplication, etc. when the numbers are large?

Integer Addition

```
def sum(int A, int B):
    for i = 1 to B:
        A += 1
    return A
```

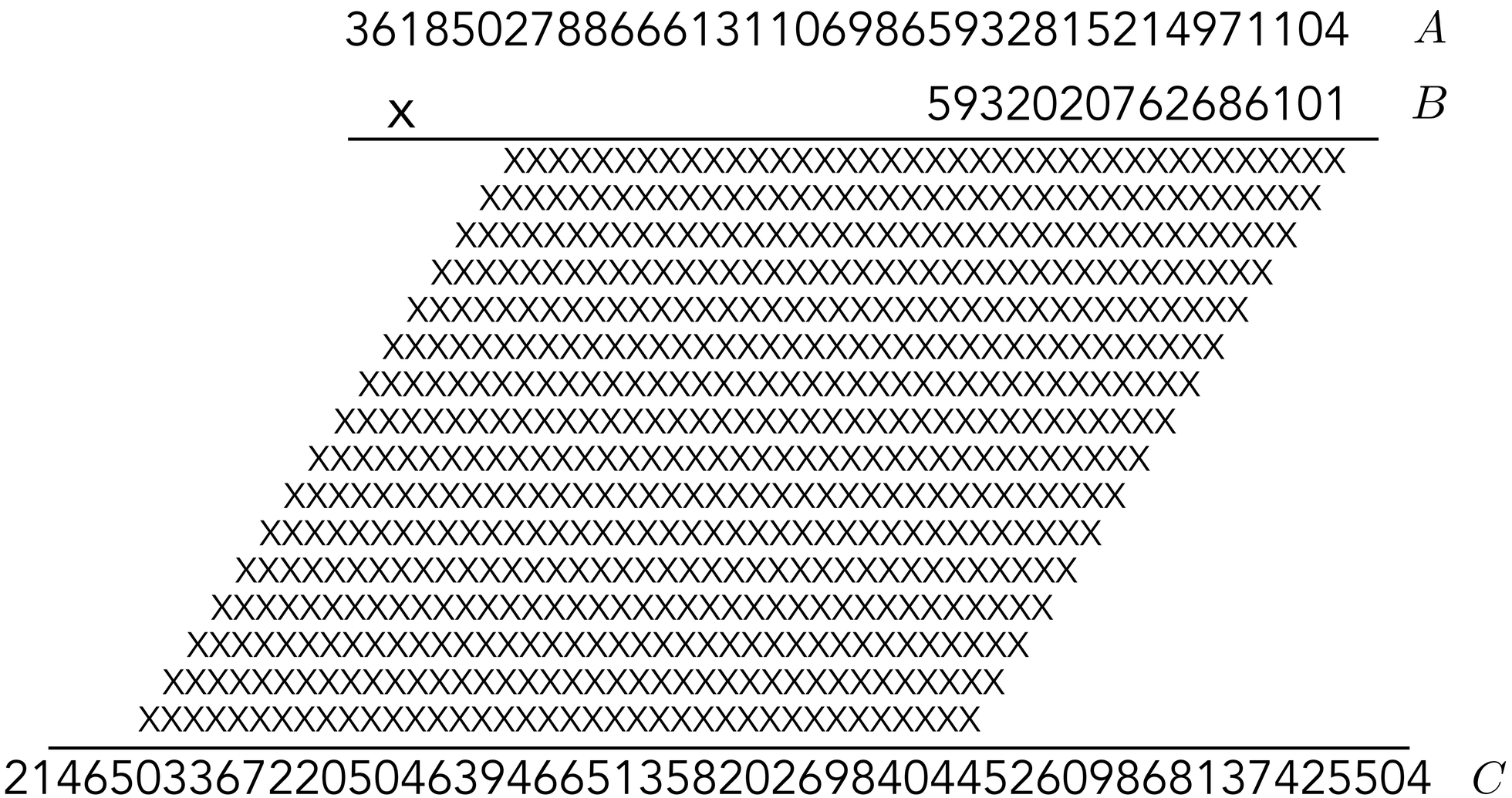
What is the running-time of this algorithm?

Integer Addition

$$
\begin{array}{rl}
361850278866613110698659328152149711 04 & A \\
+\ 65743021169260358536775932020762686101 & B \\
\hline
1019280490559216696066418648359776572 05 & C
\end{array}
$$

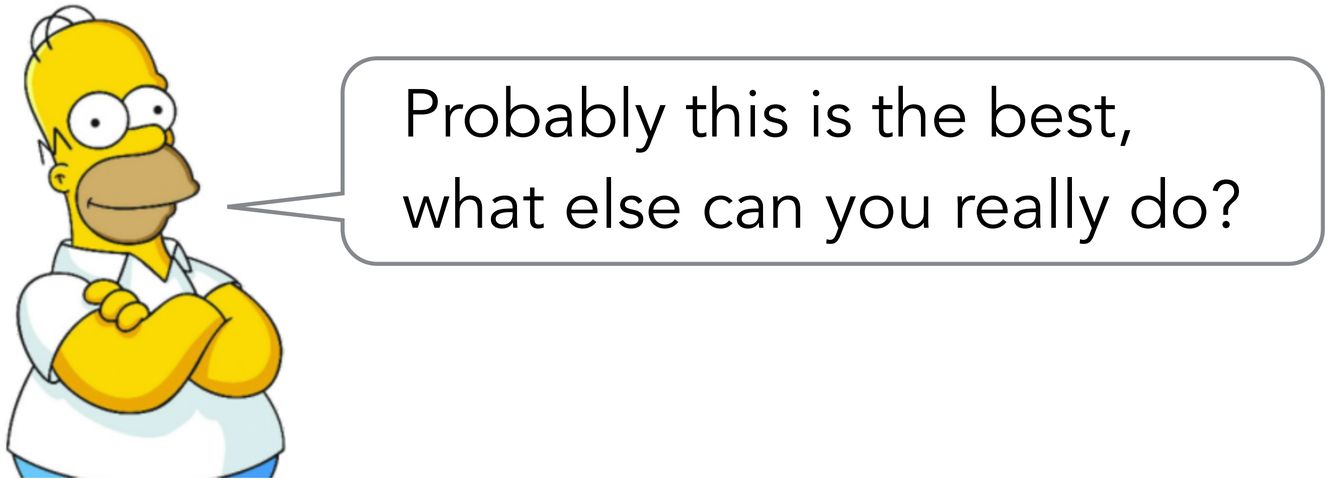
 # steps to produce C is $O(n)$

Integer Multiplication



 # steps: $O(\text{len}(A) \cdot \text{len}(B)) = O(n^2)$

Integer Multiplication



A good algorithm designer always thinks:

HOW CAN WE DO BETTER?!?

What algorithm does Python use?

Integer Multiplication

n = length of one of the numbers

a b

$A = 5\,6\,7\,8 = a \cdot 10^{n/2} + b$

$B = 1\,2\,3\,4 = c \cdot 10^{n/2} + d$

c d

$$A \cdot B = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d)$$

$$= ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$$

Use recursion!

Integer Multiplication

$$A = \underbrace{56}_{a}\underbrace{78}_{b} = a \cdot 10^{n/2} + b$$

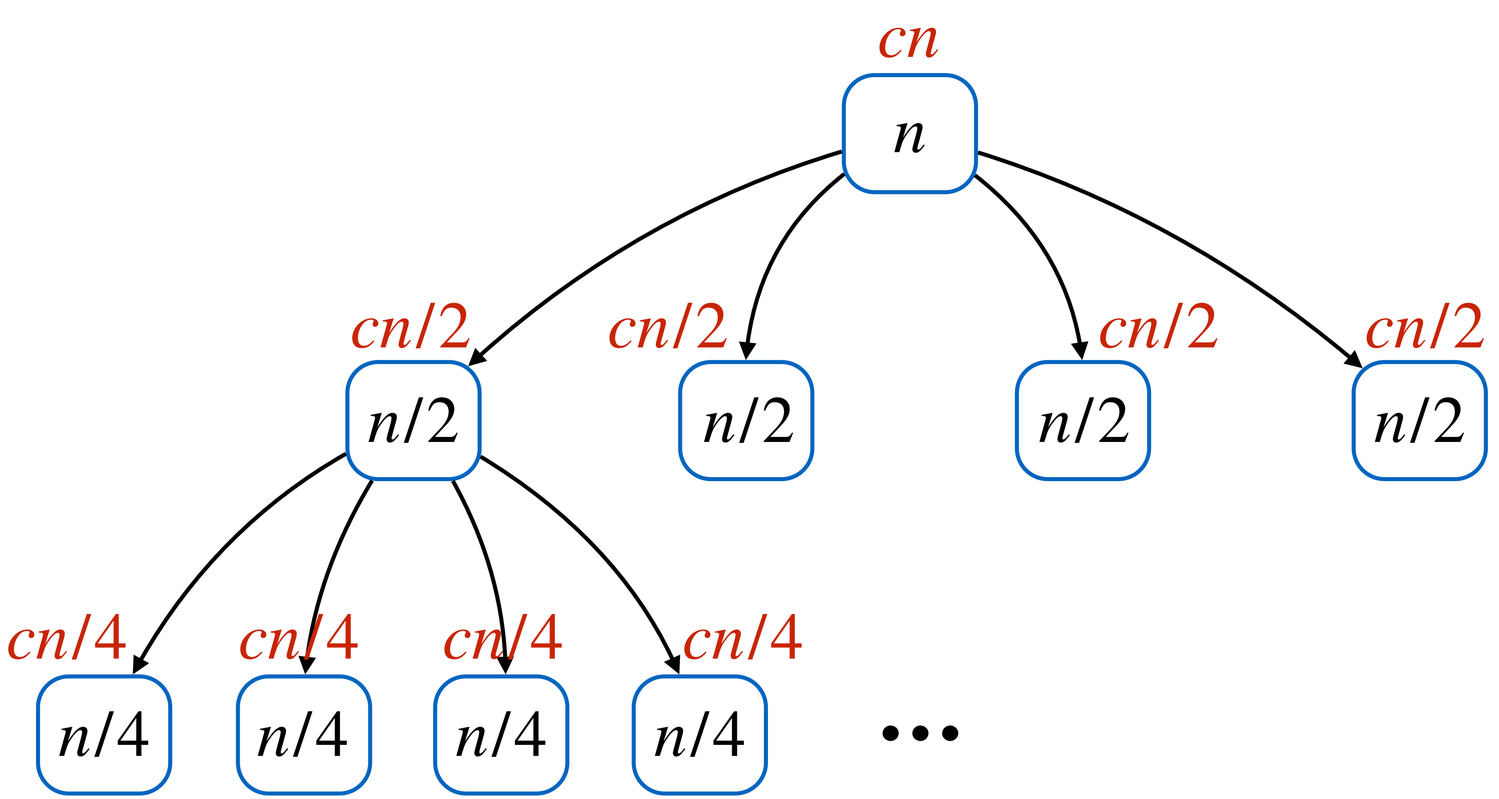
$$B = \underbrace{12}_{c}\underbrace{34}_{d} = c \cdot 10^{n/2} + d$$

$$A \cdot B = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d)$$

$$= ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$$

- Recursively compute ac, ad, bc, and bd.
- Do the multiplications by 10^n and $10^{n/2}$. $\rightarrow O(n)$
- Do the additions. $\rightarrow O(n)$

$$T(n) \leq 4 \cdot T(n/2) + O(n)$$



distinct nodes at level j: 4^j

work done per node at level j: $c(n/2^j)$

} $cn2^j$ for level j

levels: $\log_2 n$

Total: $\sum_{j=0}^{\log_2 n} cn2^j = O(n^2)$

Integer Multiplication

$$A \cdot B = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$$

Hmm, we don't really care about ad and bc.
We just care about their sum.
Maybe we can get away with 3 recursive calls?

@cilantro.the.giraffe

Integer Multiplication

$$A \cdot B = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d)$$
$$= ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$$

$$(a+b)(c+d) = ac + ad + bc + bd$$

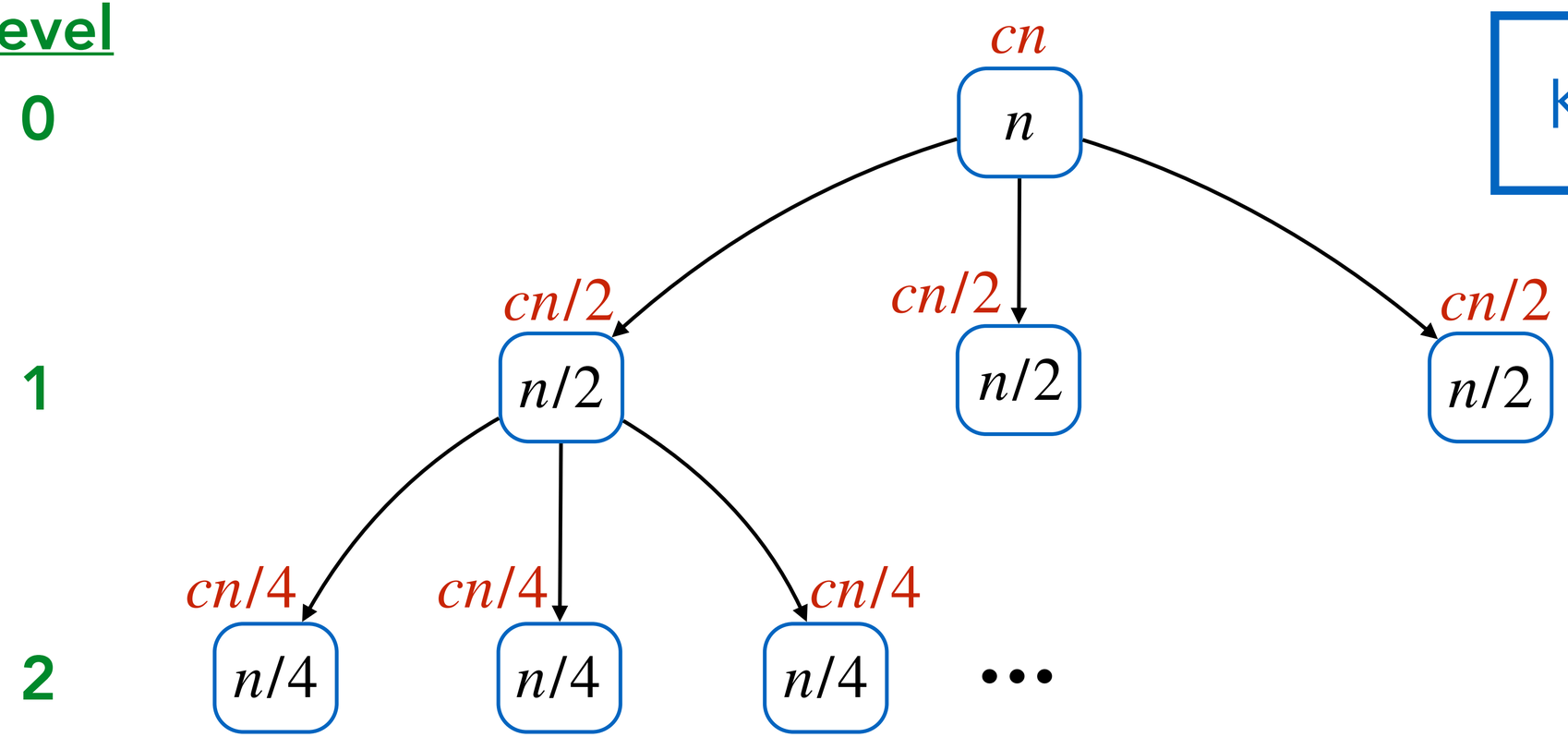
- Recursively compute: ac, bd, $(a+b)(c+d)$

- Then: $(ad + bc) = (a+b)(c+d) - ac - bd$

$$T(n) \leq 3 \cdot T(n/2) + O(n)$$

Is this better??

Karatsuba Algorithm



distinct nodes at level j: 3^j

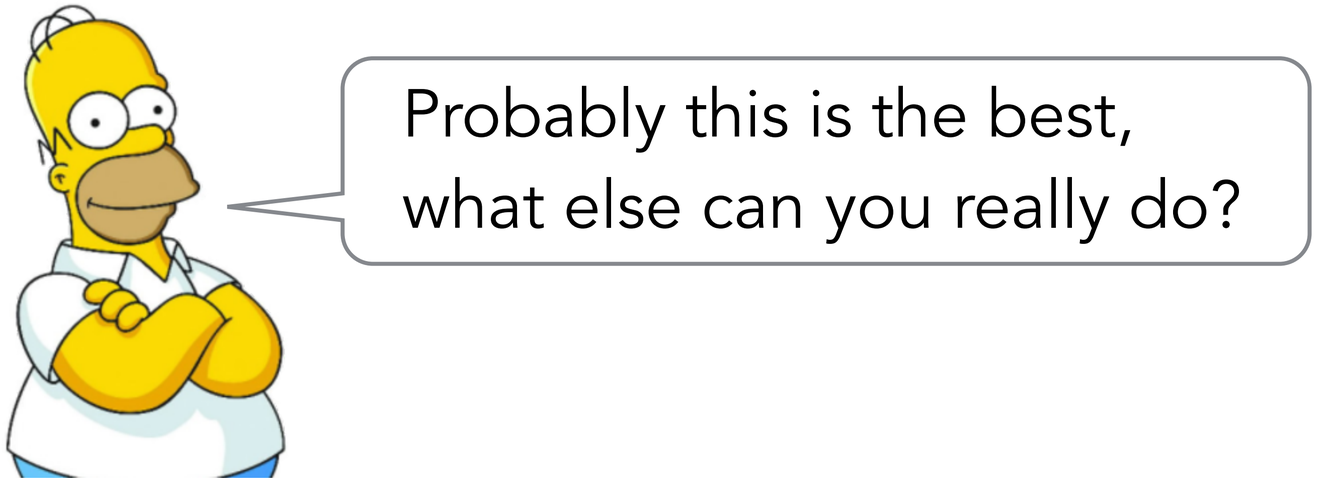
work done per node at level j: $c(n/2^j)$

} $cn(3^j/2^j)$ for level j

levels: $\log_2 n$

Total: $\sum_{j=0}^{\log_2 n} cn(3^j/2^j) = O(n^{\log_2 3})$

Integer Multiplication



A good algorithm designer always thinks:

HOW CAN WE DO BETTER?!?

Integer Multiplication

Cut the integer into 3 parts of length $n/3$ each.

Replace 9 multiplications with only 5.

$$T(n) \leq 5 \cdot T(n/3) + O(n)$$

$$T(n) = O(n^{\log_3 5})$$

Can do $T(n) = O(n^{1+\epsilon})$ for any $\epsilon > 0$.

Integer Multiplication

Fastest known: $n \log n$ Harvey, Hoeven (2019)

Lessons

It is not easy to understand the power of algorithms.

Always try to do better!

What is next?

- Graphs and more examples of efficient algorithms
- Polynomial time vs Exponential time