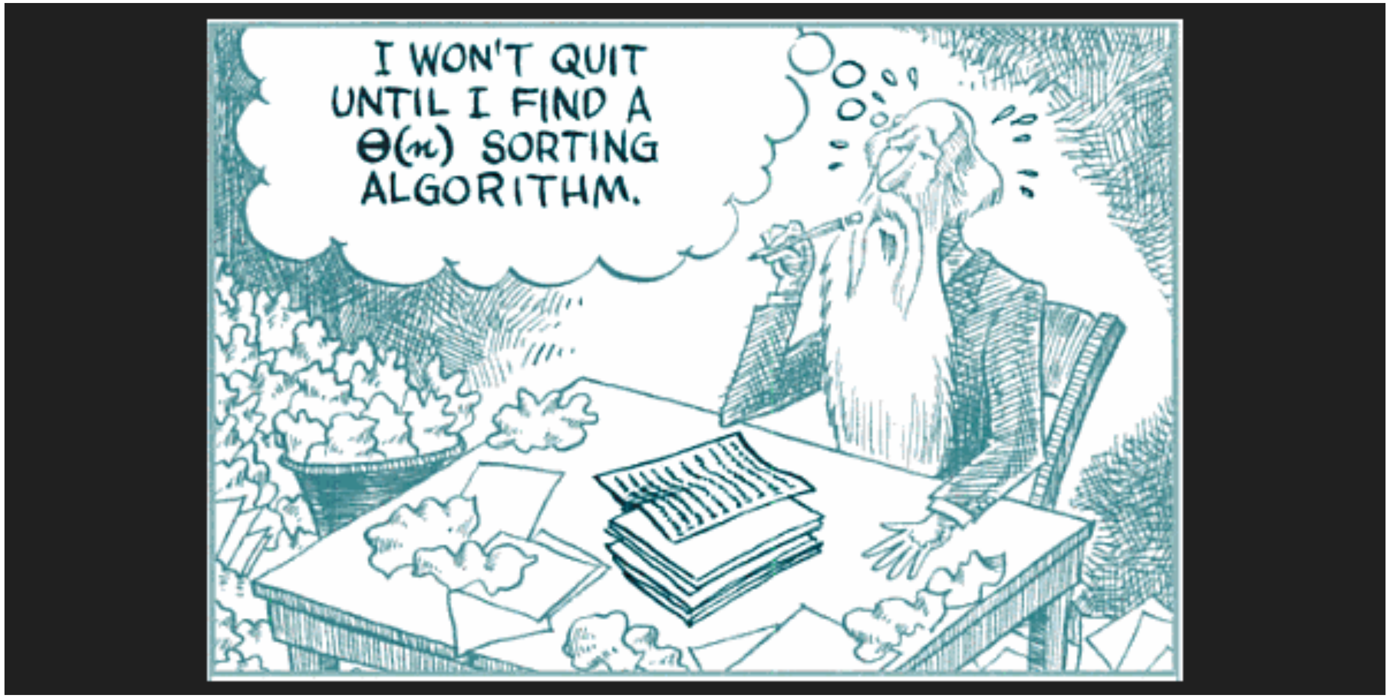
15-251
Great Theoretical Ideas in Computer Science

Lecture 9:
Introduction to Computational Complexity



February 14th, 2017

Poll

What is the running time of this algorithm?
Choose the tightest bound.

```
def twoFingers(s):
    lo = 0
    hi = len(s)-1
    while (lo < hi):
        if (s[lo] != 0 or s[hi] != 1):
            return False
        lo += 1
        hi -= 1
    return True
```

- $O(1)$
- $O(\log n)$
- $O(n^{1/2})$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

What have we done so far?

What will we do next?

What have we done so far?

> Introduction to the course

"Computer science is no more about computers than astronomy is about telescopes."

> Strings and Encodings

> Formalization of computation/algorithm

Deterministic Finite Automata

Turing Machines

What have we done so far?

> The study of computation

Computability/Decidability

- Most problems are undecidable.
- Some very interesting problems are undecidable.

But many interesting problems are decidable!

What is next?

> The study of computation

Computability/Decidability

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?

⋮

Why is computational complexity important?

Why is computational complexity important?

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

list goes on

⋮

Security, privacy, cryptography

- applications of computationally hard problems

Why is computational complexity important?

CMI

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EUCLID EVENTS

Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

1 million dollar question

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

(or maybe 6 million dollar question)

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Navier–Stokes Equation

This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.

Hodge Conjecture

The answer to this conjecture determines how much of the topology of the solution set of a system of algebraic equations can be defined in terms of further algebraic equations. The Hodge conjecture is known in certain special cases, e.g., when the solution set has dimension less than four. But in dimension four it is unknown.

Poincaré Conjecture

In 1904 the French mathematician Henri Poincaré asked if the three dimensional sphere is characterized as the unique simply connected three manifold. This question, the Poincaré conjecture, was a special case of Thurston's geometrization conjecture. Perelman's proof tells us that every three manifold is built from a set of standard pieces, each with one of eight well-understood geometries.

Birch and Swinnerton-Dyer Conjecture

Supported by much experimental evidence, this conjecture relates the number of points on an elliptic curve mod p to the rank of the group of rational points. Elliptic curves, defined by cubic equations in two variables, are fundamental mathematical objects that arise in many areas: Wiles' proof of the Fermat Conjecture, factorization of numbers into primes, and cryptography, to name three.

Goals for this week

Goals for the week

1. What is the right way to study complexity?

- using the right language and level of abstraction

- upper bounds vs lower bounds

- polynomial time vs exponential time

2. Appreciating the power of algorithms.

- analyzing some cool (recursive) algorithms

What is the right language and level of abstraction for studying computational complexity?

What is the meaning of:

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

We have to be careful

Size matters

Value matters

Model matters

Size matters

sorting bazillion numbers **>** sorting 2 numbers.

Running time of an algorithm depends on **input length**.

n = input length/size

n is usually: # bits in a binary encoding of input.

sometimes: explicitly defined to be something else.

GREAT IDEA # 1

Running time of an algorithm is a function of **n**.

(But what is **n** going to be mapped to?)

We have to be careful

Size matters ✓

Value matters

Model matters

Value matters

Not all inputs are created equal!

Among all inputs of length **n**:

- some might take 2 steps
- some might take bazillion steps.

GREAT IDEA # 2

Running time of an algorithm is a worst-case function of **n**.

n $\mapsto$ # steps taken by the worst input of length **n**

Value matters

Why worst-case?

We are not dogmatic about it.

Can study "average-case" (random inputs)

Can try to look at "typical" instances.

Can do "smoothed analysis".

…

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.

We have to be careful

Size matters 

Value matters 

Model matters

Model matters

TM, C, Python, JAVA, CA all equivalent:

With respect to ***decidability***, model does **<u>not</u>** matter.

The same is not true with respect to *complexity*!

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

How many steps required to decide *L*?

Facts:

$O(n \log n)$ is the best for 1-tape TMs.

$O(n)$ is the best for 2-tape TMs.

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

A function in Python:

# of steps

```
def twoFingers(s):
    lo = 0                                   1
    hi = len(s)-1                            1
    while (lo < hi):                         1
        if (s[lo] != 0 or s[hi] != 1):       3? 4? 5?
            return False                     1
        lo += 1                              1
        hi -= 1                              1
    return True                              1
```

Seems like $O(n)$

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

hi -= 1

Initially hi = n-1

How many bits to store hi ? $\sim \log_2 n$

If n-1 is a power of 2:

hi = 1 0 0 0 0 0 … 0

-1

hi = 0 1 1 1 1 1 … 1 $\sim \log_2 n$ steps

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

A function in Python:

# of steps

```
def twoFingers(s):
    lo = 0 ................................ 1
    hi = len(s)-1 ......................... 1
    while (lo < hi): ...................... 1
        if (s[lo] != 0 or s[hi] != 1): .... 3? 4? 5?
            return False .................. 1
        lo += 1 ........................... 1
        hi -= 1 ........................... log n ?
    return True ........................... 1
```

$O(n \log n)$?

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

if (s[lo] != 0 **or** s[hi] != 1):

Initially lo = 0, hi = n-1

Does it take n steps to go from s[0] to s[n-1] ?

Model matters

$$L = \{0^k 1^k : k \geq 0\}$$

A function in Python:

# of steps

```
def twoFingers(s):
    lo = 0 ......................................... 1
    hi = len(s)-1 .................................. 1
    while (lo < hi): ............................... 1
        if (s[lo] != 0 or s[hi] != 1): ............. n ??
            return False ........................... 1
        lo += 1 .................................... 1
        hi -= 1 .................................... log n ?
    return True .................................... 1
```

$O(n^2)$?

GREAT IDEA # 3

Computational model does matter for running time.

Model matters

Which model is the best model?

No such thing.

- Be clear about what the model is!
- Be clear about what constitutes a step in the model.

GREAT IDEA # 4

All reasonable deterministic models are polynomially equivalent.

Model matters

Which model does this correspond to ?

```
def twoFingers(s):
    lo = 0                              ........  1
    hi = len(s)-1                       ........  1
    while (lo < hi):                    ........  1
        if (s[lo] != 0 or s[hi] != 1):  ........  3? 4? 5?
            return False                ........  1
        lo += 1                         ........  1
        hi -= 1                         ........  1
    return True                         ........  1
```

$O(n)$

Model matters

The Random-Access Machine (RAM) model

Good combination of reality/simplicity.

+ , - , / , *, <, >, etc.	e.g. 245*12894	takes 1 step
memory access	e.g. A[94]	takes 1 step

Actually:

Assume arithmetic operations take 1 step **<u>IF</u>** numbers are bounded by poly(**n**).

Unless specified otherwise, we use this model.

(more on this next lecture)

Putting great ideas # 1, # 2 and # 3 together

Defining running time

With a specific **computational model** in mind:

Definition:

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

$$T_A : \mathbb{N} \to \mathbb{N}$$

defined by

worst-case

$$T_A(n) = \max_{\substack{\text{instances } I \\ \text{of size } n}} \{\# \text{ steps } A \text{ takes on } I\}$$

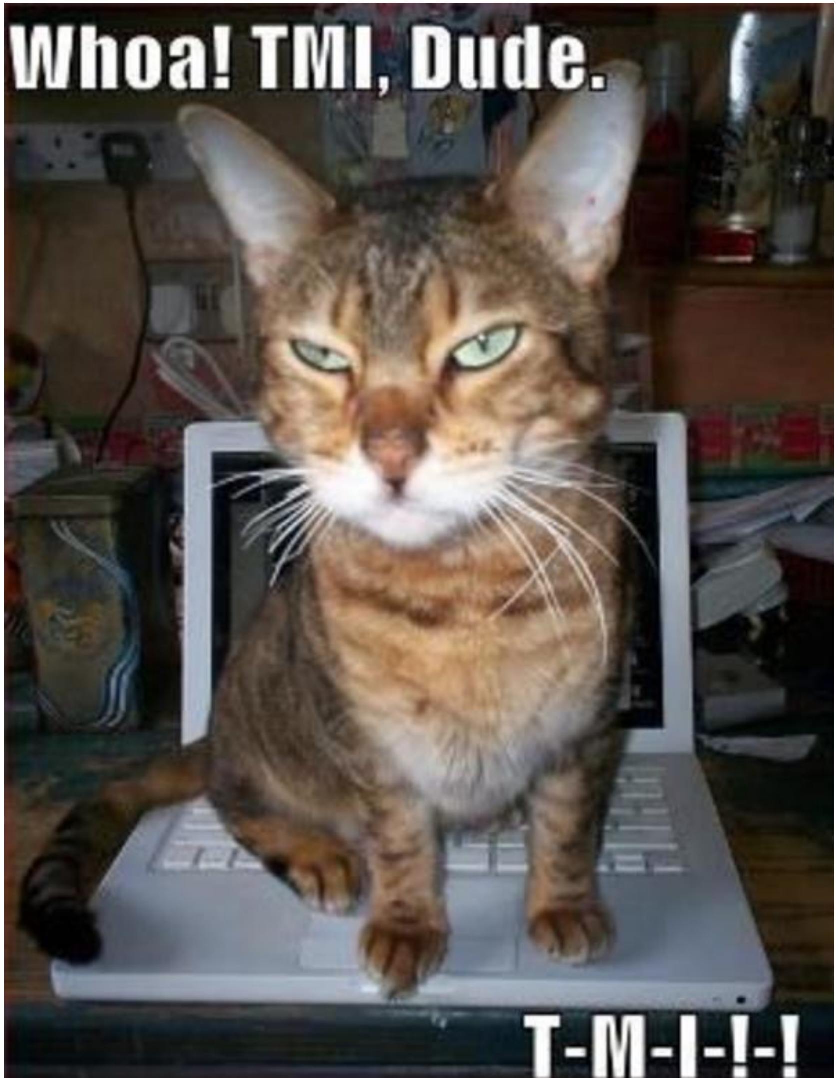
Write $T(n)$ when *A* is clear from context.

Need one more level of abstraction

There is a TM that decides PALINDROME in time

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

Analogous to
"too many significant digits".



Need one more level of abstraction

Comparing running times of two different algorithms:

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

$$T_B(n) = \frac{1}{4}n^2 + 100n^{1.5} + 1000n - 42$$

Which one is better?

GREAT IDEA/ABSTRACTION # 5

The CS way to compare functions:

$O(\cdot)$	$\Omega(\cdot)$	$\Theta(\cdot)$
$\leq$	$\geq$	$=$

Big O

Our notation for $\leq$ when comparing functions.

The right level of abstraction!

"Sweet spot"

- coarse enough to suppress details like *programming language*, *compiler*, *architecture*, ...
- sharp enough to make comparisons between different *algorithmic approaches*.

Big O

Informal: An upper bound that
ignores constant factors and ignores small n.

For $f, g : \mathbb{N}^+ \to \mathbb{R}^+$

$$f(n) = O(g(n))$$

roughly means

$$f(n) \leq g(n)$$

up to a constant factor
and ignoring small **n**.

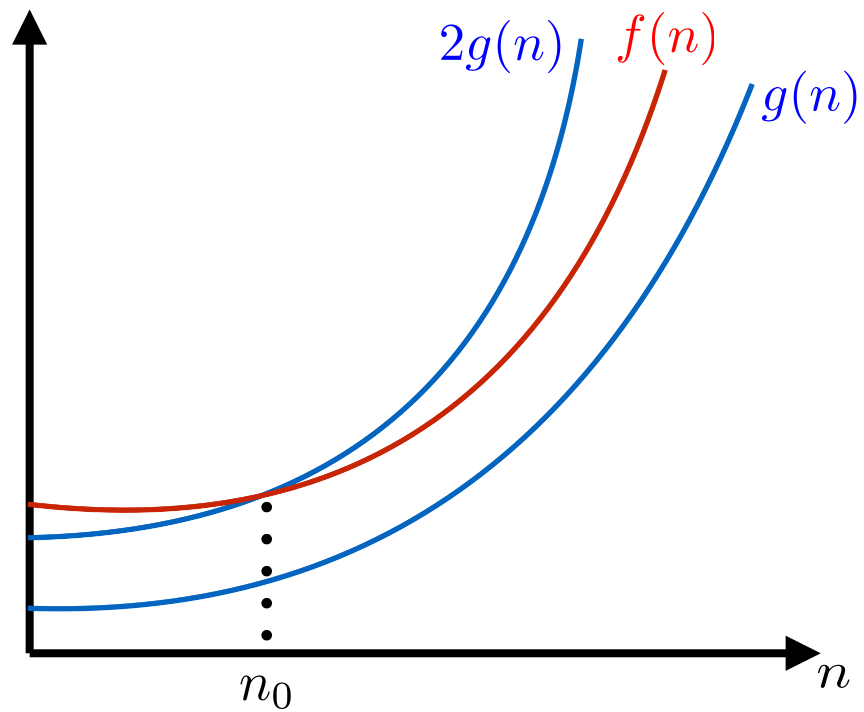
Big O

Formal Definition:

For $f, g : \mathbb{N}^+ \to \mathbb{R}^+$, we say $f(n) = O(g(n))$ if there exist constants $C,\ n_0 > 0$ such that

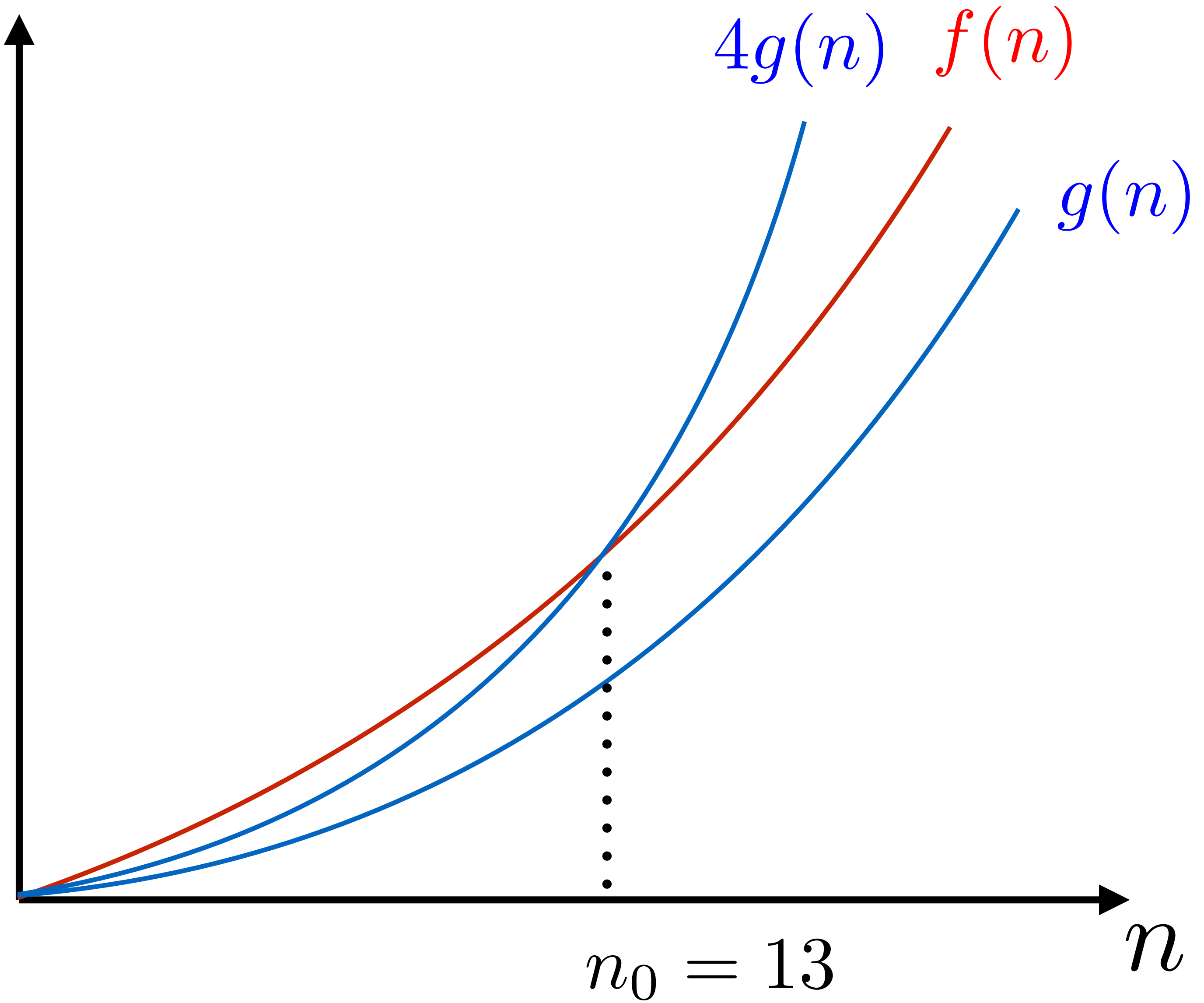
for all $n \geq n_0$, we have $f(n) \leq Cg(n)$.

(C and n_0 cannot depend on n.)



Big O

$$f(n) = 3n^2 + 10n + 30 \qquad g(n) = n^2$$



Big O

Note on notation:

People usually write: $4n^2 + 2n = O(n^2)$

Another valid notation: $4n^2 + 2n \in O(n^2)$

Big O

Common Big O classes and their names

Constant:	$O(1)$	
Logarithmic:	$O(\log n)$	
Square-root:	$O(\sqrt{n}) = O(n^{0.5})$	
Linear:	$O(n)$	
Loglinear:	$O(n \log n)$	
Quadratic:	$O(n^2)$	
Polynomial:	$O(n^k)$	(for some constant $k > 0$)
Exponential:	$O(2^{n^k})$	(for some constant $k > 0$)

n vs log n

How much smaller is log n compared to n ?

n	log n
2	1
8	3
128	7
1024	10
1,048,576	20
1,073,741,824	30
1,152,921,504,606,846,976	60

~ 1 quintillion

n vs 2^n

How much smaller is n compared to 2^n ?

2^n	n
2	1
8	3
128	7
1024	10
1,048,576	20
1,073,741,824	30
1,152,921,504,606,846,976	60

Exponential running time

If your algorithm has exponential running time
e.g. $\sim 2^n$

No hope of being practical.

Some exotic functions

1	n	2^n
$\log^* n$	$n \log n$	3^n
$\log \log n$	n^2	$n!$
$\log n$	n^3	n^n
$\sqrt{n}$	$n^{O(1)}$	2^{2^n}
$n/ \log n$	$n^{\log n}$	$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ (n times)

Big Omega

$$O(\cdot) \text{ is like } \leq$$

$$\Omega(\cdot) \text{ is like } \geq$$

$O(\cdot)$

Informal: An upper bound that
ignores constant factors and ignores small n.

$\Omega(\cdot)$

Informal: A lower bound that
ignores constant factors and ignores small n.

$$f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$$

Big Omega

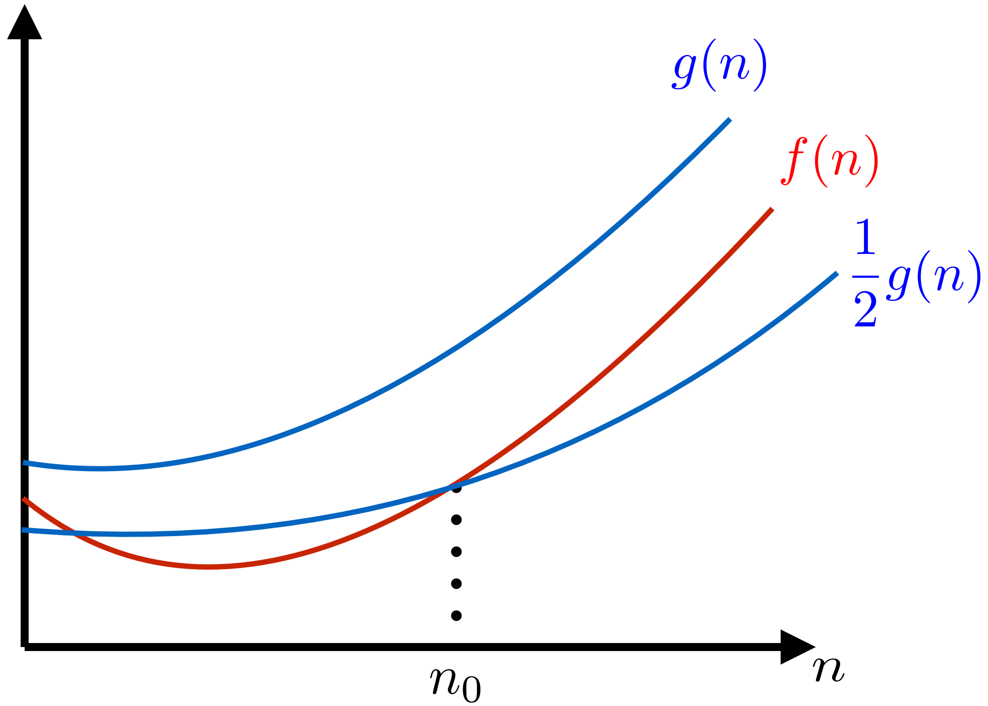
Formal Definition:

For $f, g : \mathbb{N}^+ \to \mathbb{R}^+$, we say $f(n) = \Omega(g(n))$ if

there exist constants c , $n_0 > 0$ such that

for all $n \geq n_0$, we have $f(n) \geq cg(n)$.

(c and n_0 cannot depend on n .)



Big Omega

Some Examples:

$$10^{-10}n^4 \text{ is } \Omega(n^3)$$

$$0.001n^2 - 10^{10}n - 10^{30} \text{ is } \Omega(n^2)$$

$$n^{0.0001} \text{ is } \Omega(\log n)$$

Theta

$O(\cdot)$ is like $\leq$

$\Omega(\cdot)$ is like $\geq$

$\Theta(\cdot)$ is like $=$

Theta

Formal Definition:

For $f, g : \mathbb{N}^+ \to \mathbb{R}^+$, we say $f(n) = \Theta(g(n))$ if

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Equivalently:

There exist constants c, C, n_0 such that

for all $n \geq n_0$, we have $cg(n) \leq f(n) \leq Cg(n)$.

Theta

Some Examples:

$0.001n^2 - 10^{10}n - 10^{30}$ is $\Theta(n^2)$

$1000n$ is $\Theta(n)$

$0.00001n$ is $\Theta(n)$

Putting everything together

Now we really understand what this means:

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

(which means $T_A(n) = O(n^2)$.)

Make sure you are specifying:

- the computational model

> what constitutes a *step* in the model

- the length of the input

Goals for the week

1. **What is the right way to study complexity?**
   - using the right language and level of abstraction
   - upper bounds vs lower bounds
   - polynomial time vs exponential time

2. **Appreciating the power of algorithms.**
   - analyzing some cool (recursive) algorithms

Upper bounds vs lower bounds

GREAT IDEA # 6

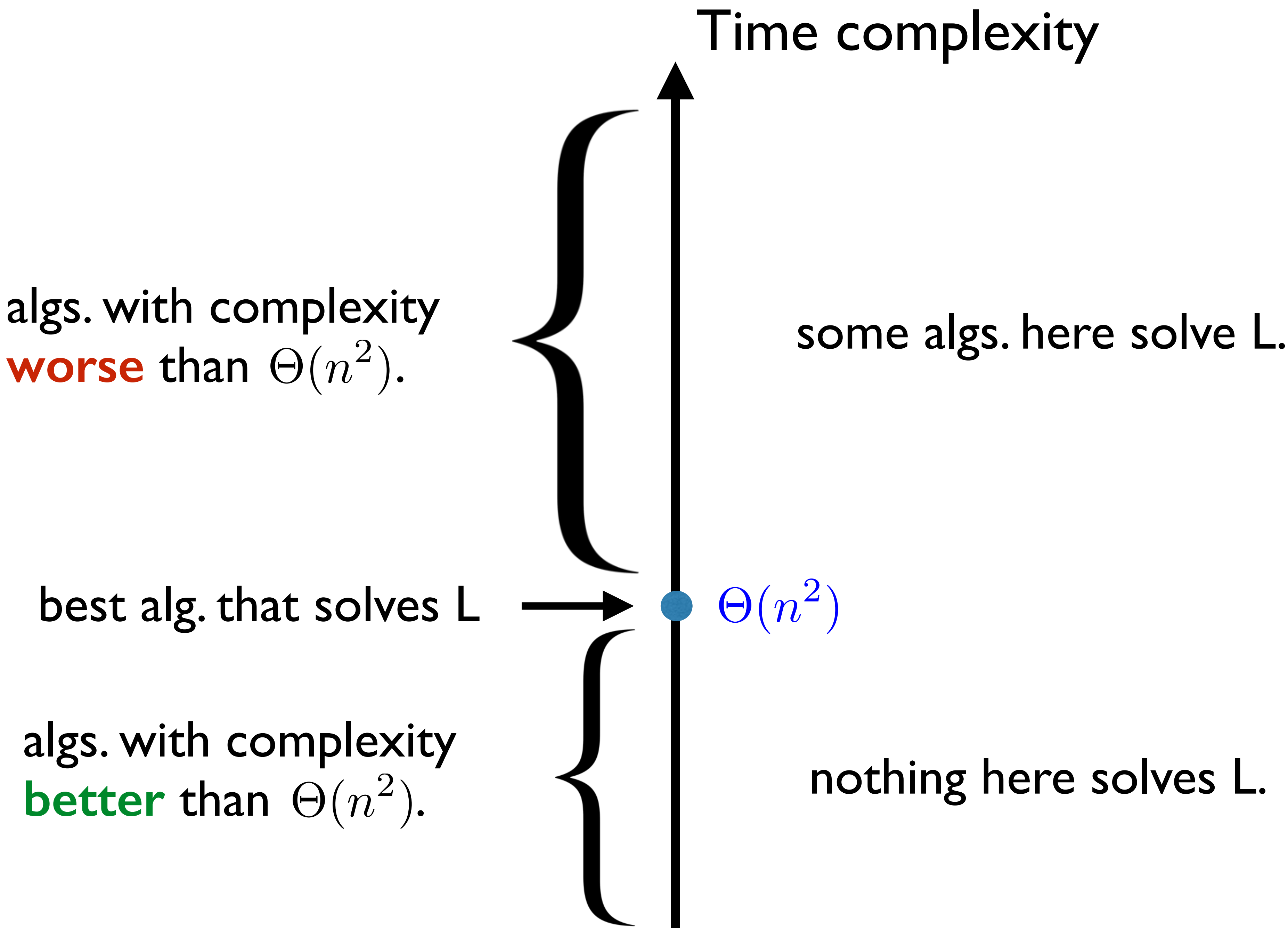
Instrinsic complexity of a problem
(upper bounds vs lower bounds)

Intrinsic complexity of a problem

The **intrinsic complexity** of a computational problem:

Asymptotic complexity of the most efficient algorithm solving it.

Intrinsic complexity



Intrinsic complexity

If you give an algorithm that solves a problem

→ upper bound on the intrinsic complexity

How do you show a lower bound on intrinsic complexity?

Argue against **all** possible algorithms that solves the problem.

The dream: Get a matching upper and lower bound.

i.e., nail down the intrinsic complexity.

Example

$$L = \{0^k 1^k : k \geq 0\}$$

```
def twoFingers(s):
    lo = 0
    hi = len(s)-1
    while (lo < hi):
        if (s[lo] != 0 or s[hi] != 1):
            return False
        lo += 1
        hi -= 1
    return True
```

In the RAM model:

$$O(n)$$

Could there be
a faster algorithm?

e.g. $O(n/\log n)$

Example

$$L = \{0^k 1^k : k \geq 0\}$$

Fact: Any algorithm that decides L must use $\geq n$ steps.

Proof: Proof is by contradiction.

Suppose there is an algorithm A that decides *L* in $< n$ steps.

Consider the input I = $0^k 1^k$ (I is a YES instance)

When A runs on input I, there must be some index j such that A never reads $I[j]$.

Let I' be the same as I, but with *j*'th coordinate reversed. (I' is a NO instance)

When A runs on I', it has the same behavior as it does on I.

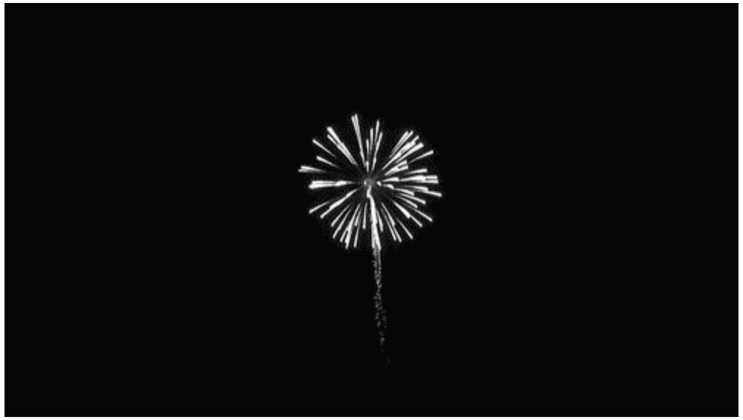
But then A cannot be a decider for *L*. *Contradiction.* $\square$

Example

This shows the intrinsic complexity of *L* is $\Omega(n)$.

But we also know the intrinsic complexity of *L* is $O(n)$.

The dream achieved. Intrinsic complexity is $\Theta(n)$.



Goals for the week

1. **What is the right way to study complexity?**

   - using the right language and level of abstraction

   - upper bounds vs lower bounds

   - polynomial time vs exponential time

2. **Appreciating the power of algorithms.**

   - analyzing some cool (recursive) algorithms

Polynomial time vs Exponential time

GREAT IDEA # 7

There is something magical about polynomial time.

What is efficient in theory and in practice ?

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?

What is efficient in theory and in practice ?

In theory:

Polynomial time	Efficient.
Otherwise	Not efficient.

What is efficient in theory and in practice ?

- Poly-time is **<u>not</u>** meant to mean "efficient in practice".

- Poly-time: extraordinarily better than brute force search.

- Poly-time: mathematical insight into problem's structure.

- Robust to notion of what is an
  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

What is efficient in theory and in practice ?

Brute-Force Algorithm: Exponential time

what we care about most in 15-251

usually the "magic" happens here

Algorithmic Breakthrough: Polynomial time

what we care about more in 15-451

Blood, sweat, and tears: Linear time

What is efficient in theory and in practice ?

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