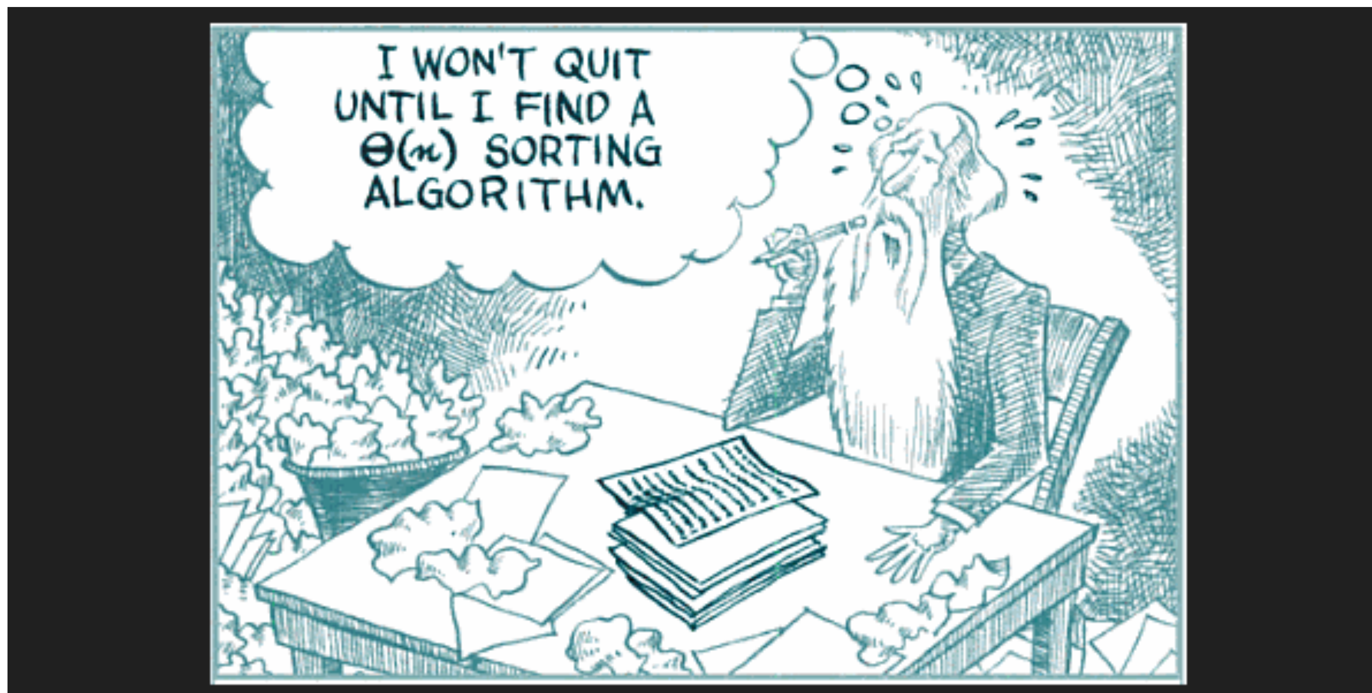
15-251
Great Theoretical Ideas in Computer Science

Lecture 7:
Introduction to Computational Complexity



September 20th, 2016

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."

- Combinatorial games

  Fun topic. Good practice writing proofs.

- Formalization of computation/algorithm

  Deterministic Finite Automata

  Turing Machines

What have we done so far?

> The study of computation

Computability

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

But many interesting problems are decidable!

What is next?

> The study of computation

Computability

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?

⋮

What is next?

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EUCLID EVENTS

Millennium Problems

1 million dollar question

(or maybe 6 million dollar question)

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.

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.

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.

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

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 running time of recursive functions

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

We have to be careful

Model matters

Size matters

Value matters

Model matters

Church-Turing Thesis says:

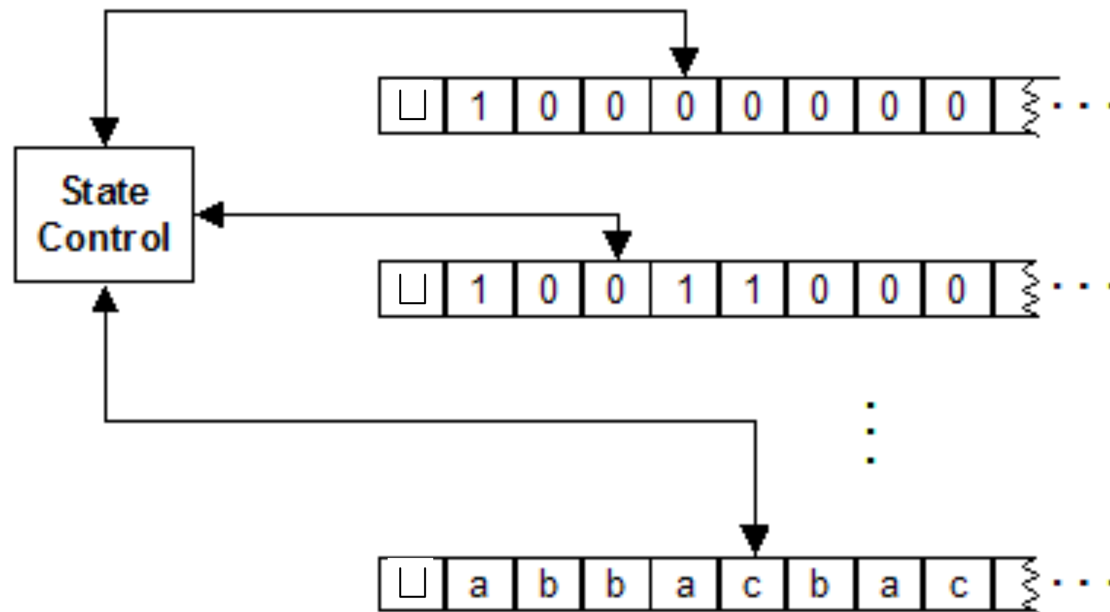
With respect to decidability, model does **not** matter.

The same is not true with respect to complexity!

Model matters

Multitape Turing Machine

Ordinary TM with multiple tapes, each with its own head.



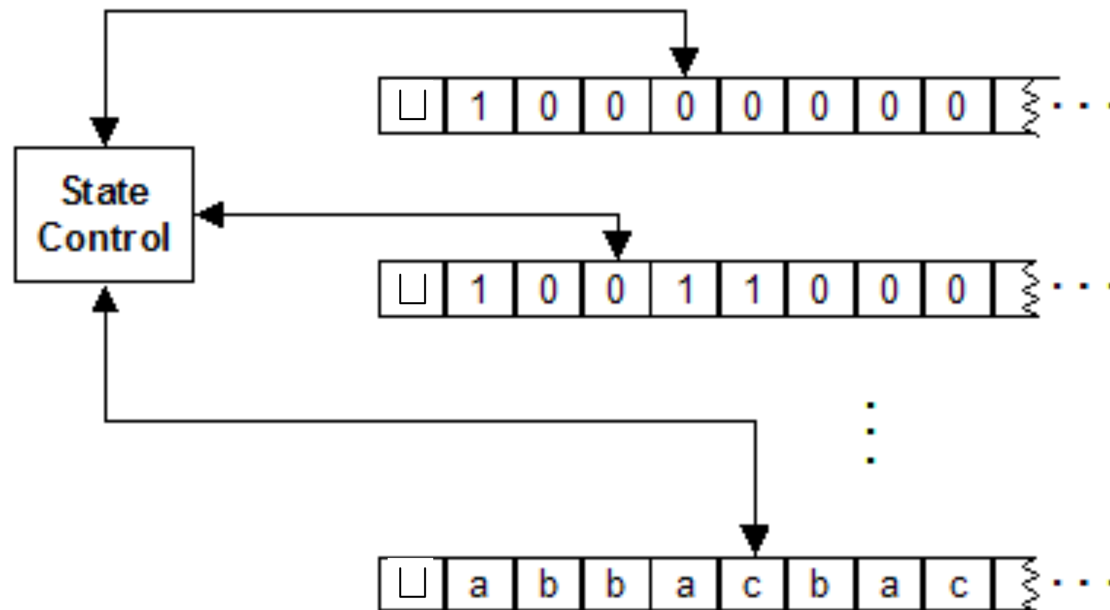
Multiple Tape/Head Turing Machines

Number of tapes is fixed (cannot grow with input size).

Model matters

Multitape Turing Machine

Ordinary TM with multiple tapes, each with its own head.



Multiple Tape/Head Turing Machines

Is it more powerful?

Every multitape TM has an equivalent single tape TM.

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

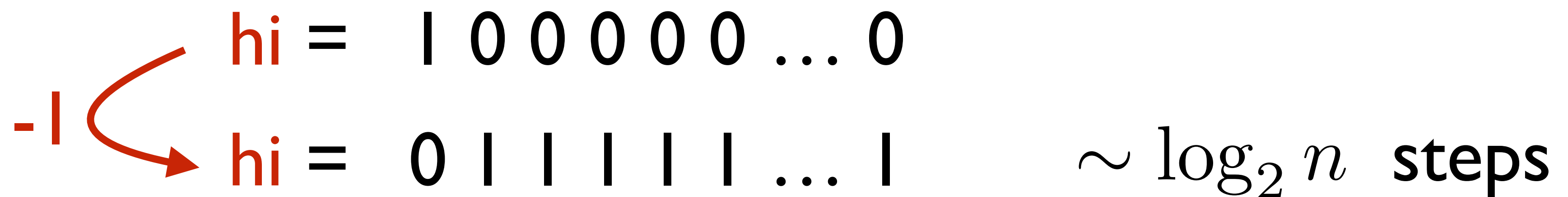
$$L = \{0^k 1^k : k \geq 0\}$$

hi -= 1

Initially hi = n-1

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

If n-1 is a power of 2:



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
```

$O(n^2)$?

Model matters

SO

Number of steps (running time)
depends on the particular model you choose.

Which one is the best model?

No such thing.

1. Be clear about what the model is!
2. 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	take 1 step
memory access	e.g. A[94]	takes 1 step

Technically:

We'll assume arithmetic operations take 1 step if the numbers are bounded by a polynomial in n.

Unless specified otherwise, we will use this model.

We have to be careful

Model matters

Size matters

Value matters

Size matters

Sorting bazillion numbers will take more time than sorting 2 numbers.

Running time of an algorithm depends on input length.

n = input length

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

sometimes: explicitly defined to be something else.

Running time of an algorithm is a function of n.

We have to be careful

Model matters

Size matters

Value matters

Value matters

Not all inputs are created equal!

Among all inputs of length **n**:

- some might make your algorithm take 2 steps
- some might make your algorithm take bazillion steps.

Better to be safe than sorry.

Shoot for worst-case guarantees.

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.
- Matches our worst-case notion of an alg. solving a prob.
- Hard to define "typical" instances.
- Random instances are often not representative.
- Often much easier to analyze.

Defining running time

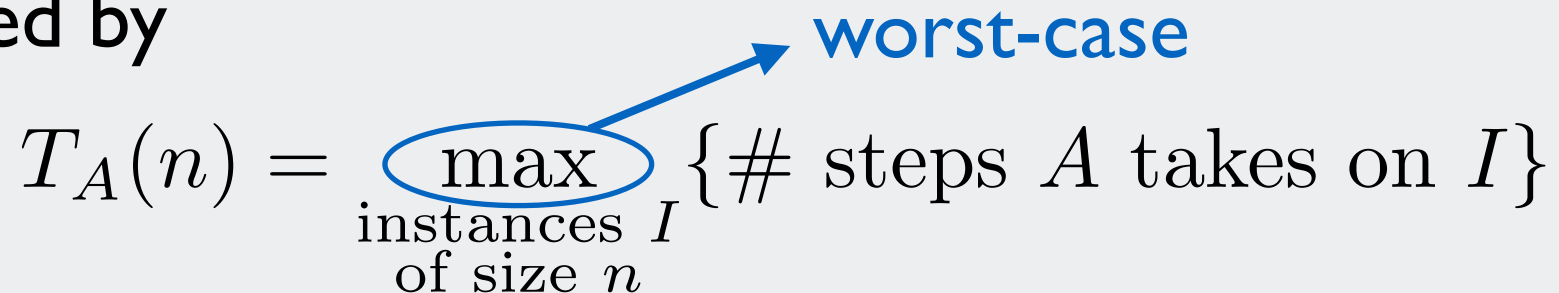
With a specific computational model in mind:

Definition:

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

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

defined by



We drop the subscript *A*, and write $T(n)$ when *A* is clear.

Need one more level of abstraction

OK, so running time of an algorithm is just a function.

For example, we could say, 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".

For example, we really don't care about constant factors.

Need one more level of abstraction

OK, so running time of an algorithm is just a function.

How do I compare the 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?

The CS way to compare functions:

$$O(\cdot) \qquad \Omega(\cdot) \qquad \Theta(\cdot)$$

$$\leq \qquad \geq \qquad =$$

Big O

Our notation for $\leq$ when comparing functions.

The right level of abstraction!

"Sweet spot"

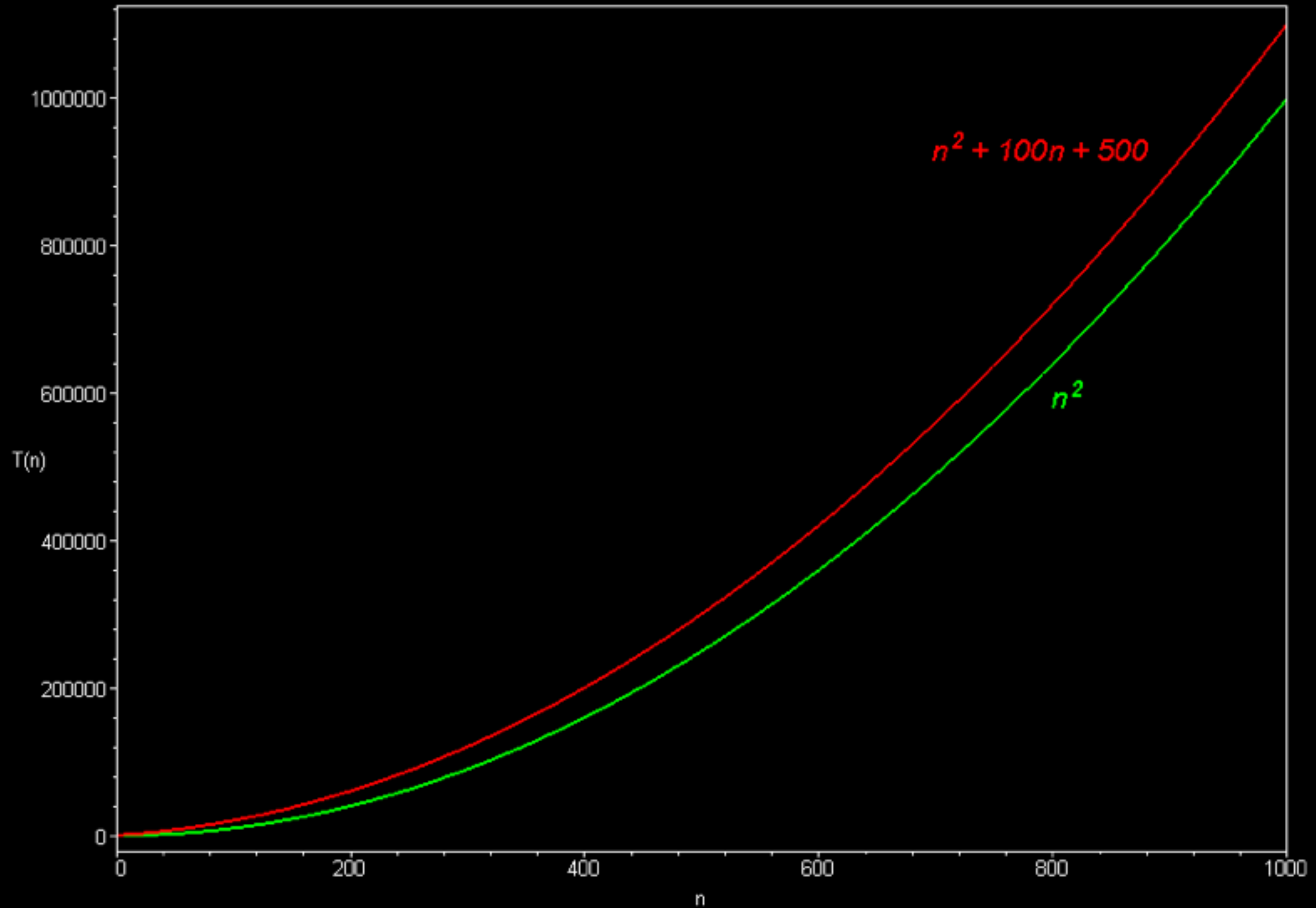
- coarse enough to suppress details like programming language, compiler, architecture,…
- sharp enough to make comparisons between different algorithmic approaches.

Big O

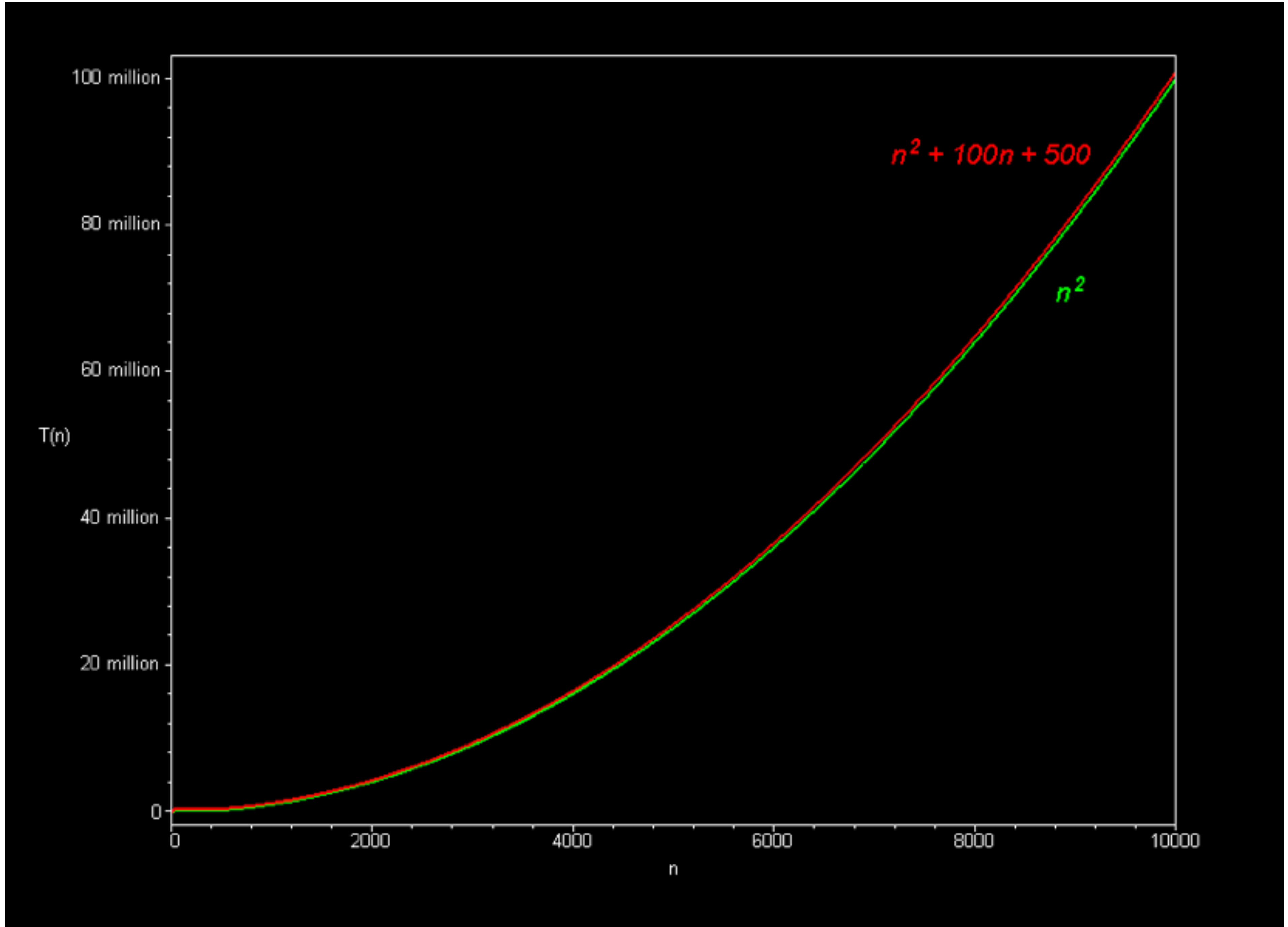
Informal: An upper bound that ignores constant factors and ignores small n.

Note: Suppressing constant factors means suppressing lower order additive terms.

Big O



Big O



Big O

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.

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

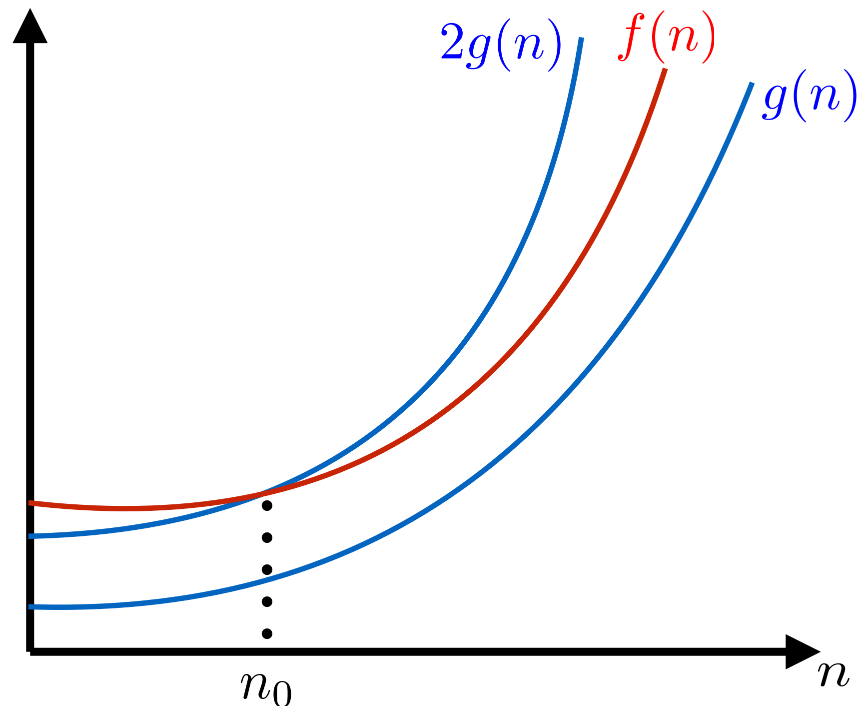
Formal Definition:

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

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.)

Example:

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

Take $C = 4,\quad n_0 = 13$.

When $n \geq 13,\; 10n + 30 \leq 10n + 3n = 13n \leq n^2$.

So $f(n) = 3n^2 + 10n + 30 \leq 3n^2 + n^2 = 4n^2 = 4g(n)$

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.)

Example:

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

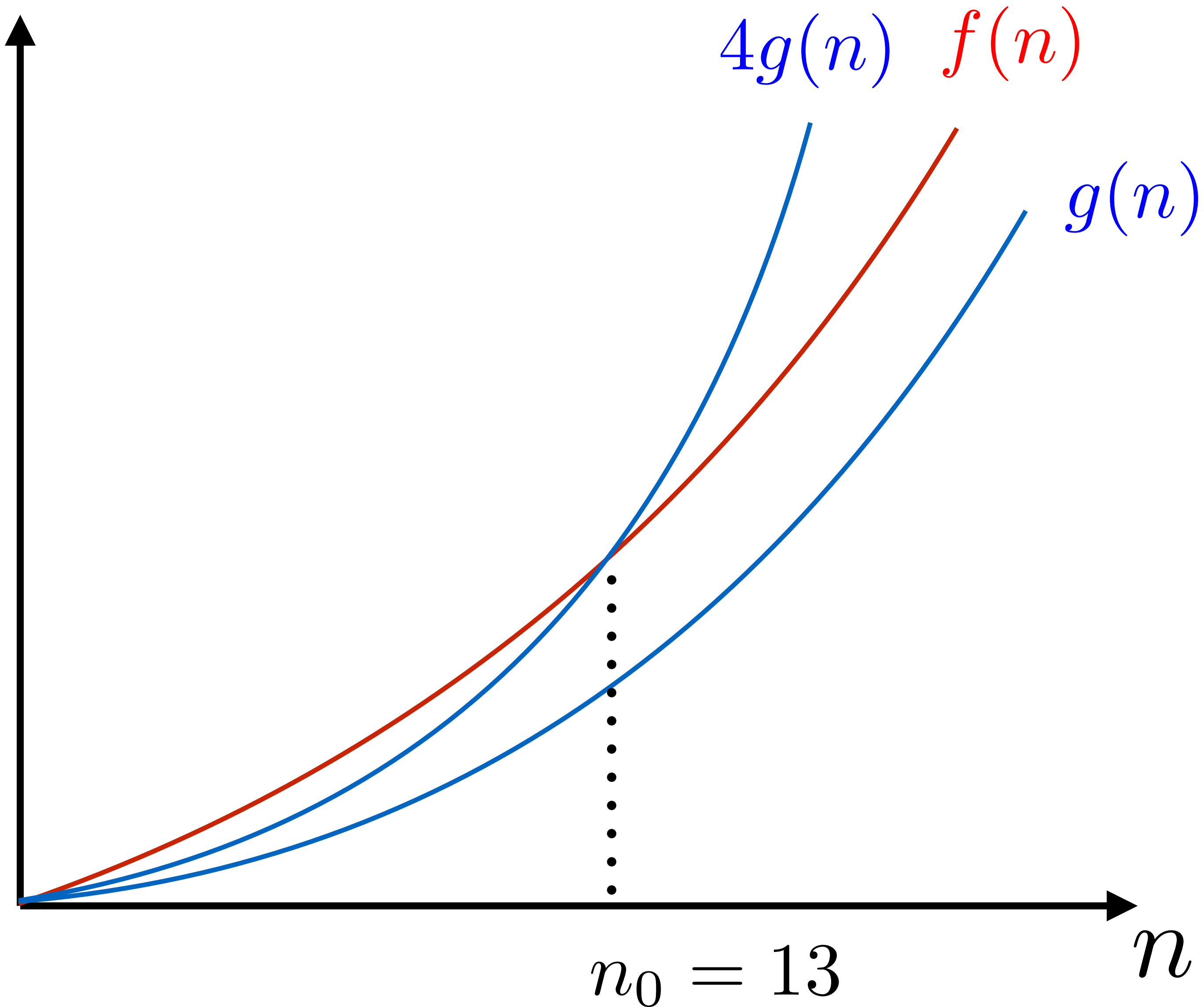
Take $C = 43,\quad n_0 = 1$. When $n \geq 1$,

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

$$= 43n^2 = 43g(n)$$

Big O

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



Big O

Proving f(n) is O(g(n)) is like a game:

You pick constants C, n_0

Adversary picks $n \geq n_0$

You win if $f(n) \leq Cg(n)$

You need to make sure you always win.

Big O

$1000n$ is $O(n)$

$0.0000001n$ is $O(n)$

$0.1n^2 + 10^{20}n + 10^{10000}$ is $O(n^2)$

n is $O(2^n)$

$0.0000001n^2$ <u>is not</u> $O(n)$

$n \log n$ <u>is not</u> $O(n)$

$\log_9 n$ is $O(\log n)$

$$\log_b(n) = \frac{\log_k(n)}{\log_k(b)}$$

constant

10^{10} is $O(1)$

<u>Note on notation:</u>

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

Better notation would be $4n^2 + 2n \in O(n^2)$

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.

Differences between functions

$$\log n <<< \sqrt{n} << n < n \log n << n^2 << n^3 <<< 2^n <<< 3^n$$

Some exotic functions

1

$\log^* n$

$\log \log n$

$\log n$

$\sqrt{n}$

$n / \log n$

n

$n \log n$

n^2

n^3

$n^{O(1)}$

$n^{\log n}$

2^n

3^n

$n!$

n^n

2^{2^n}

$2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$ (n times)

<u>Fastest algorithm for multiplication:</u>

$$n \cdot (\log n) \cdot 2^{O(\log^* n)}$$

Poll

Select all that apply.

$\log(n!)$ is:

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(2^n)$

Beats me

Big Omega

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

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

$O(\cdot)$

Informal: An <u>upper bound</u> that ignores constant factors and ignores small n.

$\Omega(\cdot)$

Informal: A <u>lower bound</u> that ignores constant factors and ignores small n.

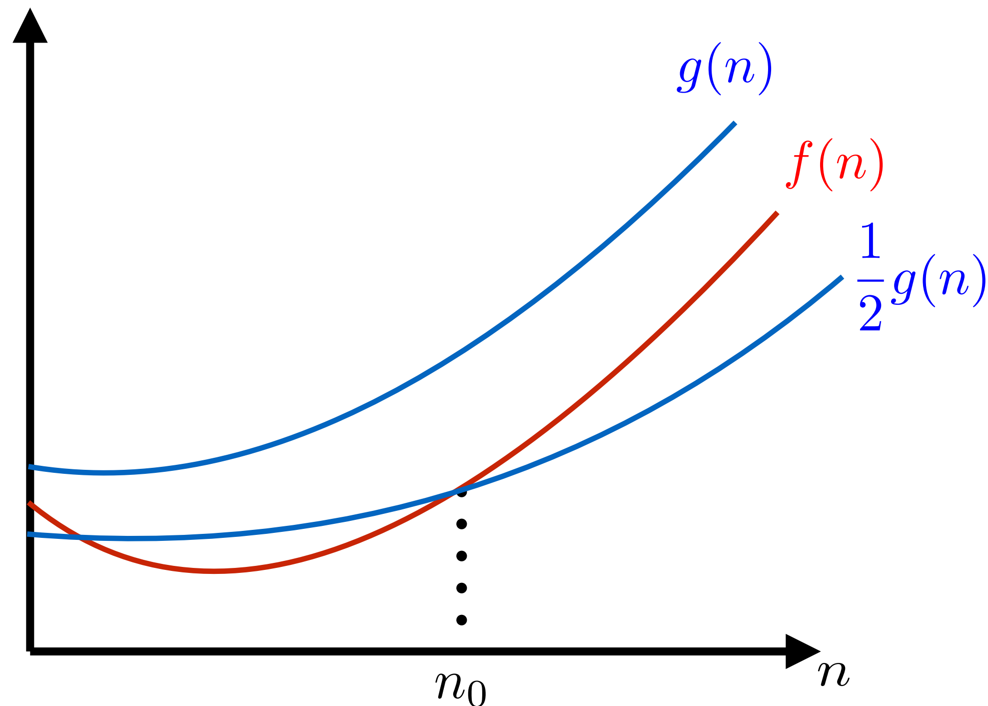
Big Omega

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:

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

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

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

$n^{1.0001}$ is $\Omega(n \log n)$

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

Alright!

Suppose we agree on:

- **the computational model**
  - > what operations are allowed
  - > what is the complexity of each operation
- **the length of the input**

We like to say things like:

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

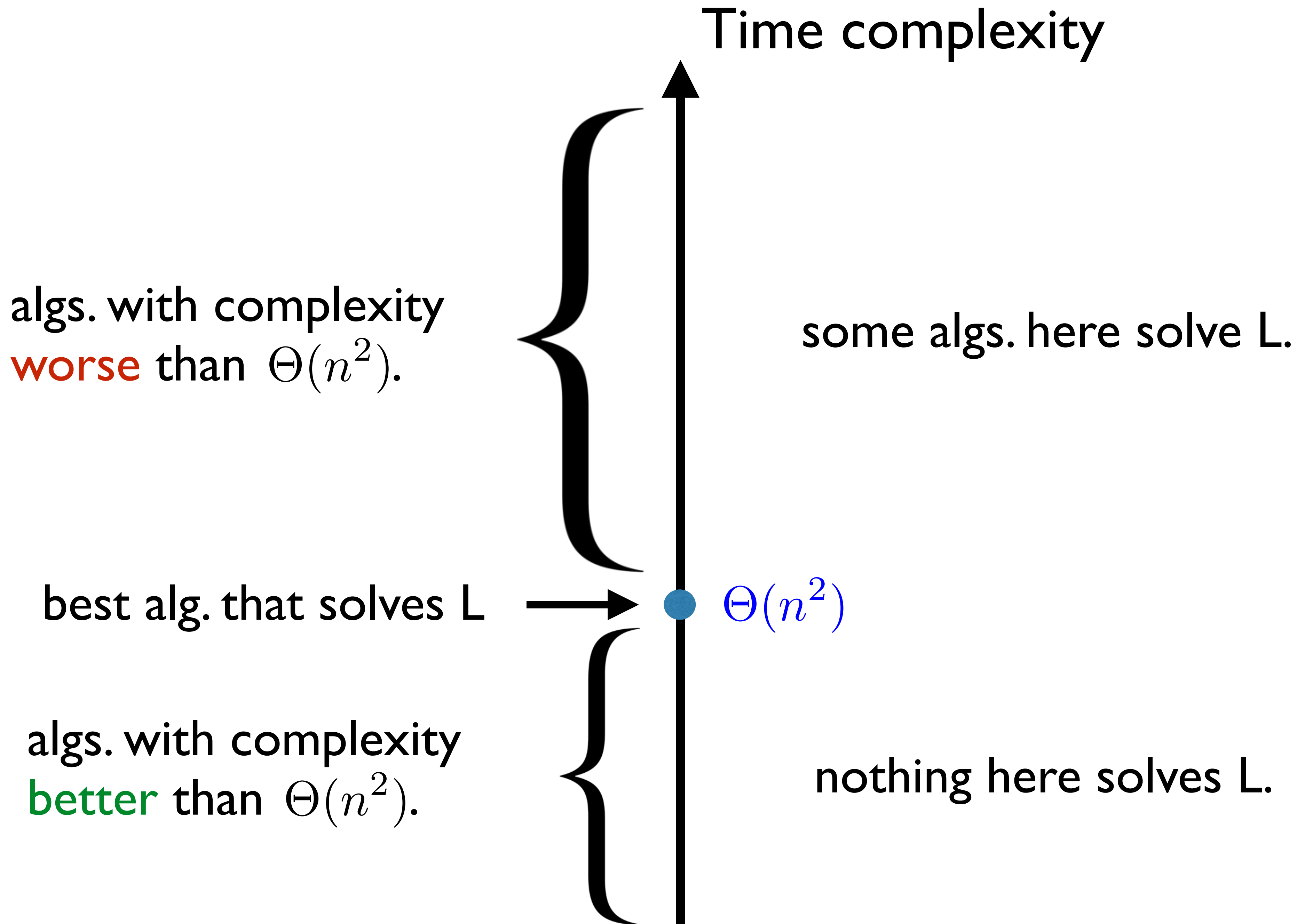
(which means $T_A(n) = O(n^2)$.)

Upper bounds vs lower bounds

Intrinsic complexity of a computational problem

The **intrinsic complexity** of a problem is the asymptotic complexity of the most efficient algorithm solving it.

Intrinsic complexity



Intrinsic complexity

If you give an algorithm that solves a problem

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 $\Theta(n)$.

Polynomial time vs Exponential 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.

- Poly-time is not meant to mean "efficient in practice"

- It means "You have done something extraordinarily better than brute force (exhaustive) search."

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

What is efficient in theory and in practice ?

In theory:

Polynomial time	Efficient.
Otherwise	Not efficient.

- 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 ?

In theory:

Polynomial time	Efficient.
Otherwise	Not efficient.

- Big exponents don't really arise.

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

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

What is efficient in theory and in practice ?

In theory:

Polynomial time	Efficient.
Otherwise	Not efficient.

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

Strong Church Turing Thesis

Church Turing Thesis

Church-Turing Thesis:

The intuitive notion of "computable" is captured by functions computable by a Turing Machine.

Physical Church-Turing Thesis:

Any computational problem that can be solved by a physical device, can be solved by a Turing Machine.

Strong Church-Turing Thesis:

The intuitive notion of "efficiently computable" is captured by functions efficiently computable by a TM.

Strong Church Turing Thesis

Experience suggests it is true for all *deterministic* models.

First main challenger in 1970s:

Randomized computation.

In light of research from 1980s, we believe
SCCT holds even with randomized computation.

Second main challenger in 1980s:

Quantum computation.

In light of research from 1990s, we believe
SCCT is **not** true!

Challenge all ideas!