# [320] Complexity + Big O

Meenakshi Syamkumar

# Outline

Performance and Complexity

What is a step?

Counting Executed Steps

Big O: for functions/curves

Big O: for algorithms

# Performance vs. Complexity

**Things that affect performance (total time to run):**

- ????

# Performance vs. Complexity

**Things that affect performance (total time to run):**

- speed of the computer (CPU, etc)

- speed of Python (quality+efficiency of interpretation)

- **algorithm**: strategy for solving the problem

- **input size:** how much data do we have?

# Performance vs. Complexity

**Things that affect performance (total time to run):**

- speed of the computer (CPU, etc)

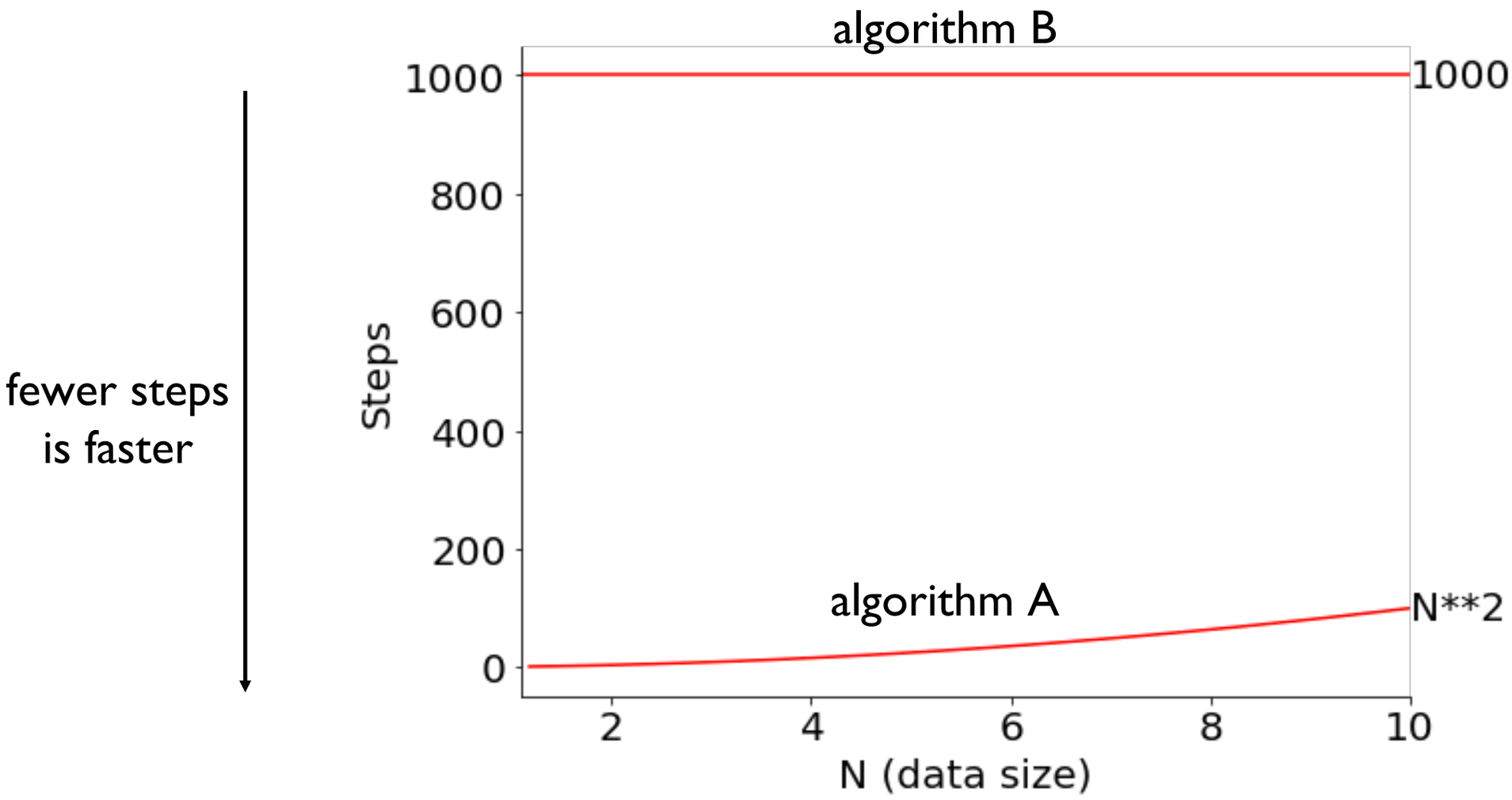
- speed of Python (quality+efficiency of interpretation)

- **algorithm:** strategy for solving the problem

- **input size:** how much data do we have?

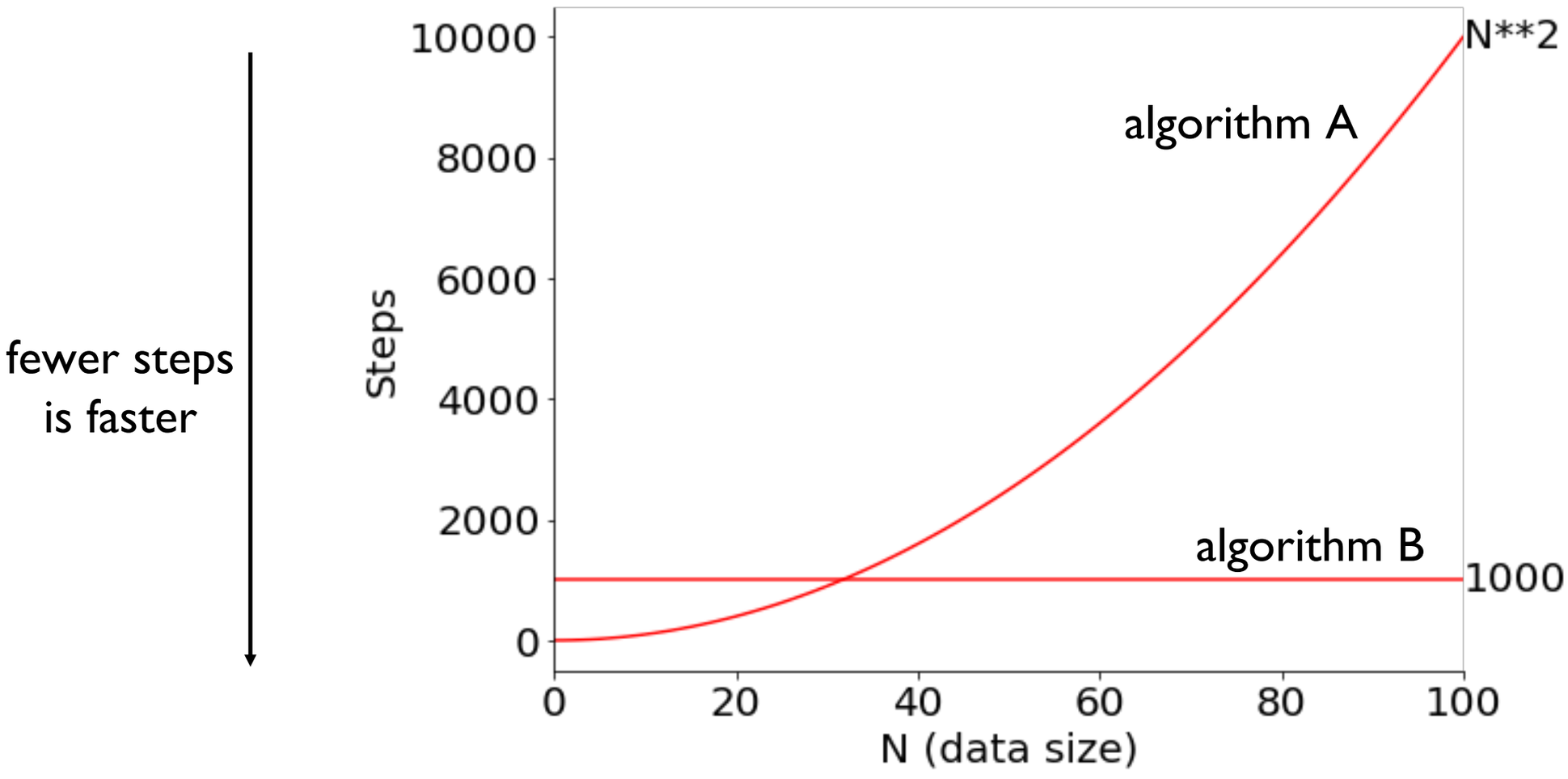
**complexity analysis:** how many steps must the algorithm perform, as a function of input size?

# Which algorithm is better?



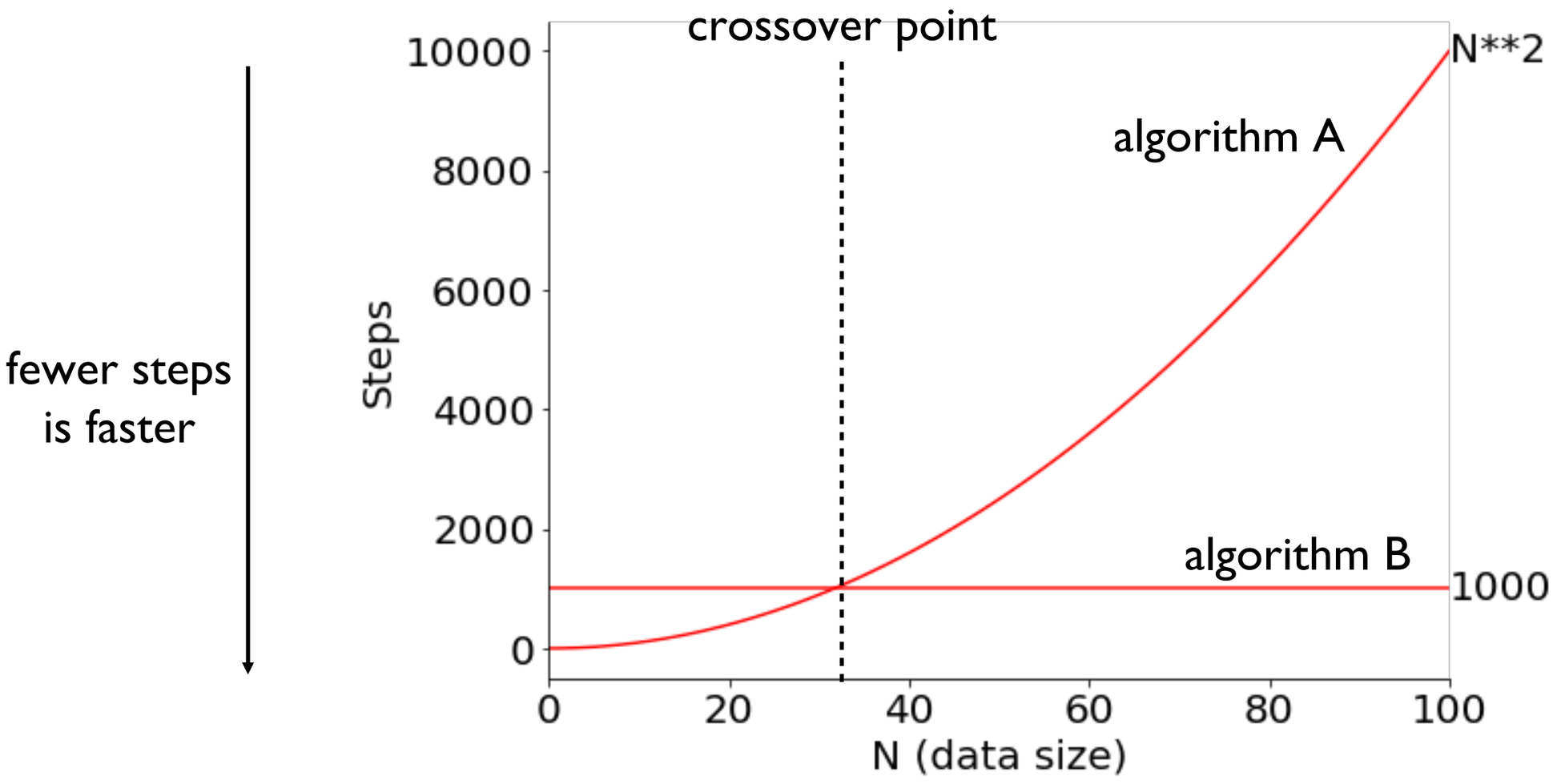

Do you prefer A or B?

# Which algorithm is better?



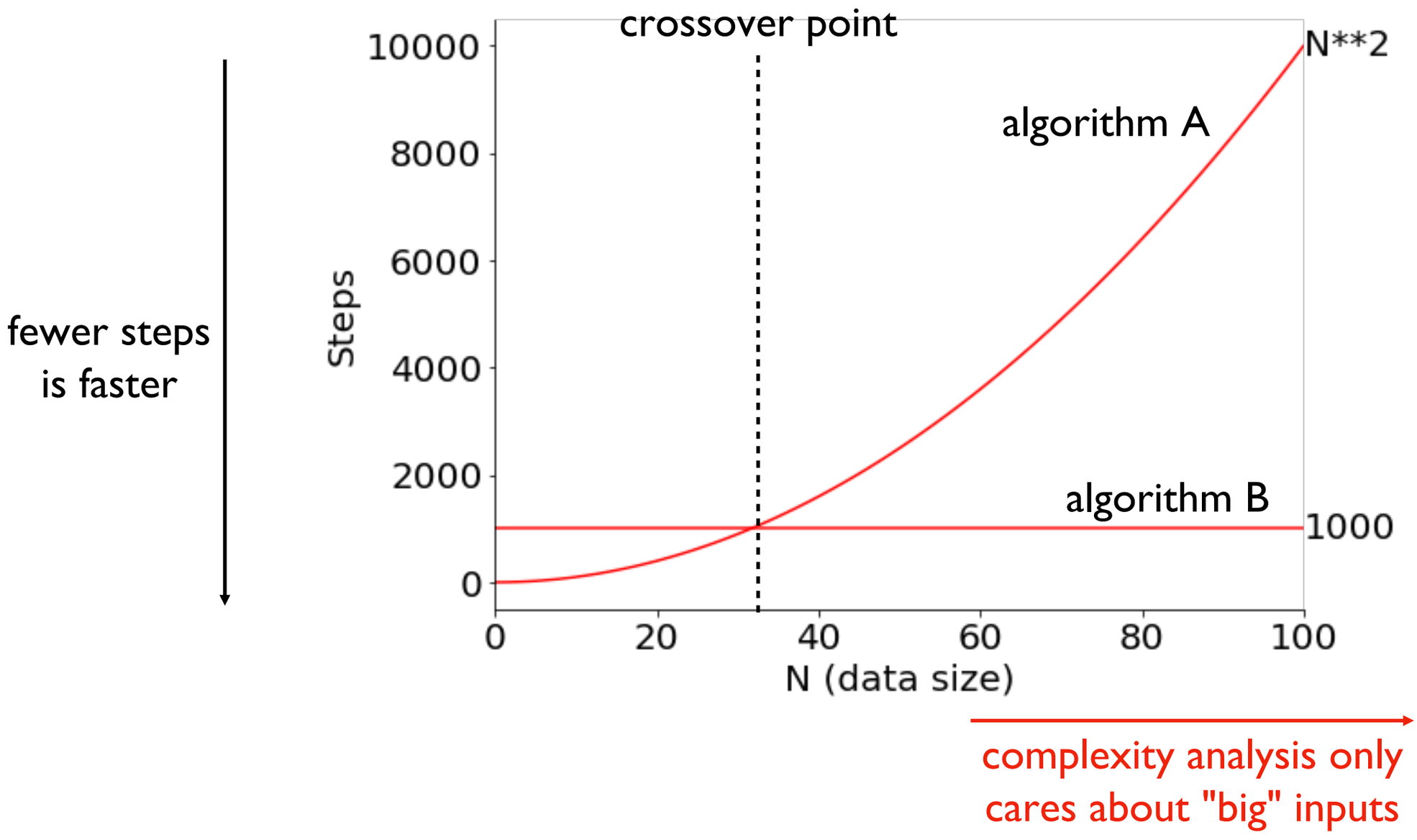

Do you prefer A or B?

# Which algorithm is better?



# Which algorithm is better?

you might still reasonably
care about this portion!




What is the asymptotic behavior of the function?

# Performance vs. Complexity

**Things that affect performance (total time to run):**

- speed of the computer (CPU, etc)

- speed of Python (quality+efficiency of interpretation)

- **algorithm:** strategy for solving the problem

- **input size:** how much data do we have?

what is this?

**complexity analysis:** how many **steps** must the algorithm perform, as a function of input size?

# Outline

Performance and Complexity

What is a step?

Counting Executed Steps

Big O: for functions/curves

Big O: for algorithms

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

input size is length of this list

```
input_nums = [2, 3, ...]
```

```
STEP  odd_count = 0
STEP  odd_sum = 0
STEP  for num in input_nums:
STEP      if num % 2 == 1:
STEP          odd_count += 1
STEP          odd_sum += num
STEP  odd_avg = odd_sum
STEP  odd_avg /= odd_count
```

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
```

STEP
```
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum
odd_avg /= odd_count
```

also a valid breakdown into steps

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
```

STEP
```
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum / odd_count
```

One line can do a lot, so no reason to
have lines and steps be equivalent

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
```

STEP
```
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum / odd_count
```

Sometimes a single line is not a single step:
`found = X in L`

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum / odd_count
```

???

# What is a step?

"bounded" doesn't mean "fixed"

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum / odd_count
```

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP

```
odd_count = 0
odd_sum = 0
```

STEP
(whole loop execution,
not one pass through)

```
for num in input_nums:
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP

```
odd_avg = odd_sum / odd_count
```

???

is this a valid way to identify steps?

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

not a "step", because exec time depends on input size

STEP

```
odd_count = 0
odd_sum = 0
```

STEP
(whole loop execution, not one pass through)

```
for num in input_nums:
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP

```
odd_avg = odd_sum / odd_count
```

# What is a step?

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

not a "step", because exec time depends on input size

STEP

```
odd_count = 0
odd_sum = 0
```

STEP
(whole loop execution, not one pass through)

```
for num in input_nums:
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP

```
odd_avg = odd_sum / odd_count
```

Note! A loop that iterates a bounded number of times (not proportional to input size) COULD be a single step.

# Outline

Performance and Complexity

What is a step?

Counting Executed Steps

Big O: for functions/curves

Big O: for algorithms

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

STEP
```
odd_count = 0
odd_sum = 0
```

STEP
```
for num in input_nums:
```

STEP
```
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```

STEP
```
odd_avg = odd_sum / odd_count
```

How many total steps will **execute** if
`len(input_nums) == 10`?

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]

odd_count = 0
odd_sum = 0
for num in input_nums:
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
odd_avg = odd_sum / odd_count
```

1 STEP

+ 11 STEP

+ 10 STEP

+ 1 STEP

= 23 steps

For **N** elements, there will be **2*N+3** steps

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]

?    STEP  odd_count = 0
?    STEP  odd_sum = 0
?    STEP  for num in input_nums:
?    STEP      if num % 2 == 1:
?    STEP          odd_count += 1
?    STEP          odd_sum += num
?    STEP  odd_avg = odd_sum
?    STEP  odd_avg /= odd_count
```

How many total steps will **execute** if
`len(input_nums) == 10`?

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

| | | |
|---|---|---|
| 1 | STEP | `odd_count = 0` |
| 1 | STEP | `odd_sum = 0` |
| 11 | STEP | `for num in input_nums:` |
| 10 | STEP | `    if num % 2 == 1:` |
| 0 to 10 | STEP | `        odd_count += 1` |
| 0 to 10 | STEP | `        odd_sum += num` |
| 1 | STEP | `odd_avg = odd_sum` |
| 1 | STEP | `odd_avg /= odd_count` |

How many total steps will **execute** if
`len(input_nums) == 10`?

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

| | | |
|---|---|---|
| 1 | STEP | `odd_count = 0` |
| + 1 | STEP | `odd_sum = 0` |
| + 11 | STEP | `for num in input_nums:` |
| + 10 | STEP | `    if num % 2 == 1:` |
| + 0 to 10 | STEP | `        odd_count += 1` |
| + 0 to 10 | STEP | `        odd_sum += num` |
| + 1 | STEP | `odd_avg = odd_sum` |
| + 1 | STEP | `odd_avg /= odd_count` |

For **N** elements, there will be between
**2*N+5** and **4*N+5** steps

# Counting Executed Steps

A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

```
input_nums = [2, 3, ...]
```

| | | |
|---|---|---|
| 1 | STEP | `odd_count = 0` |
| + 1 | STEP | `odd_sum = 0` |
| + 11 | STEP | `for num in input_nums:` |
| + 10 | STEP | `    if num % 2 == 1:` |
| + 0 to 10 | STEP | `        odd_count += 1` |
| + 0 to 10 | STEP | `        odd_sum += num` |
| + 1 | STEP | `odd_avg = odd_sum` |
| + 1 | STEP | `odd_avg /= odd_count` |

For **N** elements, there will be between
~~2*N+5~~ and **4*N+5** steps

usually we care about
the worst case

# Counting Executed Steps

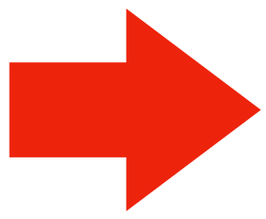
A **step** is any unit of work with bounded execution time
(it doesn't keep getting slower with growing input size)

2*N+3 **OR** 4*N+5

answer 1 answer 2

Answer 2 is never bigger than 2 times answer 1.
Answer 1 is never bigger than answer 2.

**Important:** we might not identify steps the same, but our execution counts can at most differ by a constant factor!



**can we broadly (but rigorously) categorize based on this?**

# Outline

Performance and Complexity

What is a step?

Counting Executed Steps

Big O: for functions/curves

Big O: for algorithms

# How fast?

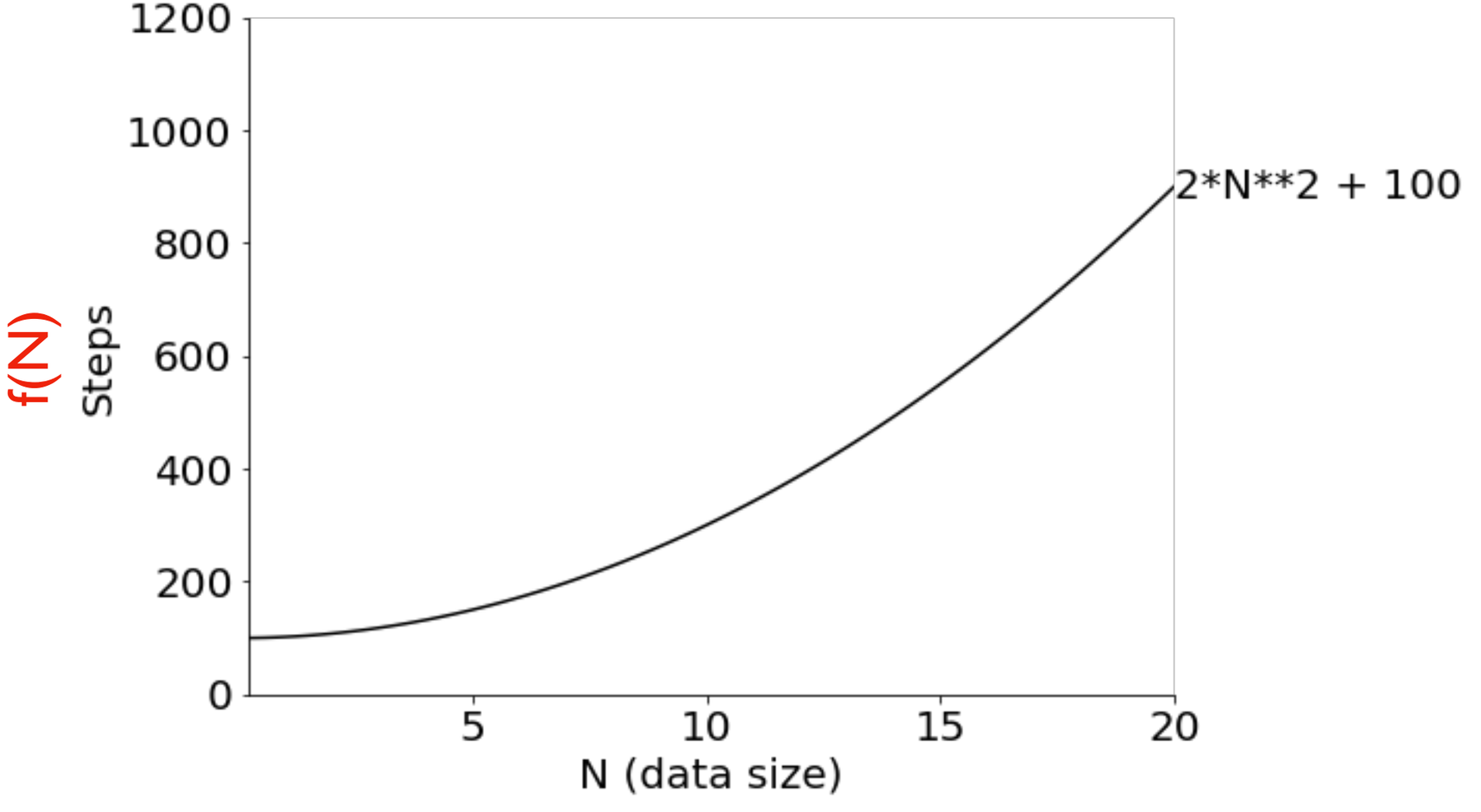
## Documentation

- https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares-complexity
- https://scikit-learn.org/stable/modules/tree.html#complexity

# Big O Notation ("O" is for "order of growth")

**Goal:** categorize functions (and algorithms) by how fast they **grow**

- **do not care** about scale
- **do not care** about small inputs
- **care** about shape of the curve
- **strategy:** find some multiple of a general function that is an upper bound

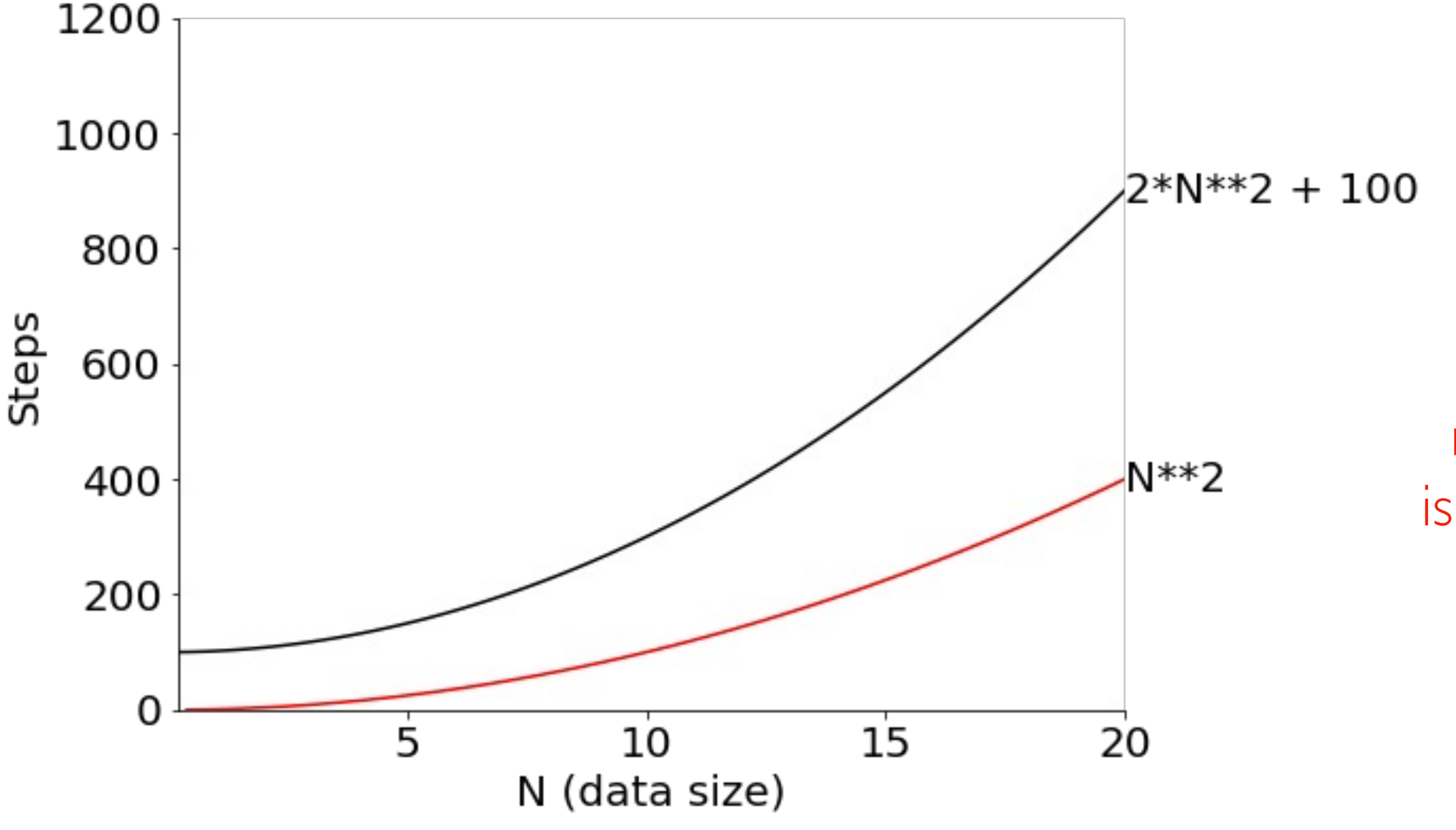



$f(N) == 2N^2 + 100$
is an $O(N^2)$ function

# Big O Notation ("O" is for "order of growth")

**Goal:** categorize functions (and algorithms) by how fast they **grow**

- **do not care** about scale
- **do not care** about small inputs
- **care** about shape of the curve
- **strategy:** find some multiple of a general function that is an upper bound



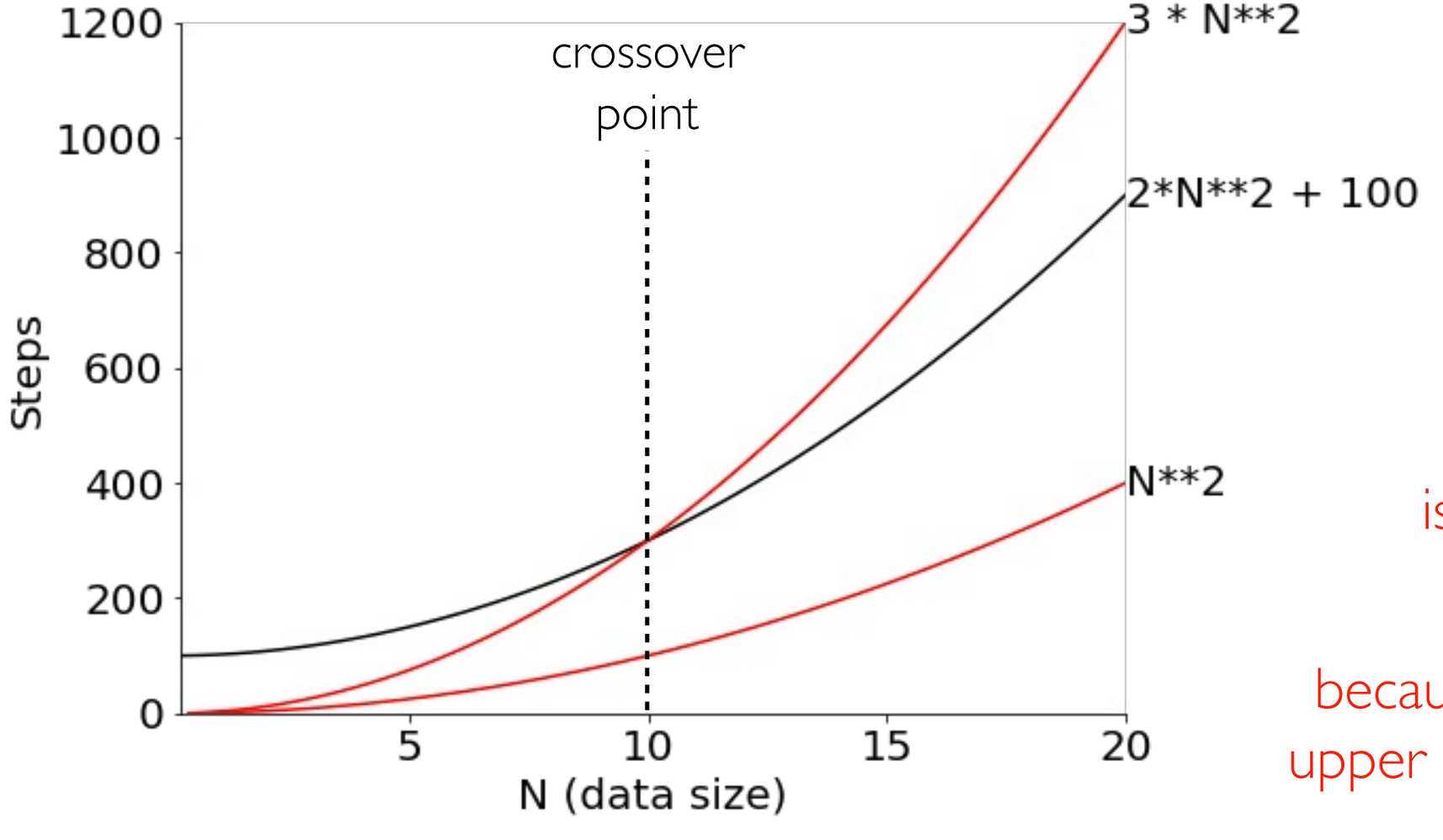

$f(N) == 2N^2 + 100$
is an $O(N^2)$ function

not because $N^2$
is an upper bound

# Big O Notation ("O" is for "order of growth")

**Goal:** categorize functions (and algorithms) by how fast they **grow**

- **do not care** about scale
- **do not care** about small inputs
- **care** about shape of the curve
- **strategy:** find some multiple of a general function that is an upper bound




$f(N) == 2N^2 + 100$
is an $O(N^2)$ function

not because $N^2$
is an upper bound

because **some multiple** is an upper bound after **some point**

# Defining Big O

care about shape of the curve

do not care about small inputs

do not care about scale

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

# Defining Big O

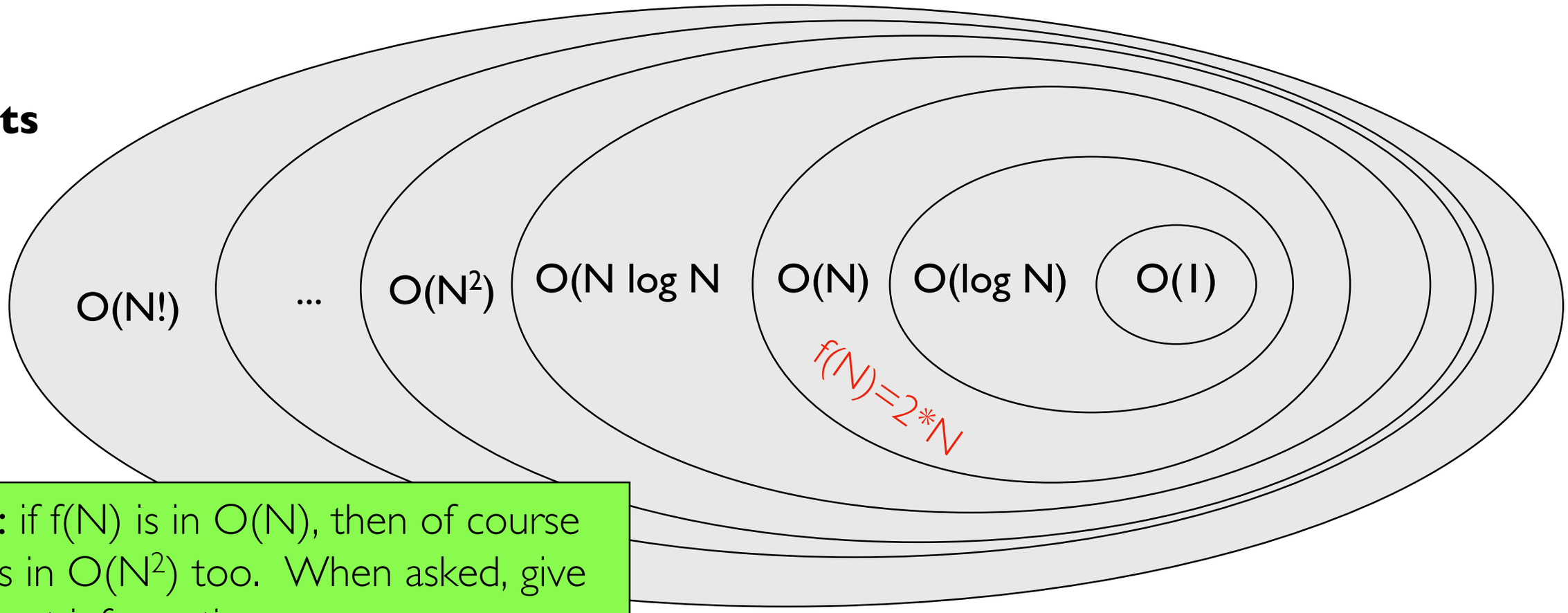
care about shape of the curve | do not care about small inputs | do not care about scale

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

---

**Sets**




**Note:** if f(N) is in O(N), then of course f(N) is in $O(N^2)$ too. When asked, give the most informative answer.

# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

---

which ones are true?

$f(N) = 2N \in O(N)$

$f(N) = 100N \in O(N^2)$

$f(N) = N^2 \in O(1000000N)$

# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

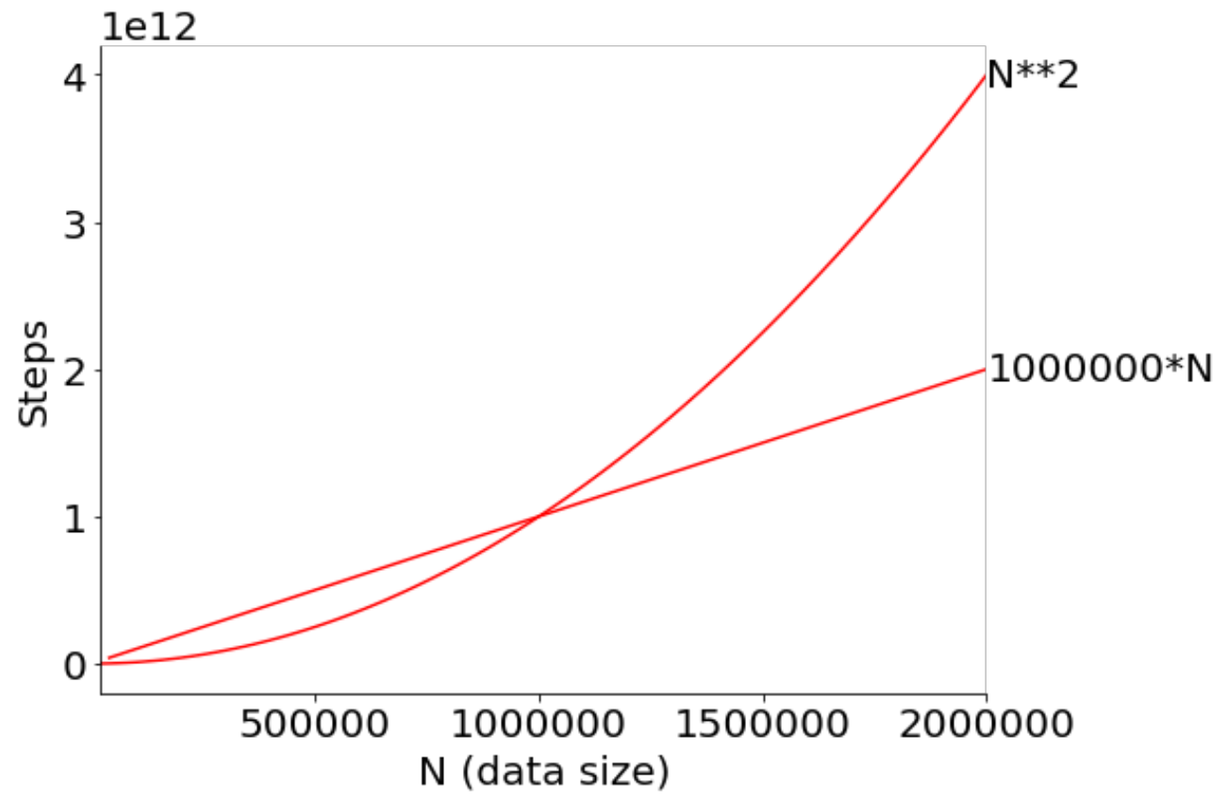
---

which ones are true?

$f(N) = 2N \in O(N)$

$f(N) = 100N \in O(N^2)$

~~$f(N) = N^2 \in O(1000000N)$~~



# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

---

# shortcuts

$O(3N^5 + N^4 + 8N^3 + 3N^2 + N + 5)$

keep leading term (if finite number)

$O(3N^5)$

drop coefficients

$O(N^5)$

# Outline

Performance and Complexity

What is a step?

Counting Executed Steps

Big O: for functions/curves

Big O: for algorithms

# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

---

We'll let **f(N)** be the number of steps that some **Algorithm A** needs to perform for input size **N**.

When we say **Algorithm A** $\in O(g(N))$,
we mean that $f(N) \in O(g(N))$

# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

---

STEP
```
odd_count = 0
odd_sum = 0
```
STEP
```
for num in input_nums:
```
STEP
```
    if num % 2 == 1:
        odd_count += 1
        odd_sum += num
```
STEP
```
odd_avg = odd_sum / odd_count
```

For N elements, there will be 2*N+3 steps

$2*N+3 \leq 3 * N$

[for big N values]

therefore

this code is O(N)

# Defining Big O

**If** $f(N) \leq C * g(N)$ *for large N values and some fixed constant C*

**Then** $f(N) \in O(g(N))$

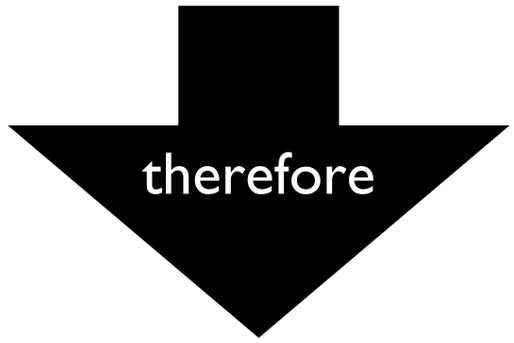
---

```
STEP  odd_count = 0
STEP  odd_sum = 0
STEP  for num in input_nums:
STEP      if num % 2 == 1:
STEP          odd_count += 1
STEP          odd_sum += num
STEP  odd_avg = odd_sum
STEP  odd_avg /= odd_count
```

For N elements, there will be between 2*N+5 and 4*N+5 steps

$4*N+5 \leq 5 * N$

[for big N values]




this code is O(N)

# Analysis of Algorithms: Key Ideas

**complexity**: relationship between input size and steps executed

**step**: an operation of bounded cost (doesn't scale with input size)

**asymptotic analysis**: we only care about very large N values for complexity (for example, assume a big list)

**worst-case**: we'll usually assume the worst arrangement of data because it's harder to do an average case analysis (for example, assume search target at the end of a list)

big O: if $f(N) \leq C * g(N)$ for large N values and some fixed constant **C**, then $f(N) \in O(g(N))$