[320] Complexity + Big O

Tyler Caraza-Harter

Video Survey Results

78 people filled the survey

87% said they would use it to review
(5 said they would skip lecture -- please don't!)

68% said "*if I don't understand something during in-person lecture, I would prefer to review the video later than ask a question in person*"

Plan: usually record videos for review for now
(no guarantees if there are technical difficulties)

But! If people aren't asking many questions during lecture, I'll stop recording videos.

Review

The situation where git cannot auto-merge is called a ______________

What is the missing step?

1. nano file.txt
2. ????
3. git commit -m "I changed file.txt"
4. git push

What type does check_output return?

How can you use time.time() to measure an operation that is much faster than calling time.time()?

Complexity and Big O: Reading

Required: Think Python, Appendix B

http://www.greenteapress.com/thinkpython/html/thinkpython022.html (skip B.4)

Optional [math heavy]:

http://web.mit.edu/16.070/www/lecture/big_o.pdf

Complexity

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?

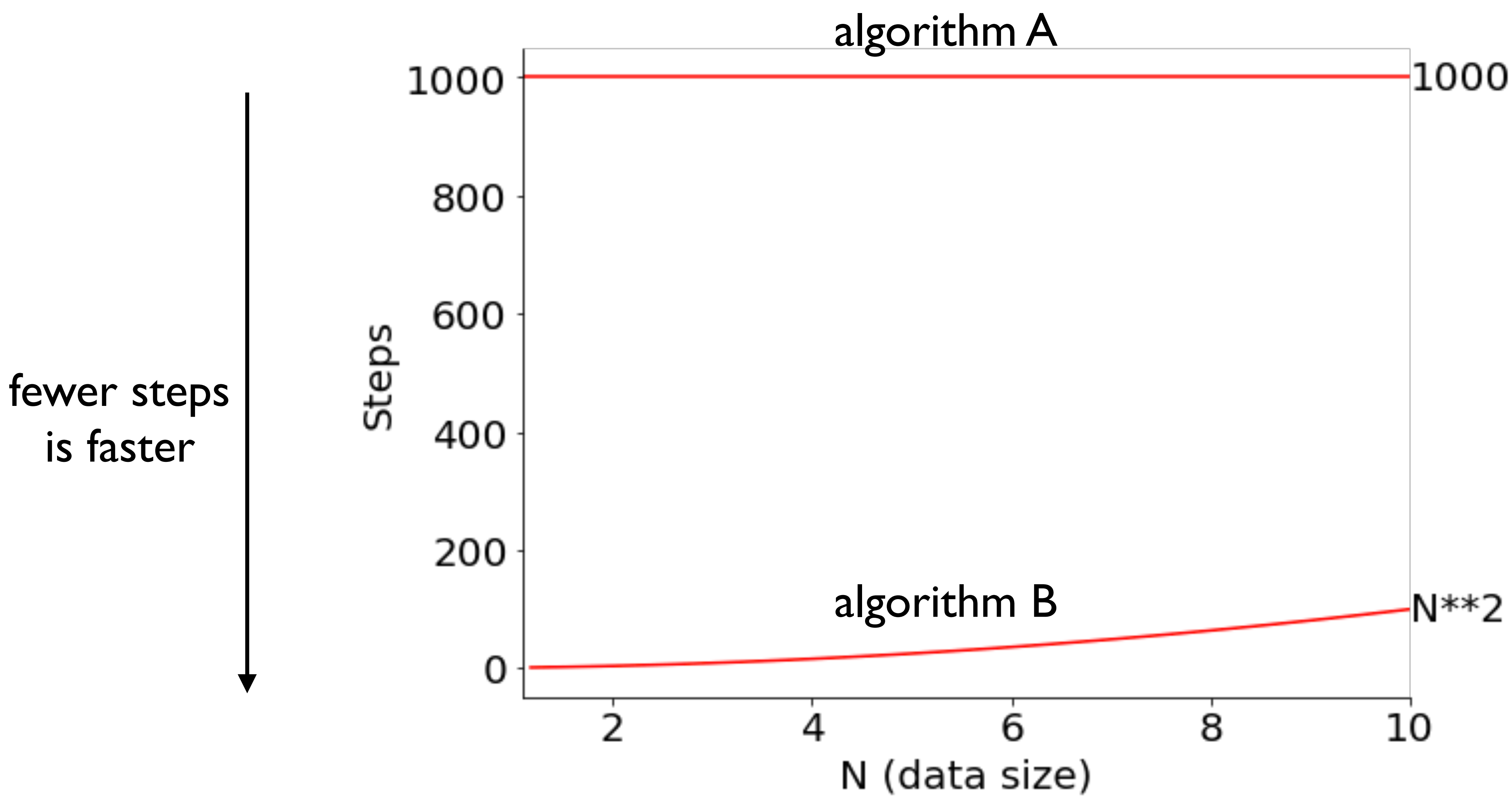
Performance vs. Complexity

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?

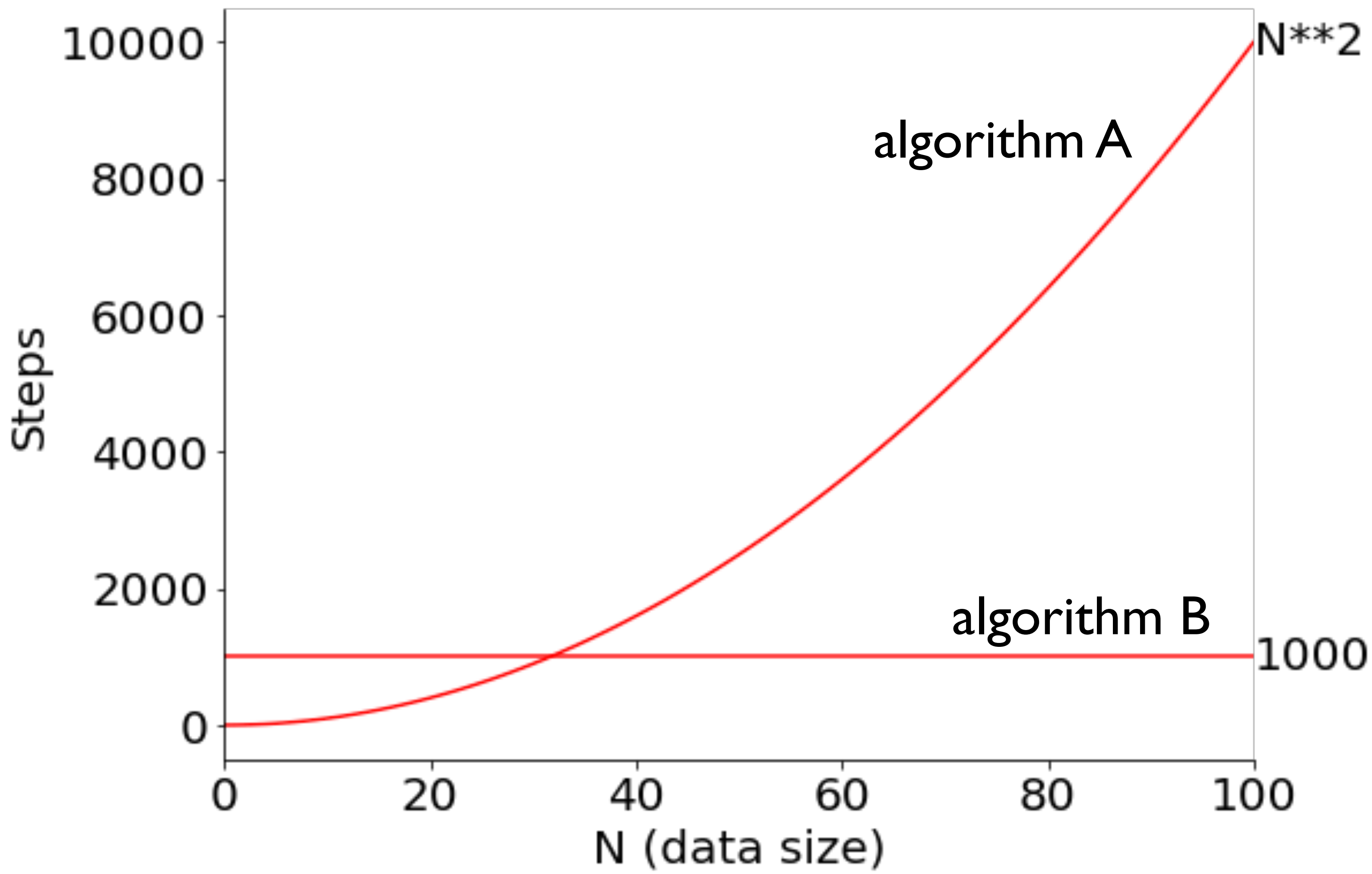
Which algorithm is better?



Do you prefer A or B?

Which algorithm is better?

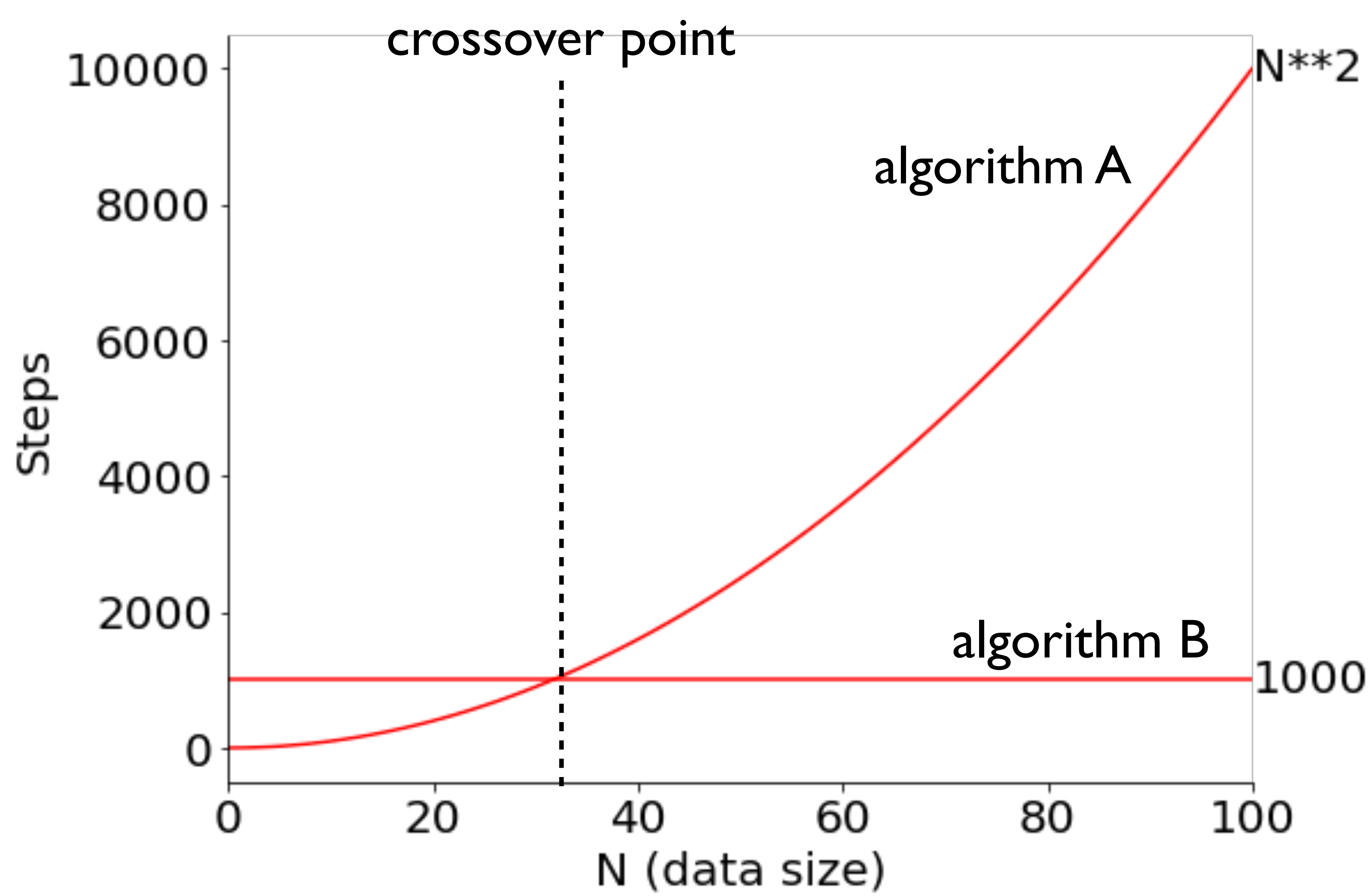
fewer steps
is faster



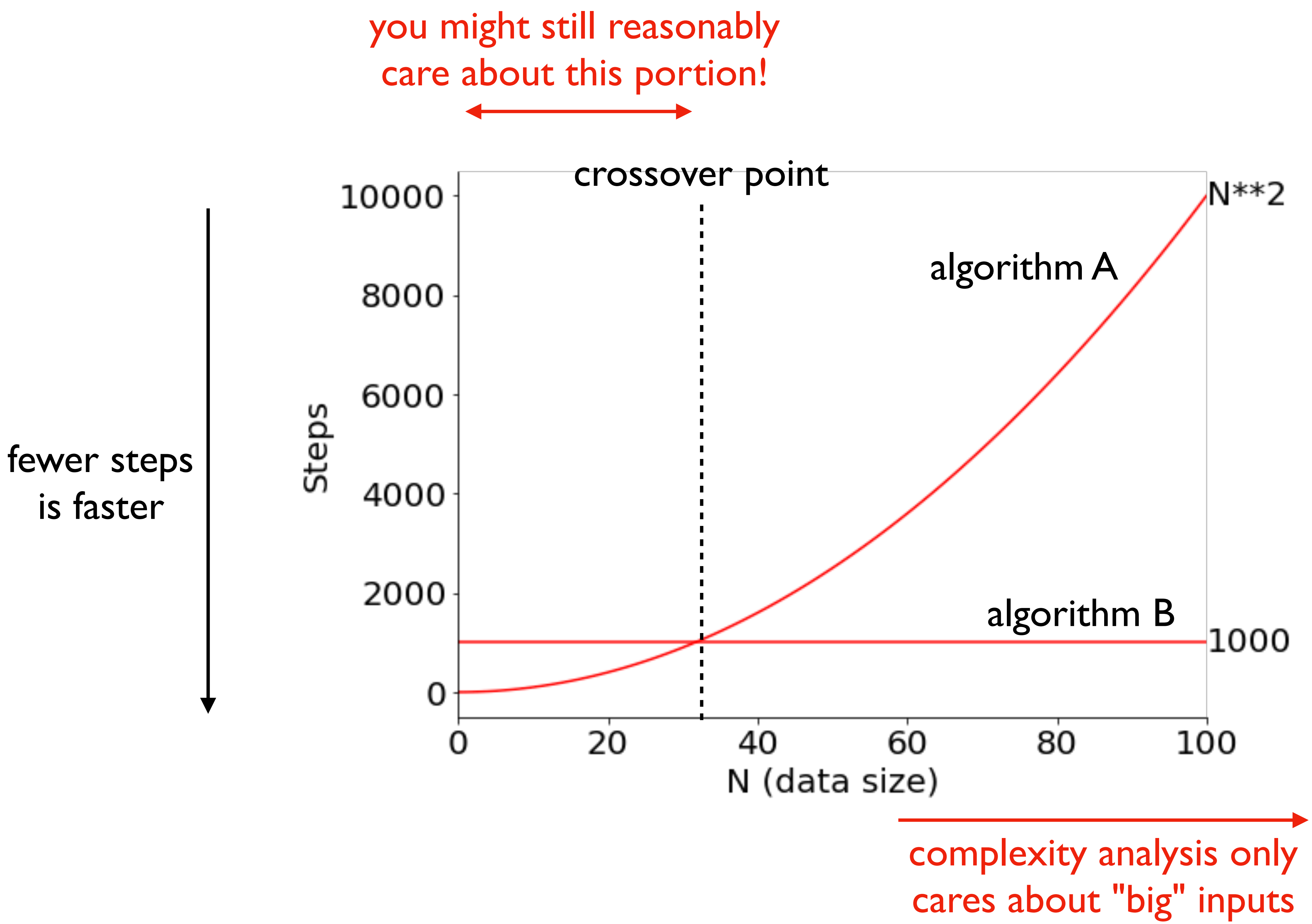
Do you prefer A or B?

Which algorithm is better?

fewer steps
is faster



Which algorithm is better?



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?

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

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?

What is a "step"?

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, ...]

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

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, ...]

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
```

STEP

STEP

STEP

STEP

???

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?

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:
    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, ...]
```

STEP

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

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

STEP

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

STEP

```
odd_avg = odd_sum / odd_count
```

Counting Executed 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, ...]

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
```

STEP

STEP

STEP

STEP

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
```

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, ...]

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
```

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

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

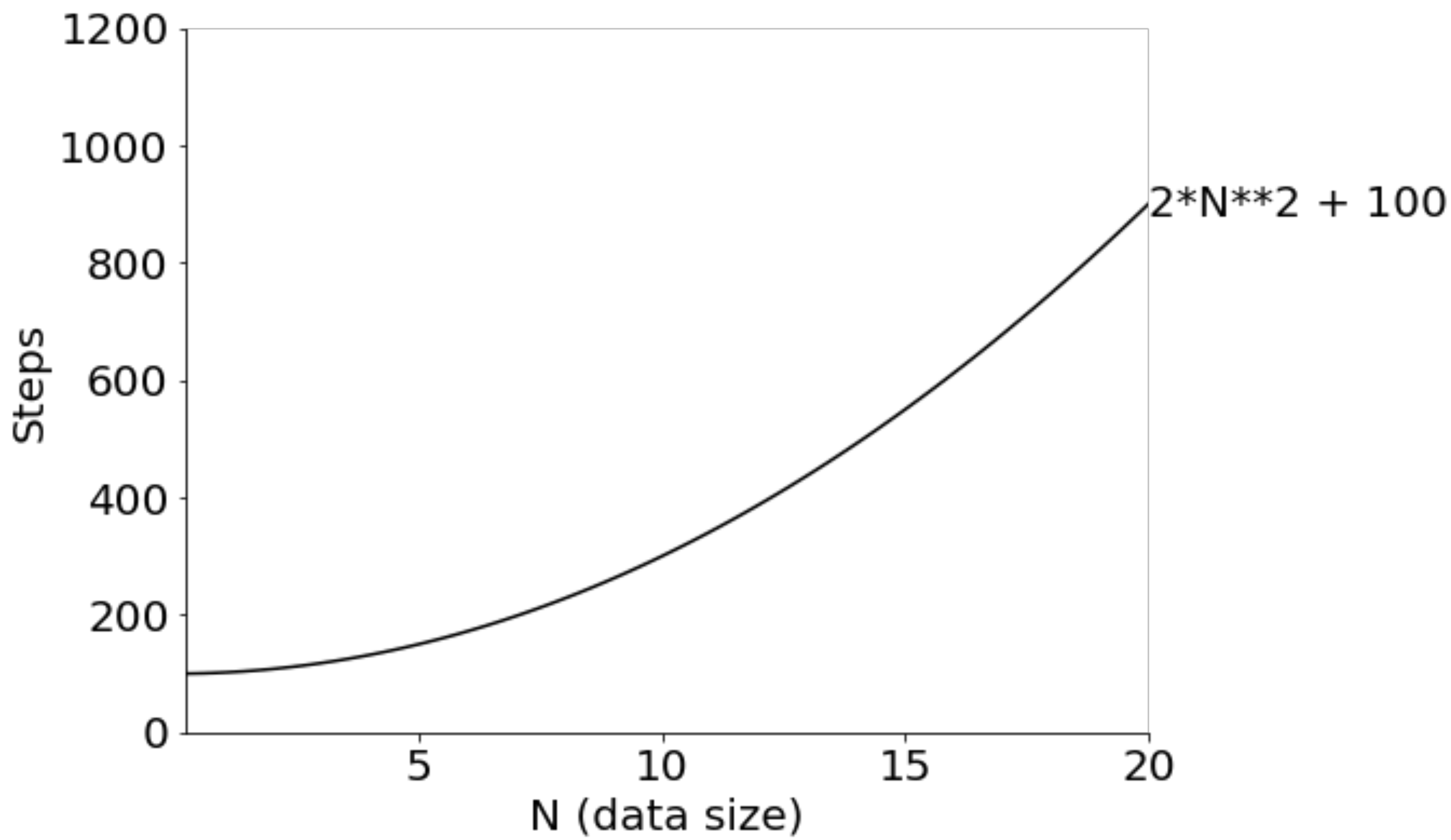
a really big "O"

O

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 is an upper bound

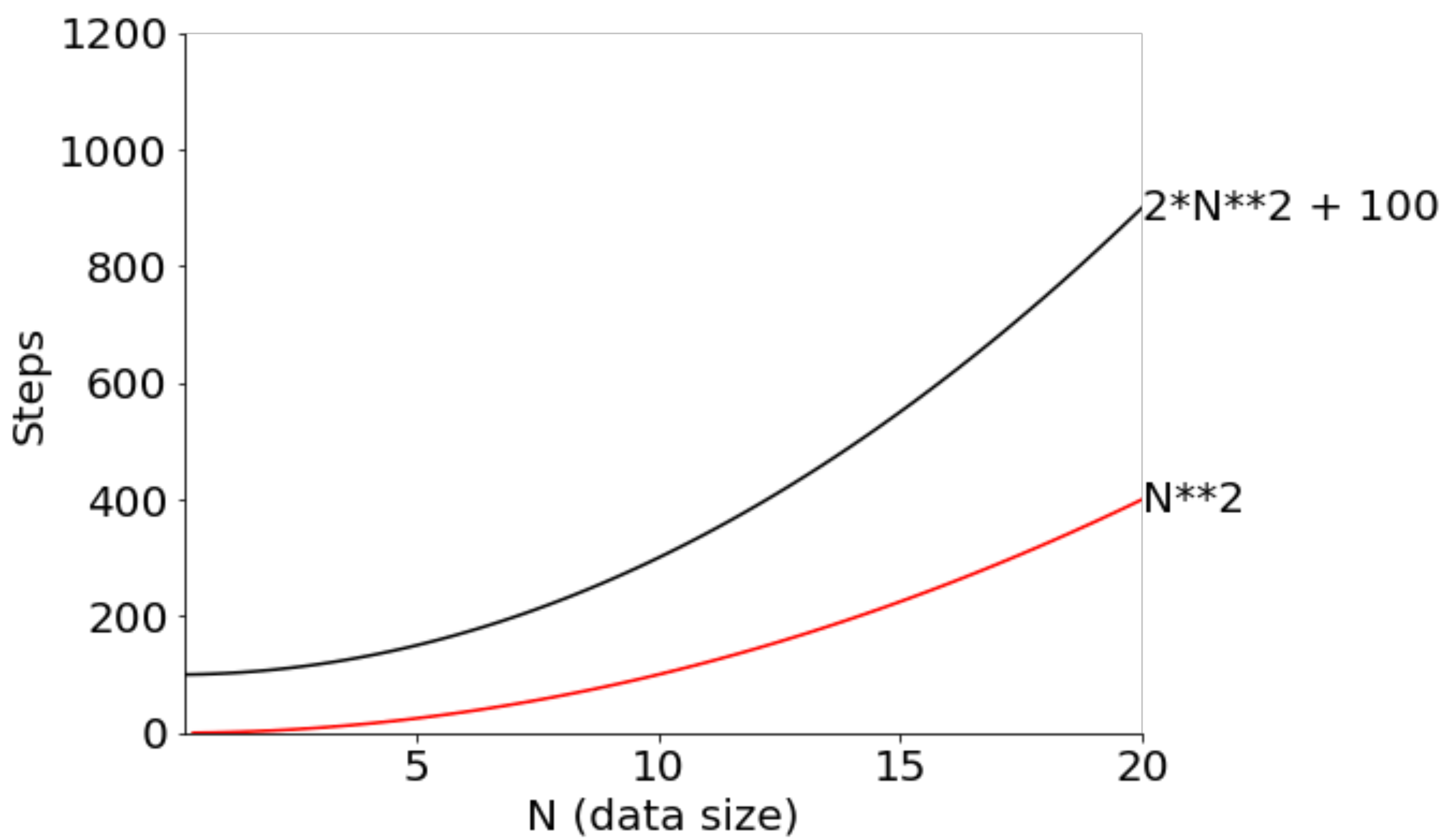


$f(N) == N^2 + 100$
is an **$O(N^2)$** function

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 is an upper bound



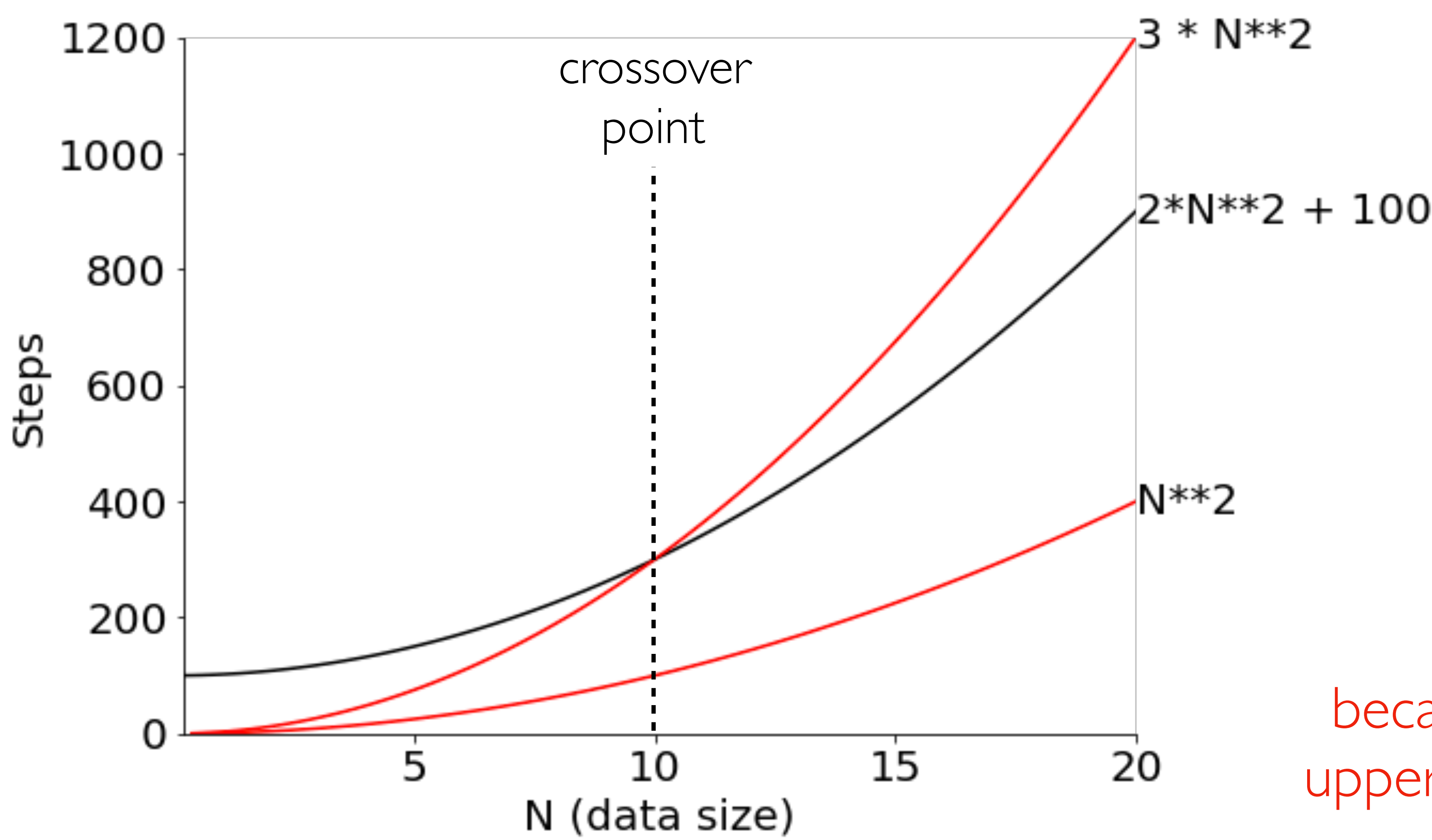
$f(N) == N^2 + 100$
is an **$O(N^2)$** function

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 is an upper bound



$f(N) == N^2 + 100$
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

If $f(N) < C * g(N)$ *for large N values and some fixed* *<u>constant</u>* *C*

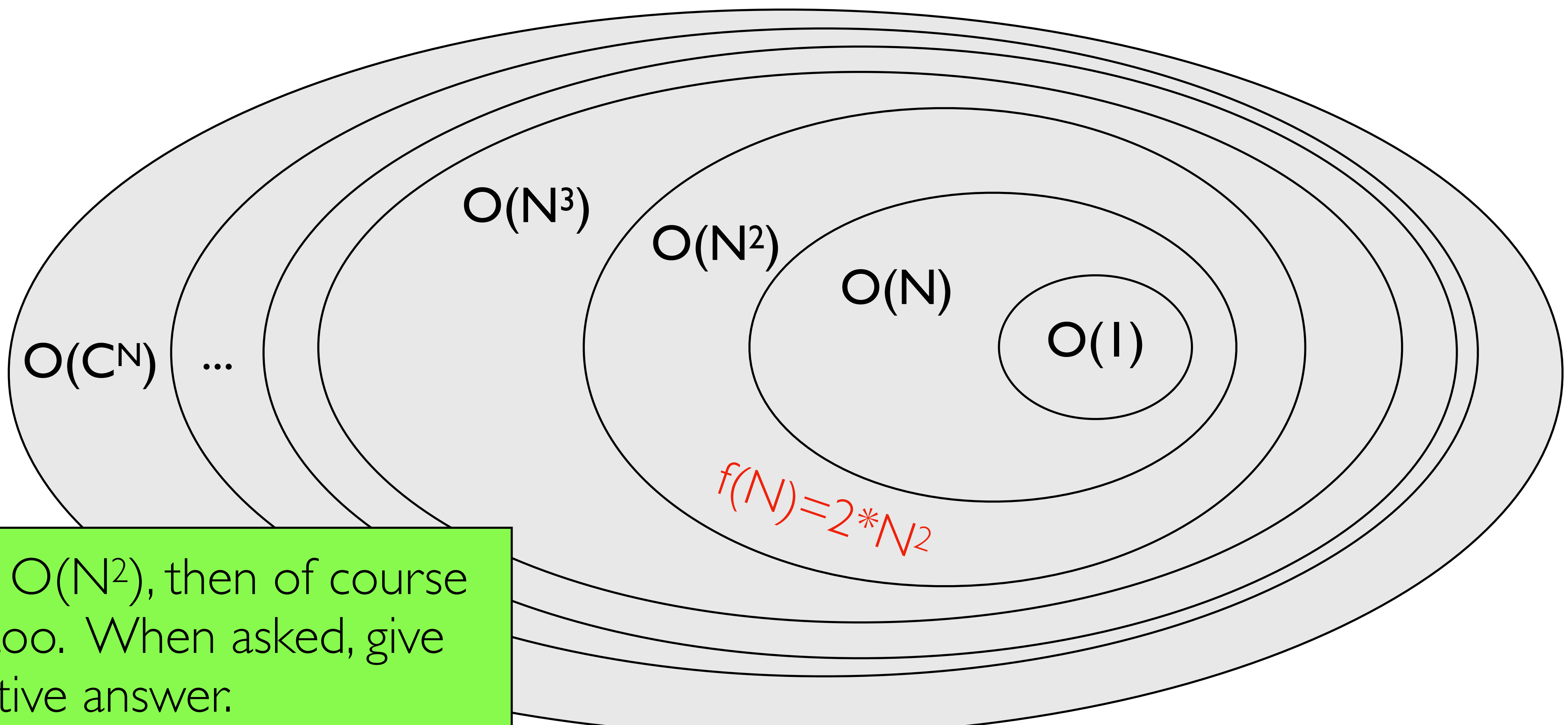
Then $f(N) \in O(g(N))$

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed <u>constant</u> C*

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

Sets



Note: if f(N) is in $O(N^2)$, then of course f(N) is in $O(N^3)$ too. When asked, give the most informative answer.

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed* *<u>constant</u>* *C*

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

---

which ones are true?

$2N \in O(N)$

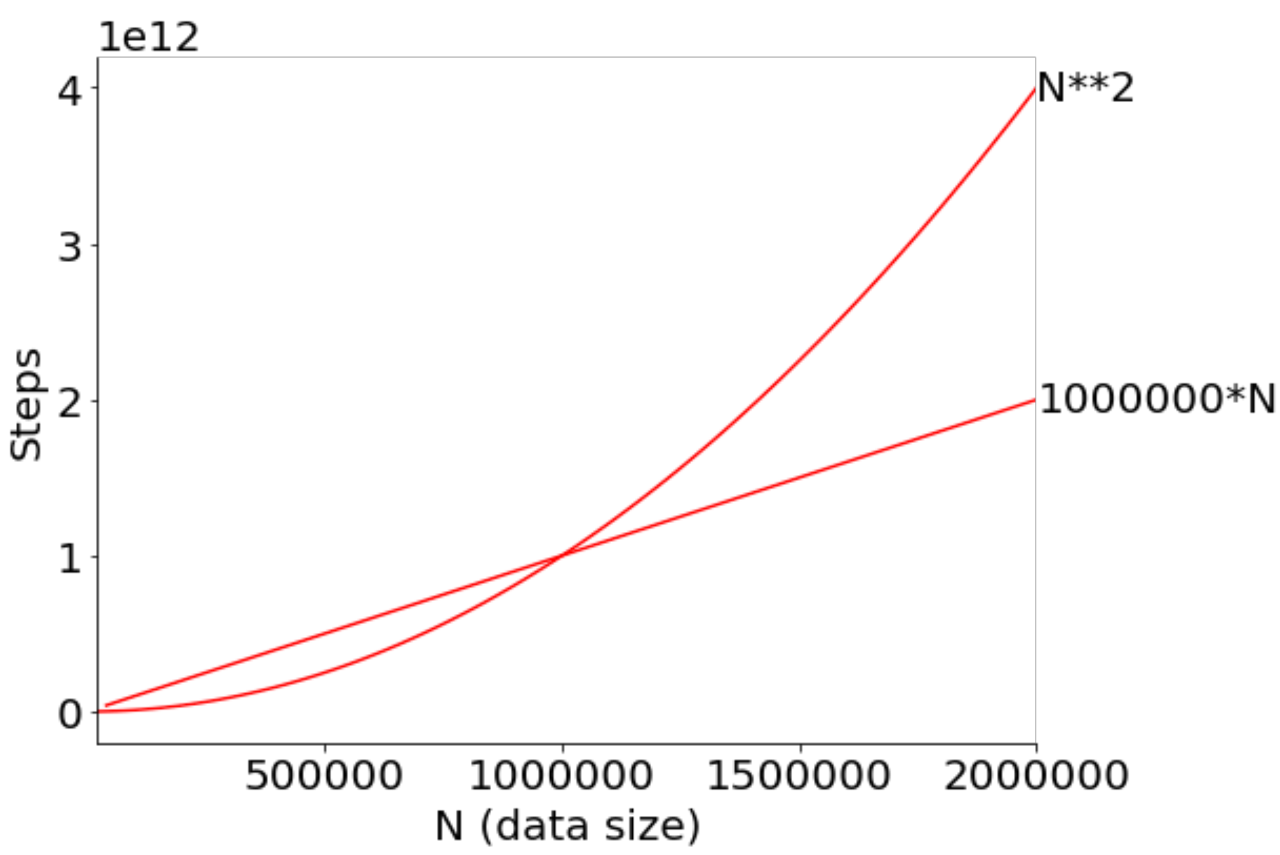
$100N \in O(N^2)$

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

Defining Big O

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

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



$2N \in O(N)$

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

$\cancel{N^2 \in O(1000000N)}$

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed <u>constant</u> C*

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

---

which ones are true?

$N^2 \in O(N^2+N+1)$

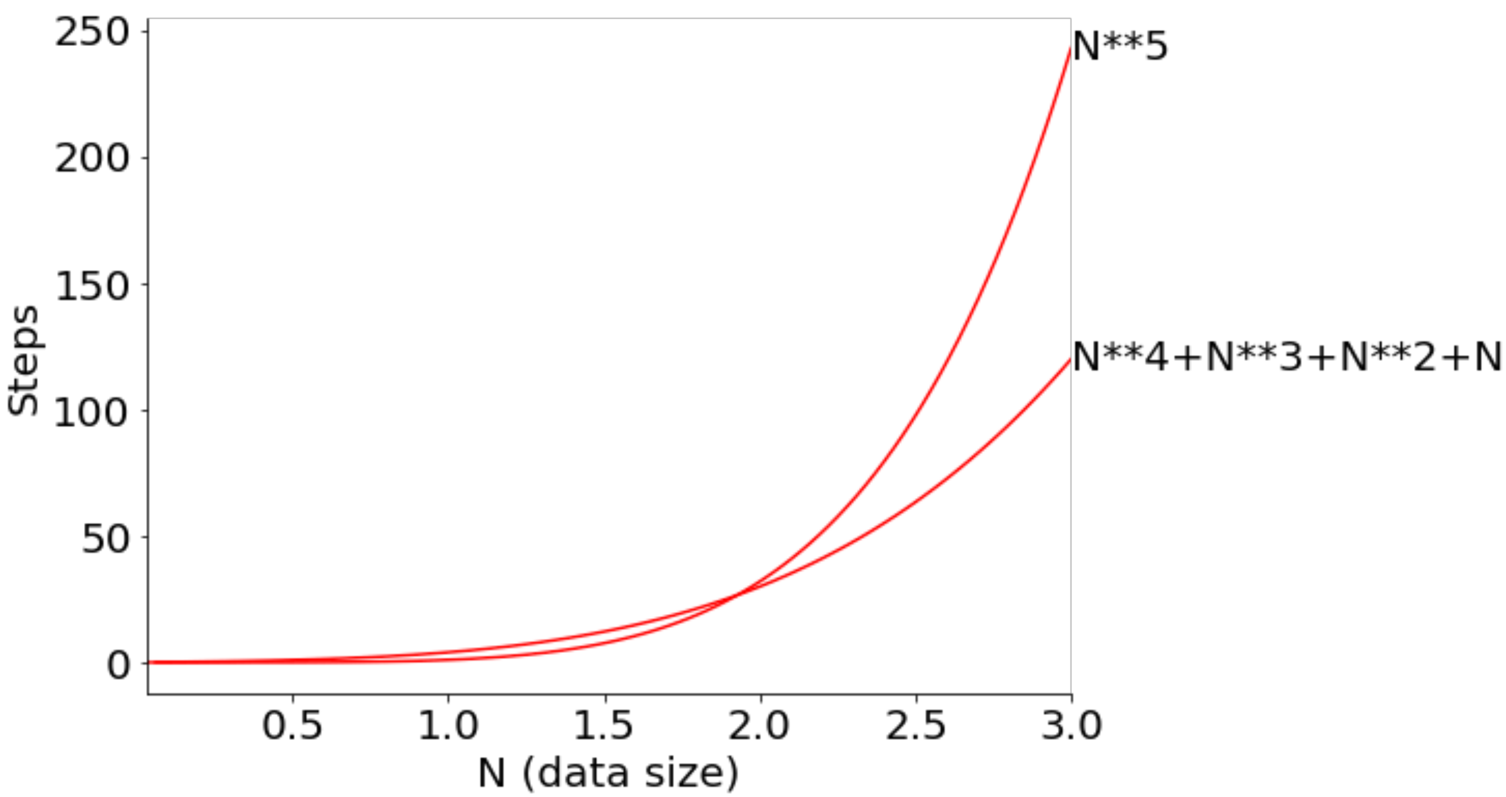
$N^2+N+1 \in O(N^2)$

$N^5 \in O(N^4 + N^3 + N^2 + N)$

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed* *<u>constant</u>* *C*

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



$N^2 \in O(N^2+N+1)$

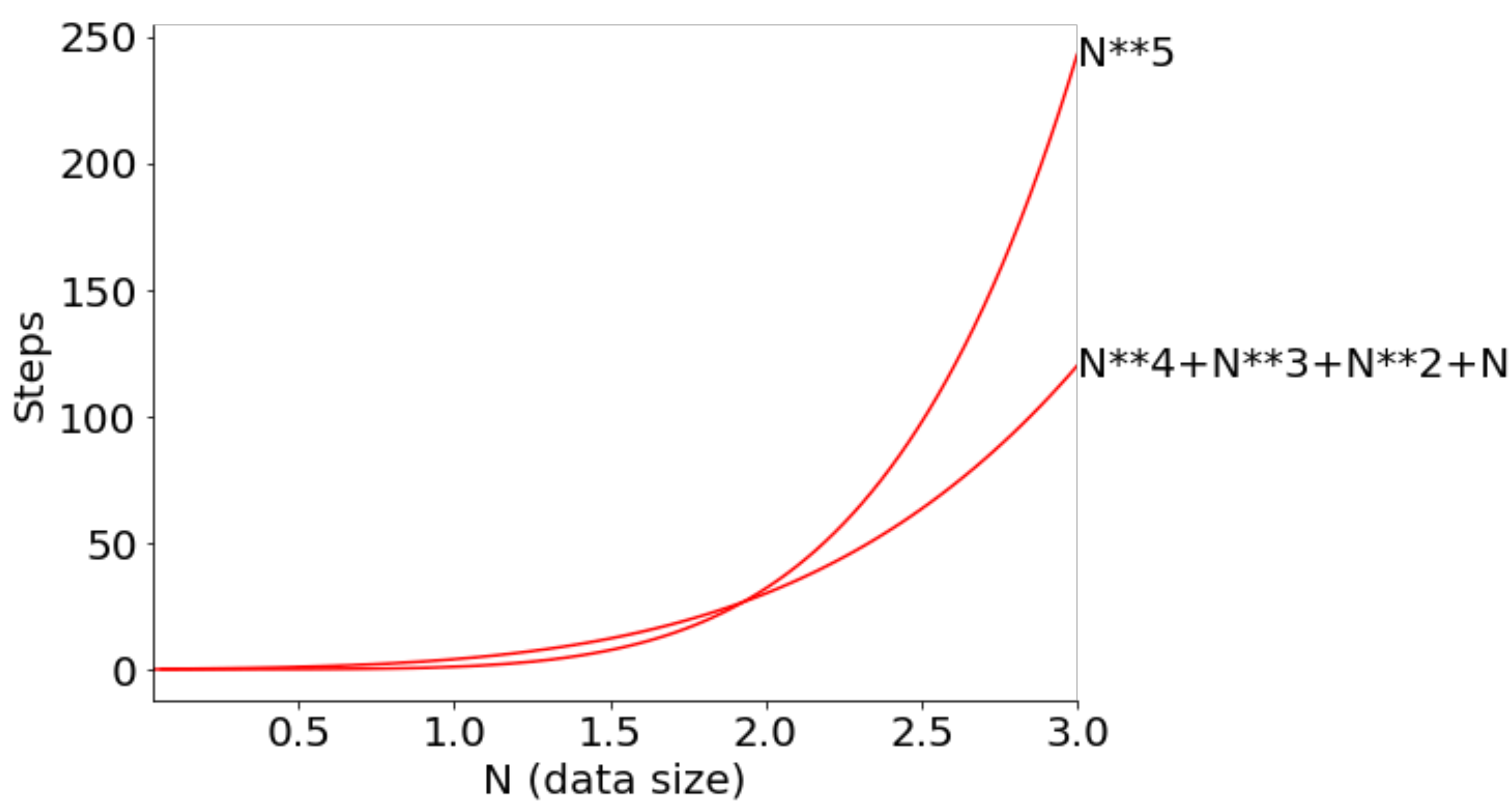
$N^2+N+1 \in O(N^2)$

$\cancel{N^5 \in O(N^4 + N^3 + N^2 + N)}$

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed <u>constant</u> C*

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



simplify when possible

$N^2 \in O(N^2 \cancel{+N+1})$

$N^2+N+1 \in O(N^2)$

$\cancel{N^5 \in O(N^4 + N^3 + N^2 + N)}$

Defining Big O

If $f(N) < C * g(N)$ *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 $f(N) < C * g(N)$ *for large N values and some fixed <u>constant</u> C*

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

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

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

$2*N+3 < 3 * N$

[for big N values]

therefore

this code is O(N)

Defining Big O

If $f(N) < C * g(N)$ *for large N values and some fixed <u>constant</u> 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

$4*N+5 < 5 * N$

[for big N values]

therefore

this code is O(N)

Examples

Coding/Plotting Example

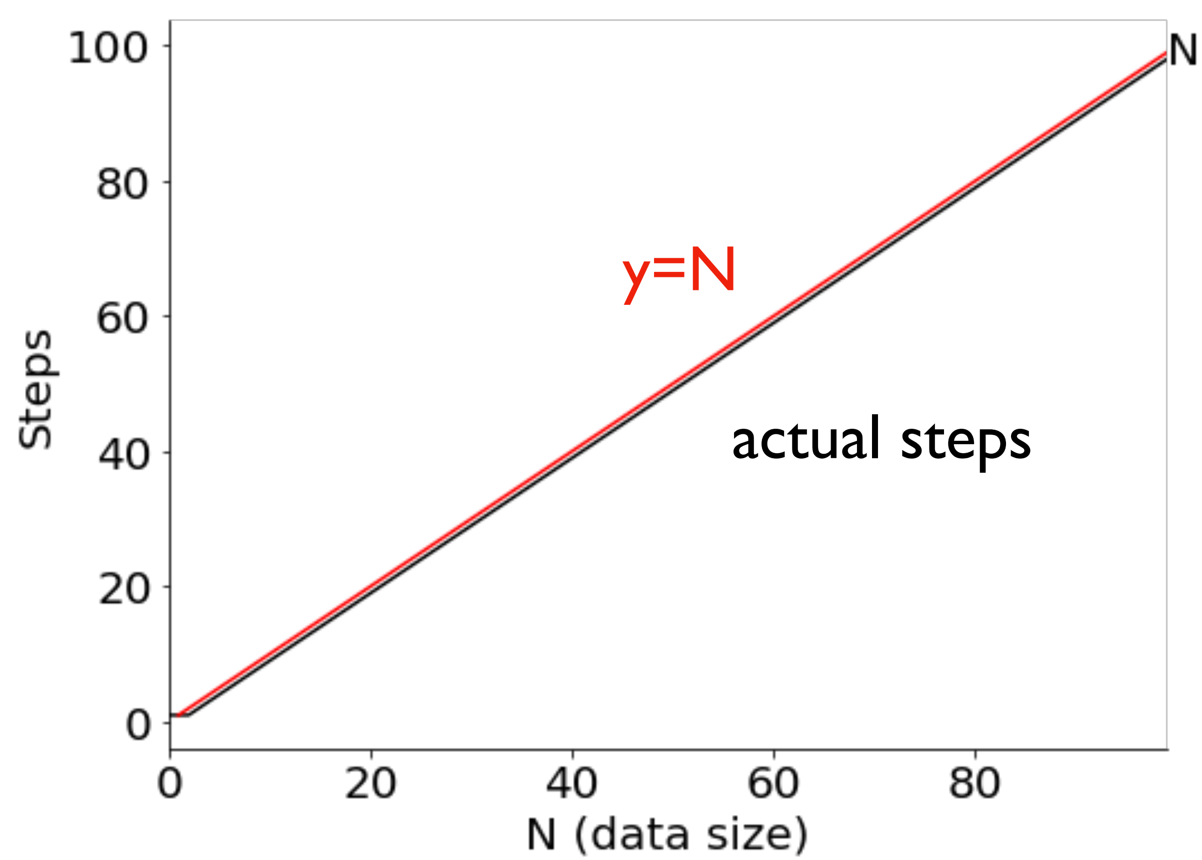
```
def is_prime(N):
    prime = True
    for factor in range(2, N):
        steps += 1
        if N % factor == 0:
            prime = False
    return prime
```

what is the complexity of each function

```
def find_primes(cap):
    primes = []
    for i in range(cap+1):
        if is_prime(i):
            primes.append(i)
    return primes
```

Coding/Plotting Example

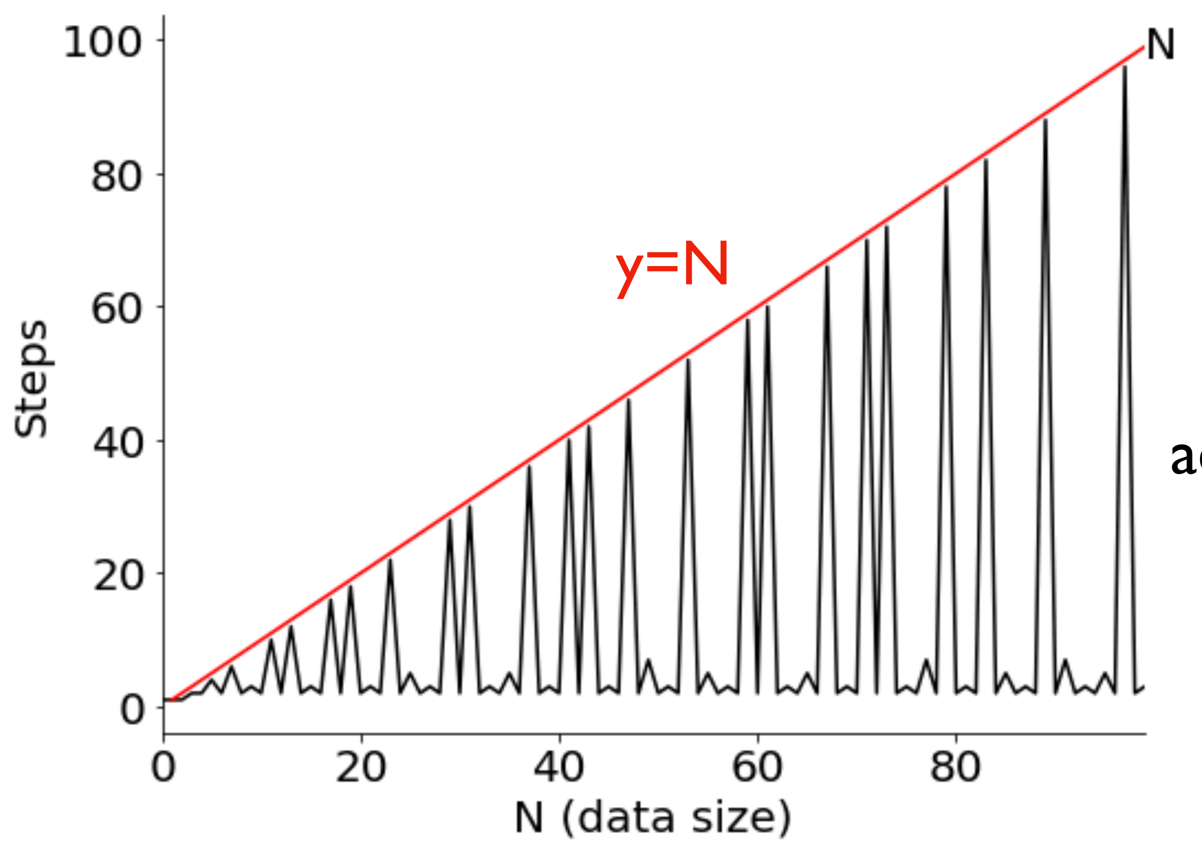
```
def is_prime(N):
    prime = True
    for factor in range(2, N):
        steps += 1
        if N % factor == 0:
            prime = False
    return prime
```



Coding/Plotting Example

```
def is_prime(N):
    prime = True
    for factor in range(2, N):
        steps += 1
        if N % factor == 0:
            prime = False
    return prime
```

what if we add a break here?

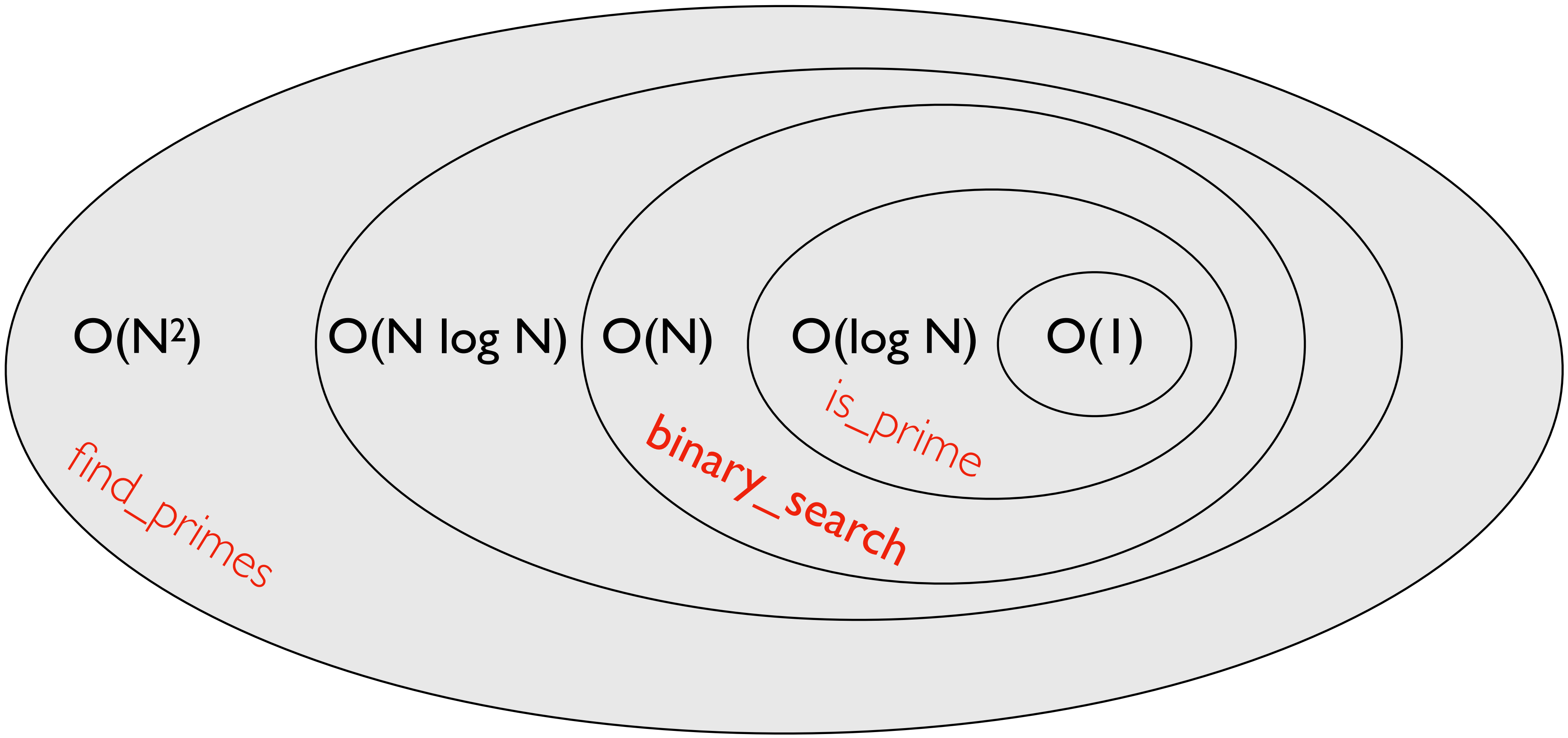


actual steps

for simplicity, we'll usually do a worst-case analysis, under which this would still be O(N)

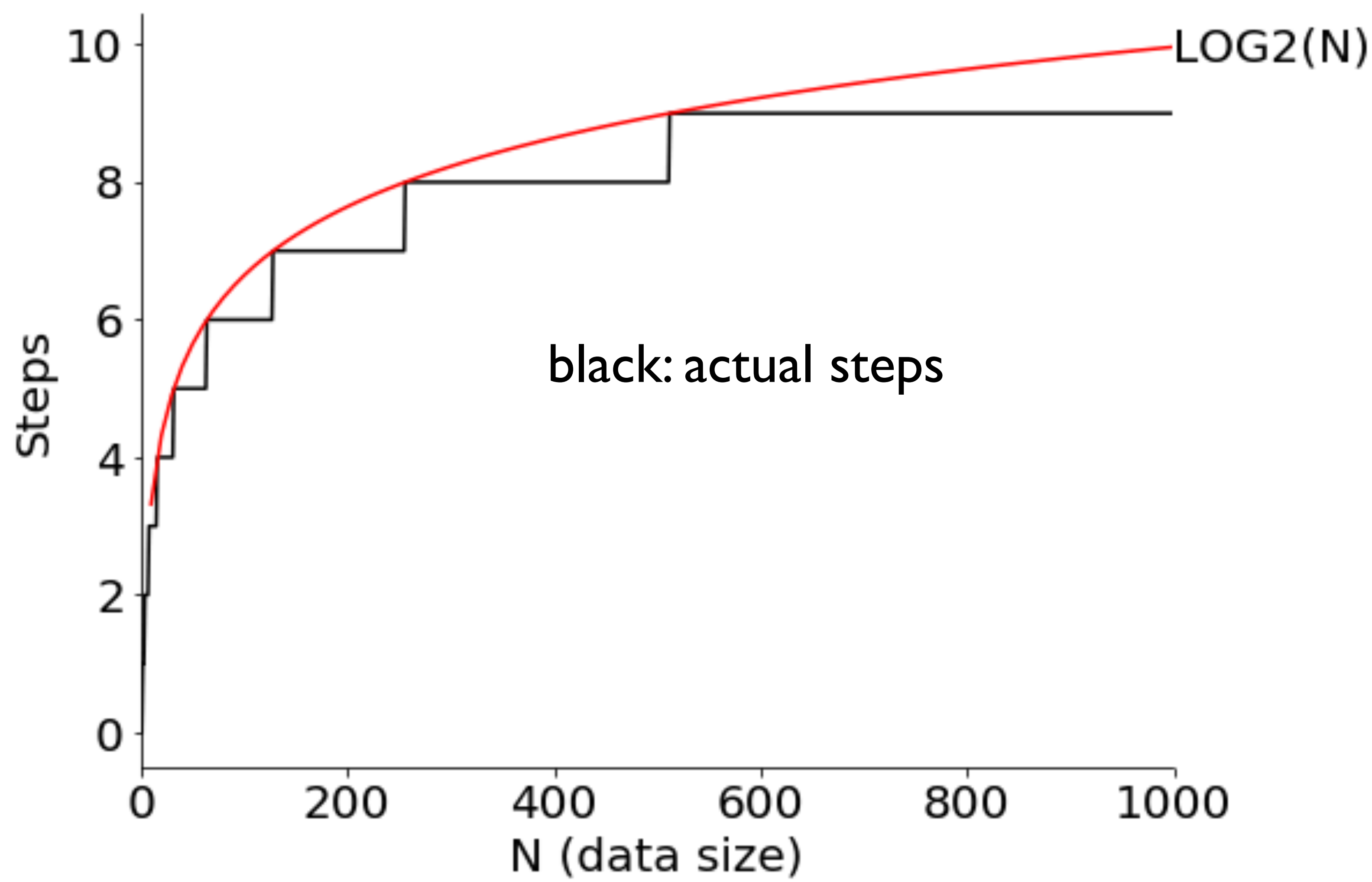
implications for `X in L`?

Binary Search: Coding Example

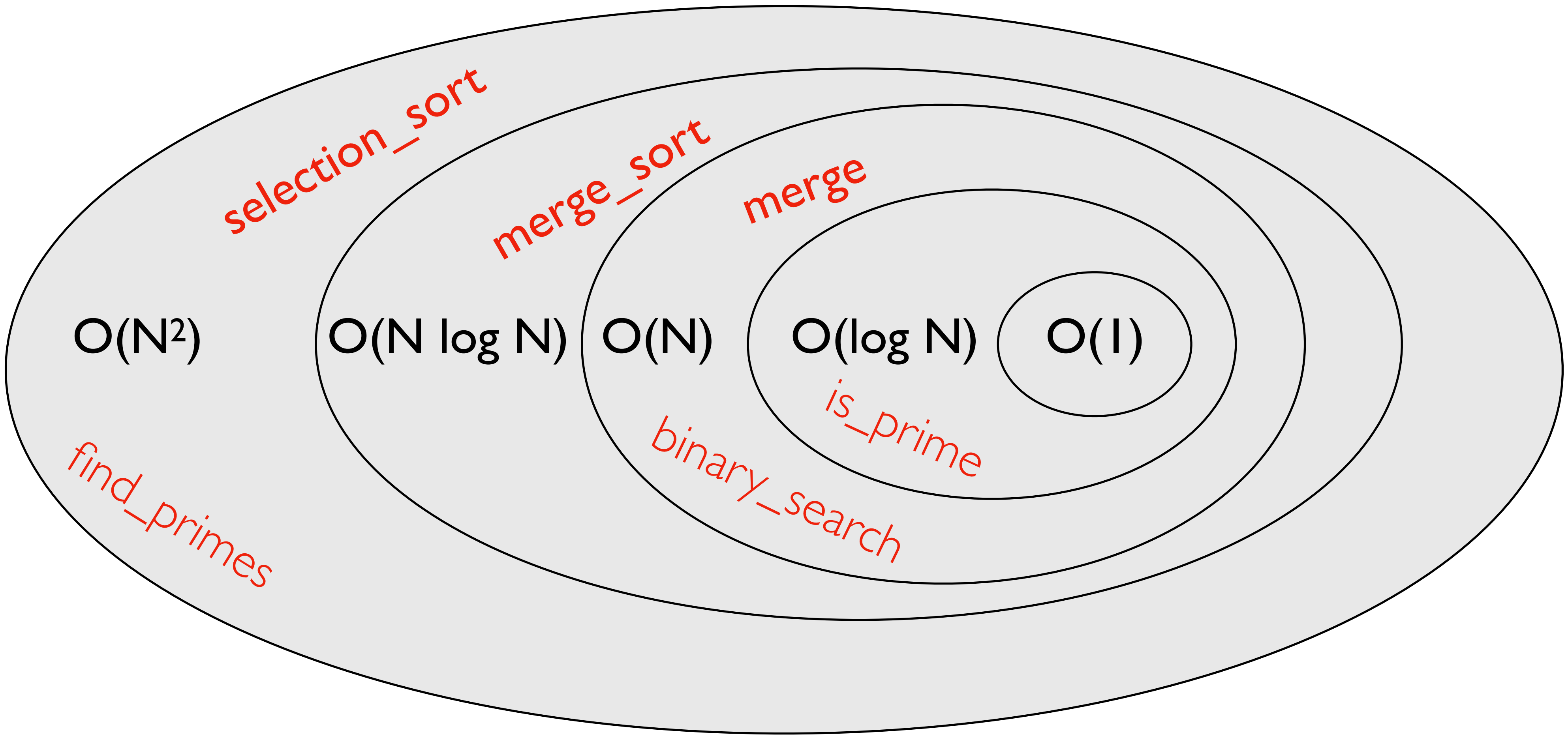


Binary Search

Binary Search: Coding Example



Sorting: Coding Examples



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