Review of Time Complexity Analysis

1/15/25

Announcements

- If you’re not on Piazza or Gradescope, make sure to join
  - Let me know if you need help
- Homework #1 is out (due 1/28)
  - Topic: warm-up problems + asymptotic notation
- Syllabus, website, and dates are finalized
- Practice quizzes on course website

Exact Time Complexity Analysis

Reminder: The RAM Model

- Each "simple" operation (+, -, =, if, call) takes 1 time step.
- Loops and subroutine calls are *not* simple operations. They depend upon the size of the data and the contents of a subroutine.
- Each memory access takes 1 step.
- There is infinite memory

Algorithmic Complexity (i.e. Time and Space Complexity)

- Estimate of time or space an algorithm requires
- Description of how time/space requirements increase with problem size

What is problem size? (a.k.a. N)

- The aspect of the input that will cause algorithmic complexity to increase
  - Array length
  - Size of a number
  - etc.
- If ambiguous, define

Example

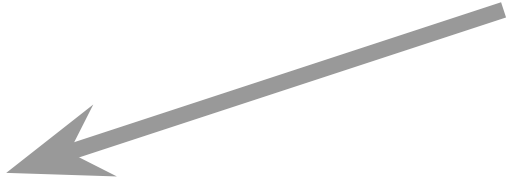
```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```

Example

```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```

Example

```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```



n + 1 steps

Example

```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```

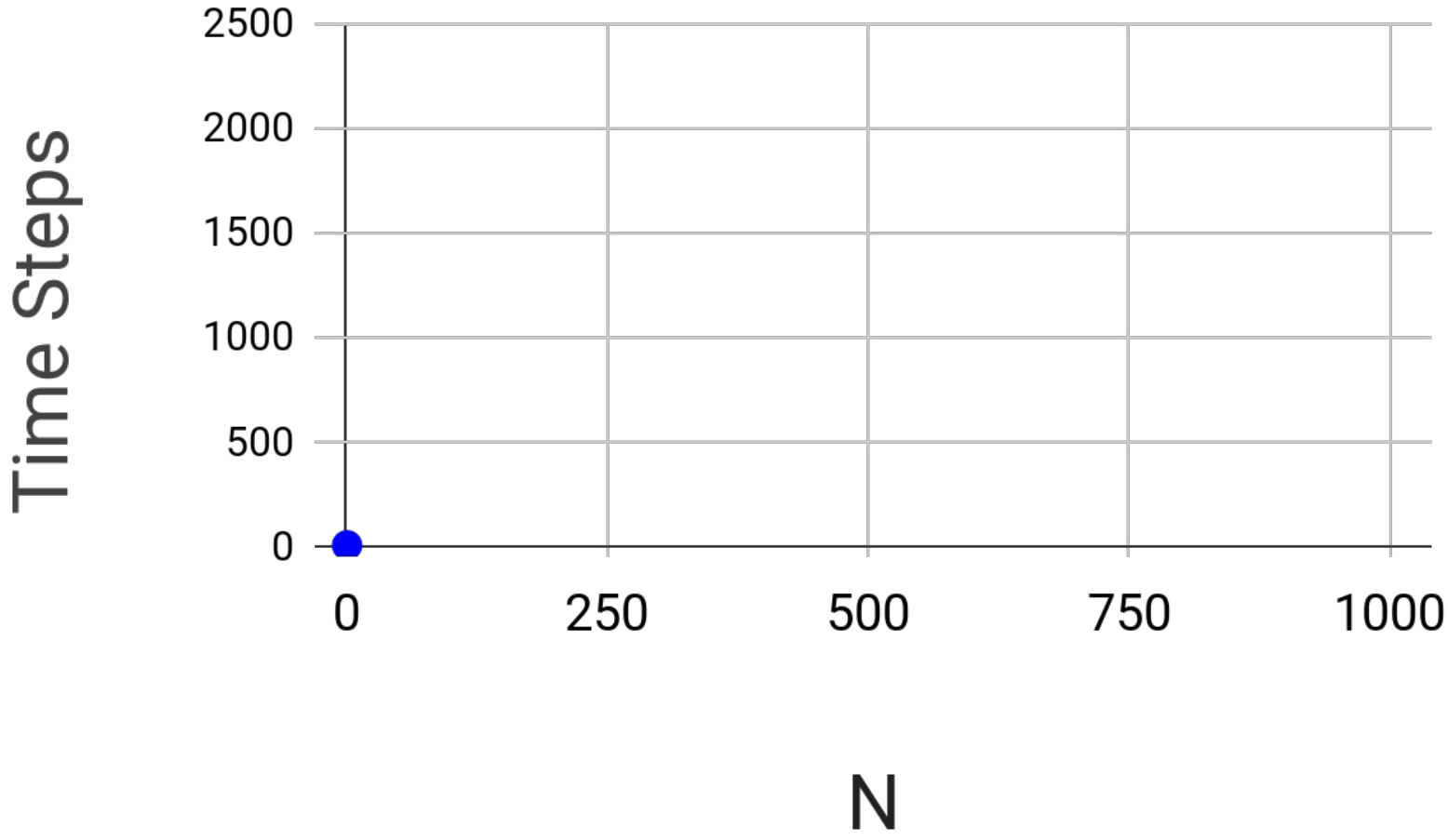
Example

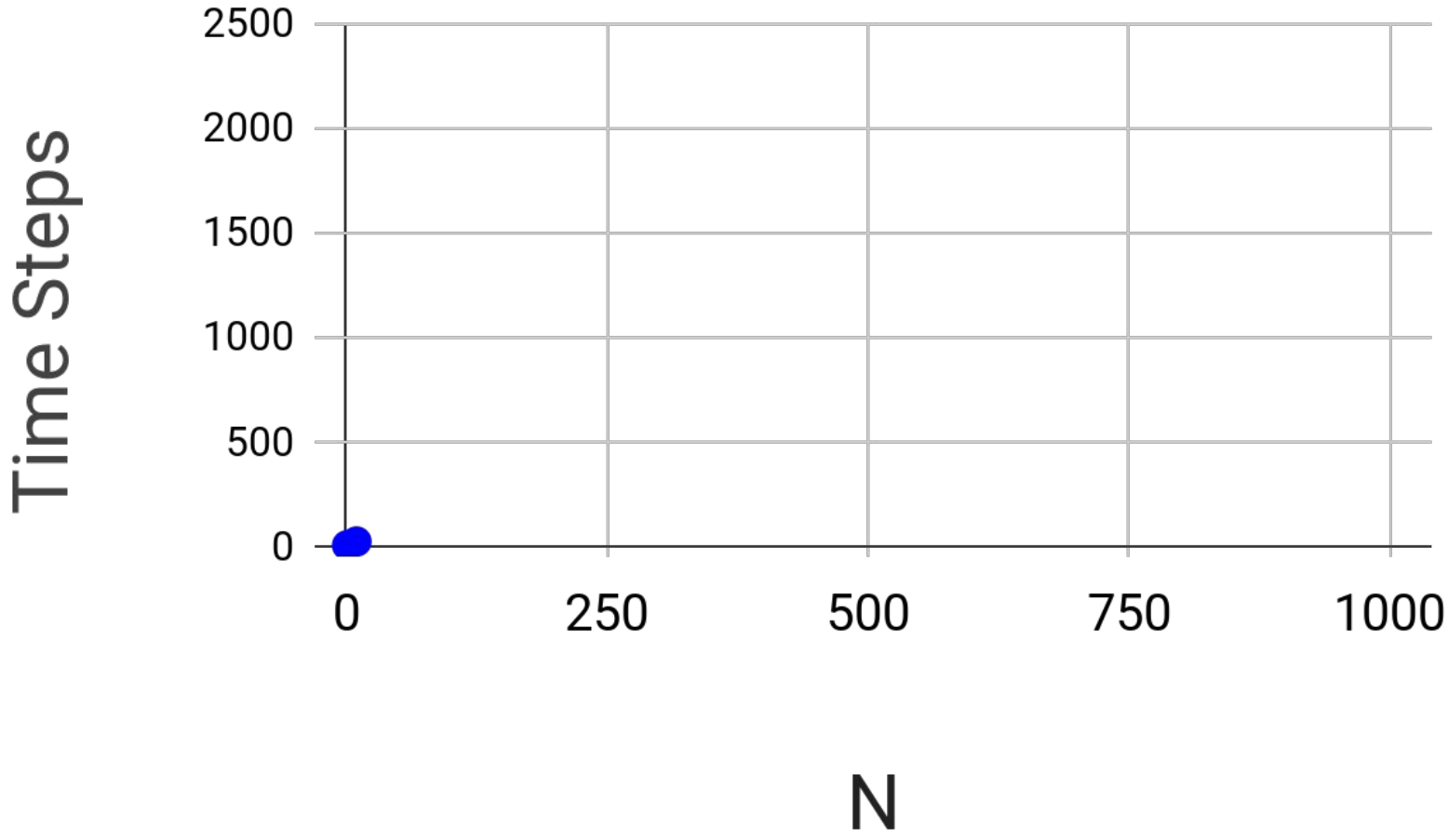
```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```

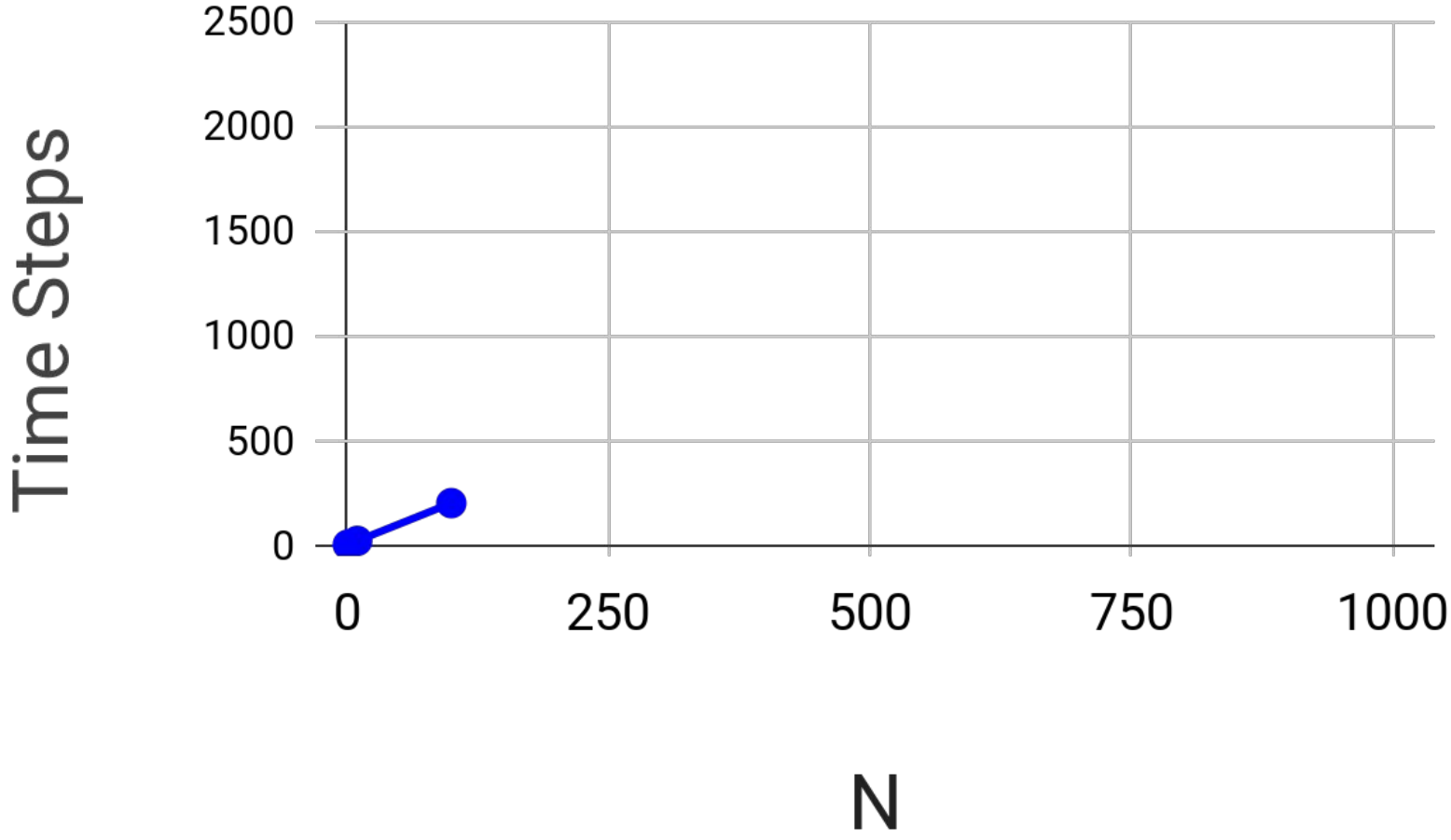
Example

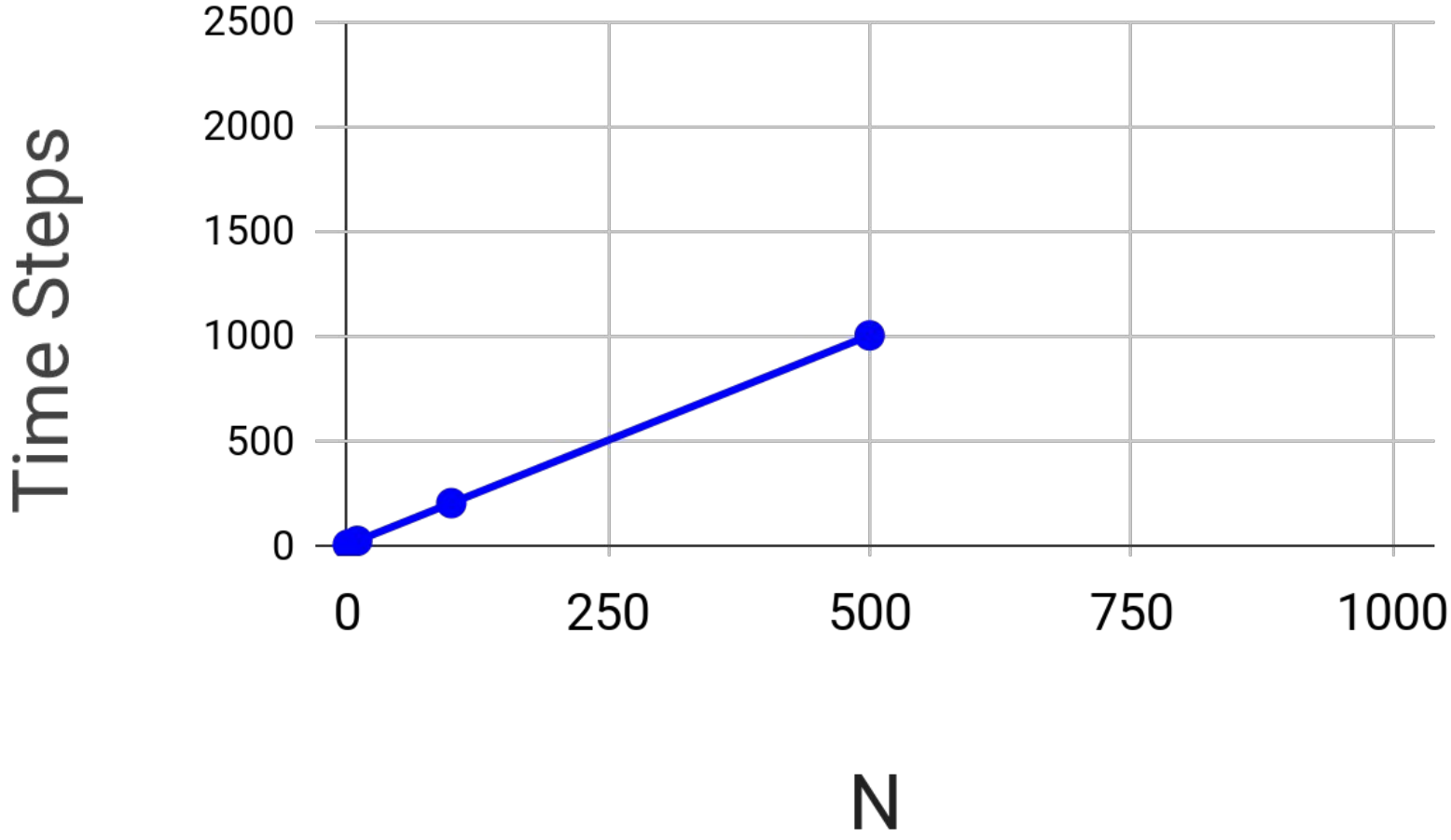
```
def sum_numbers(n):
    total = 0
    for num in range(n):
        total += num
    return total
```

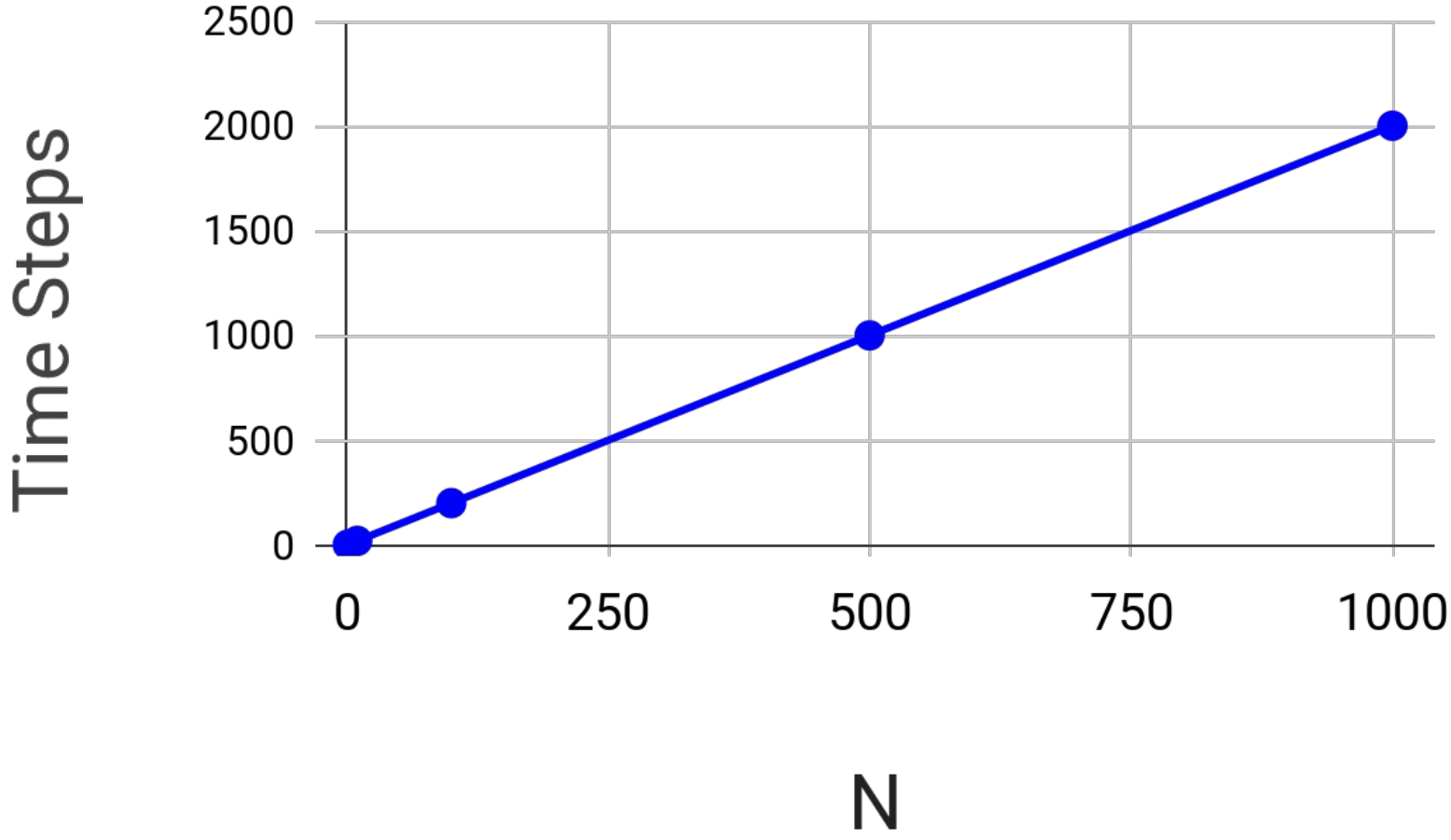
Total: 2n + 3 steps


Time Steps
2500
2000
1500
1000
500
0
0
250
500
750
1000
N

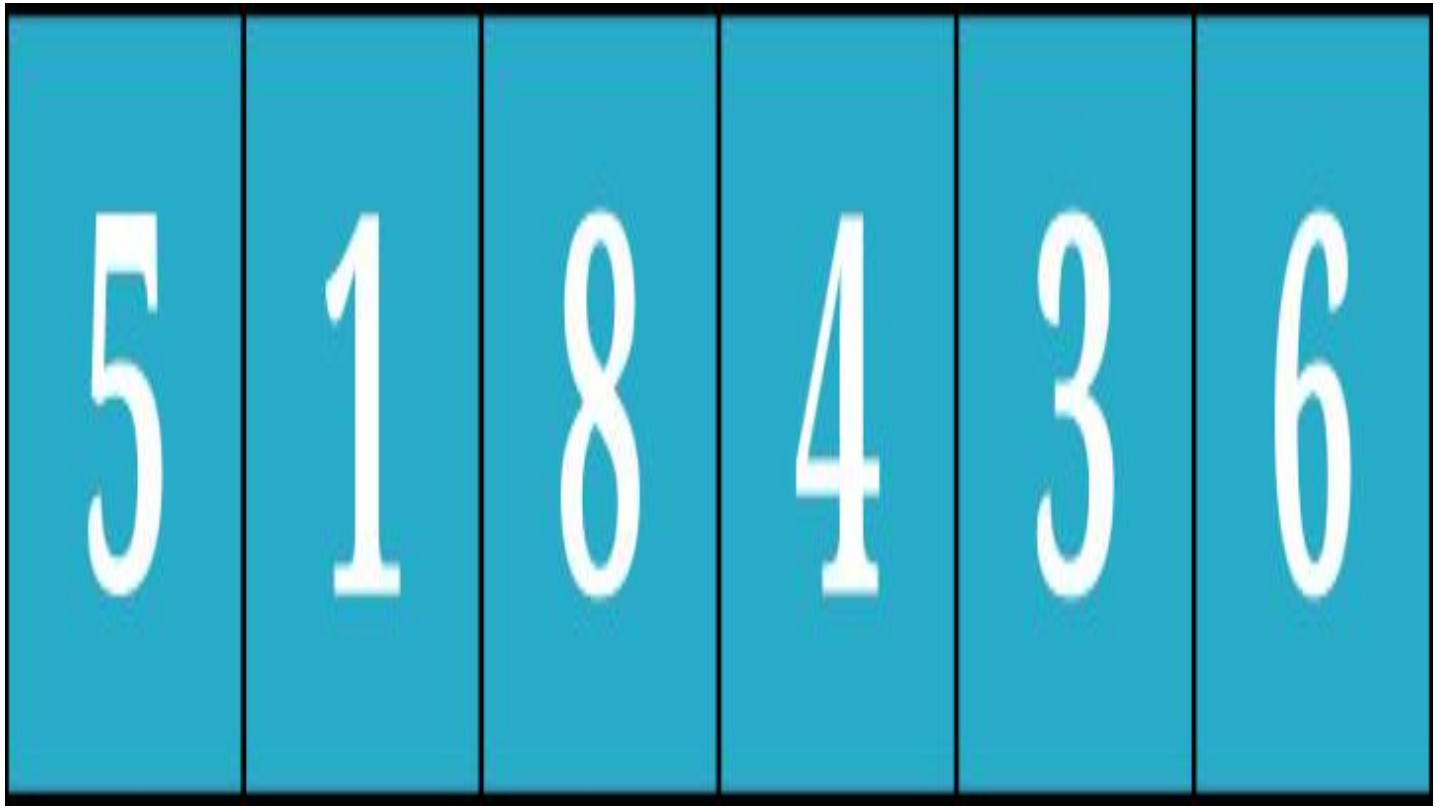

Time Steps
2500
2000
1500
1000
500
0
0
250
500
750
1000
N


Time Steps
2500
2000
1500
1000
500
0
0
250
500
750
1000
N


Time Steps
2500
2000
1500
1000
500
0
0
250
500
750
1000
N


Time Steps
2500
2000
1500
1000
500
0
0
250
500
750
1000
N

A trickier case for exact analysis: Insertion Sort



A trickier case for exact analysis: Insertion Sort

```
for i = 1 to n
    key = A[i]
    j = i - 1
    while j ≥ 0 AND A[j] > key
        A[j+1] = A[j]
        j = j -1
    A[j+1] = key
```

A trickier case for exact analysis: Insertion Sort

```
                                                    Num Exec.
for i = 1 to n                                      (n-1) + 1
    key = A[i]
    j = i - 1
    while j ≥ 0 AND A[j] > key
        A[j+1] = A[j]
        j = j -1
    A[j+1] = key
```

A trickier case for exact analysis: Insertion Sort

```
                                                    Num Exec.
for i = 1 to n                                      (n-1) + 1
    key = A[i]                                        n-1
    j = i - 1                                         n-1
    while j ≥ 0 AND A[j] > key
        A[j+1] = A[j]
        j = j -1
    A[j+1] = key
```

A trickier case for exact analysis: Insertion Sort

	Num Exec.
for i = 1 to n	(n-1) + 1
key = A[i]	n-1
j = i - 1	n-1
while j ≥ 0 AND A[j] > key	?
A[j+1] = A[j]	?
j = j -1	?
A[j+1] = key	n-1

We can analyze best, average, and worst cases

What is the best case input for insertion sort?

What is the best case input for insertion sort?

A sorted list!

Exact Analysis: Best Case

	Num Exec.
for i = 1 to n	(n-1) + 1
key = A[i]	n-1
j = i - 1	n-1
while j ≥ 0 AND A[j] > key	?
A[j+1] = A[j]	?
j = j -1	?
A[j+1] = key	n-1

Exact Analysis: Best Case

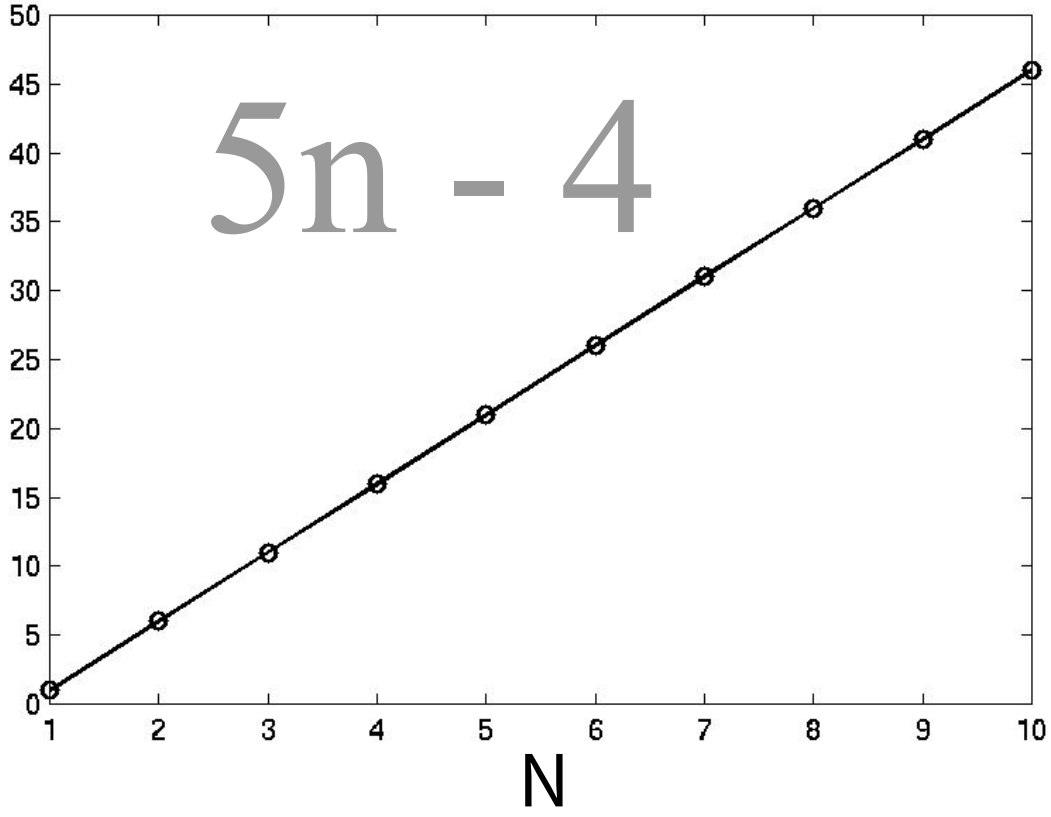
	Num Exec.
for i = 1 to n	(n-1) + 1
key = A[i]	n-1
j = i - 1	n-1
while j ≥ 0 AND A[j] > key	n-1
A[j+1] = A[j]	0
j = j -1	0
A[j+1] = key	n-1

Exact Analysis: Best Case

$$5n - 4$$

Exact Analysis: Best Case

Steps



What is the worst case input for insertion sort?

What is the worst case input for insertion sort?

A reverse sorted list!

Exact Analysis: Worst Case

	Num Exec.
for i = 1 to n	(n-1) + 1
key = A[i]	n-1
j = i - 1	n-1
while j $\geq$ 0 AND A[j] > key	?
A[j+1] = A[j]	?
j = j -1	?
A[j+1] = key	n-1

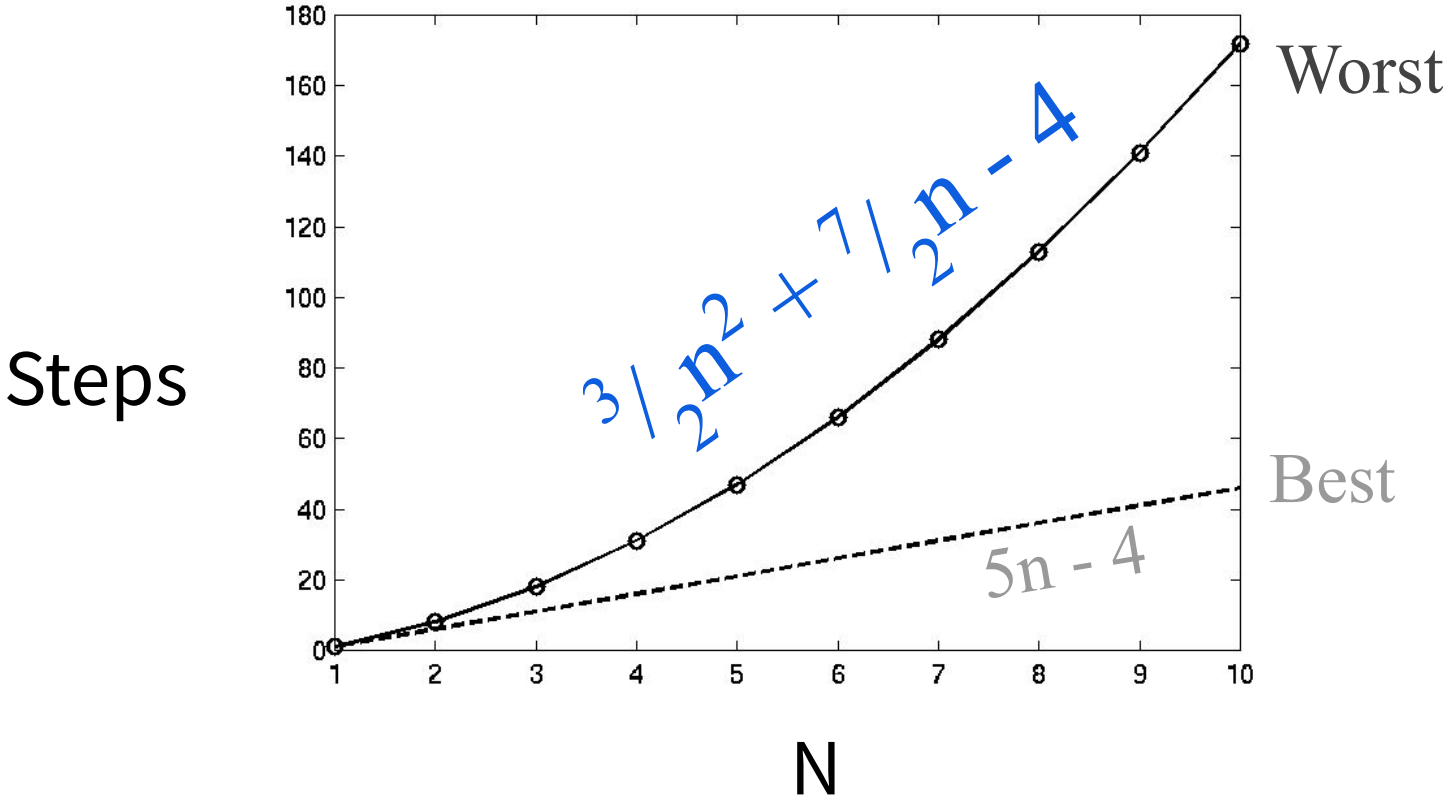
Exact Analysis: Worst Case

	Num Exec.
for i = 1 to n	(n-1) + 1
key = A[i]	n-1
j = i - 1	n-1
while j $\geq$ 0 AND A[j] > key	n(n-1)/2+n-1
A[j+1] = A[j]	n(n-1)/2
j = j -1	n(n-1)/2
A[j+1] = key	n-1

Exact Analysis: Worst Case

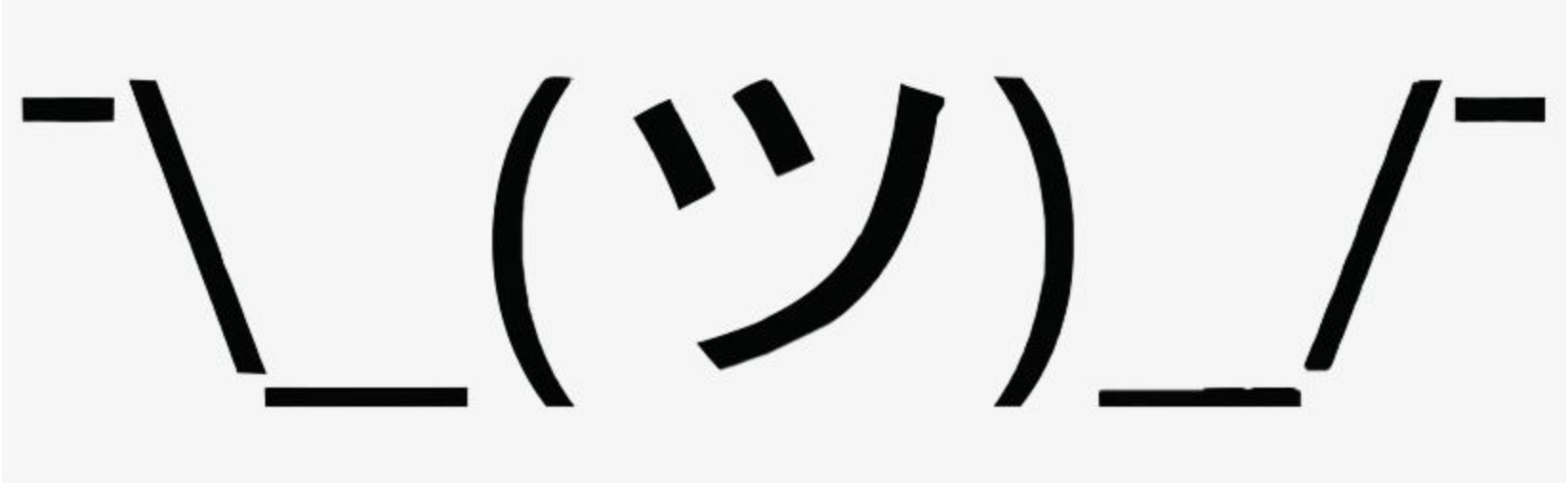
$$^{3}/_{2}n^{2} + ^{7}/_{2}n - 4$$

Exact Analysis: Worst Case

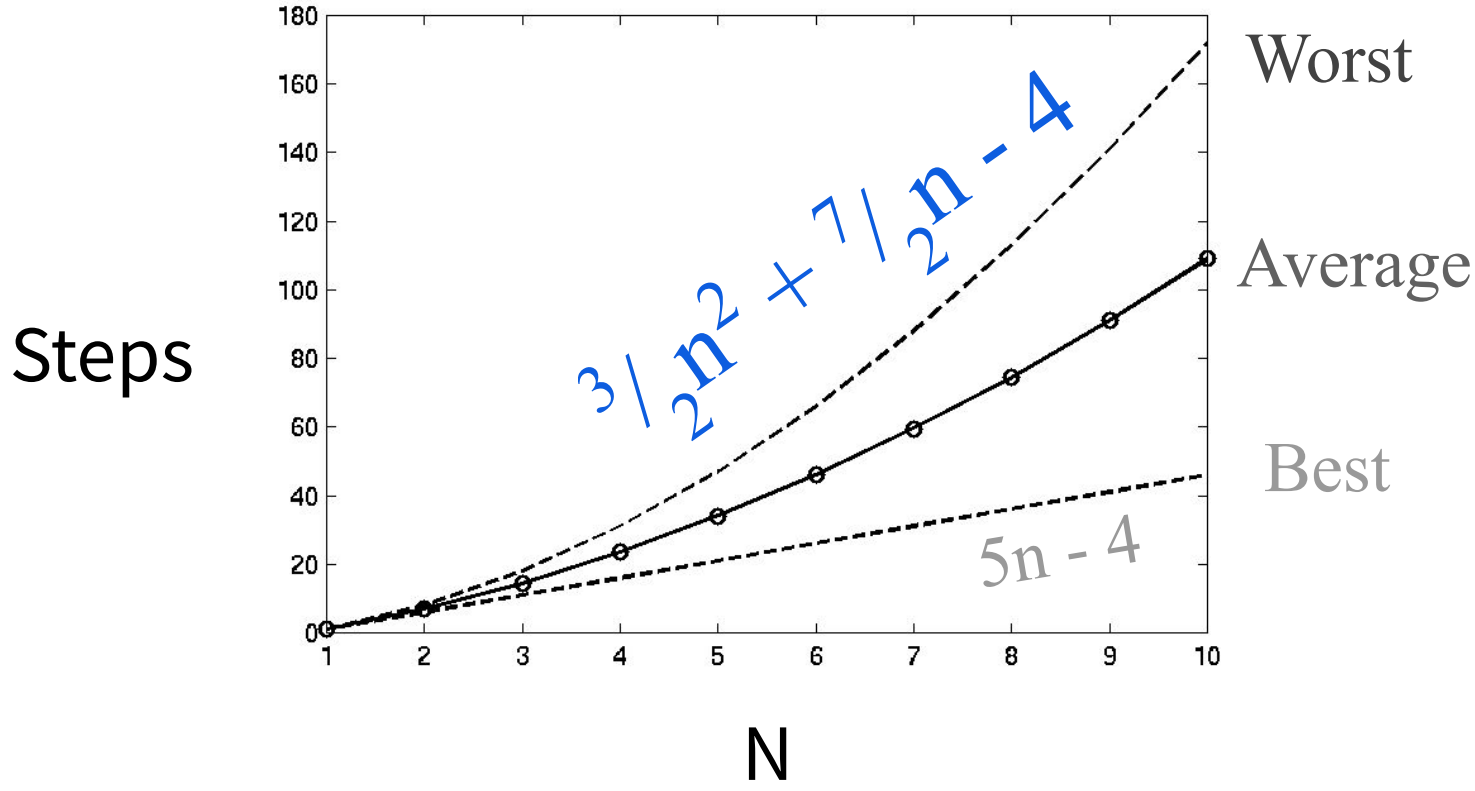


What is the average case input for insertion sort?

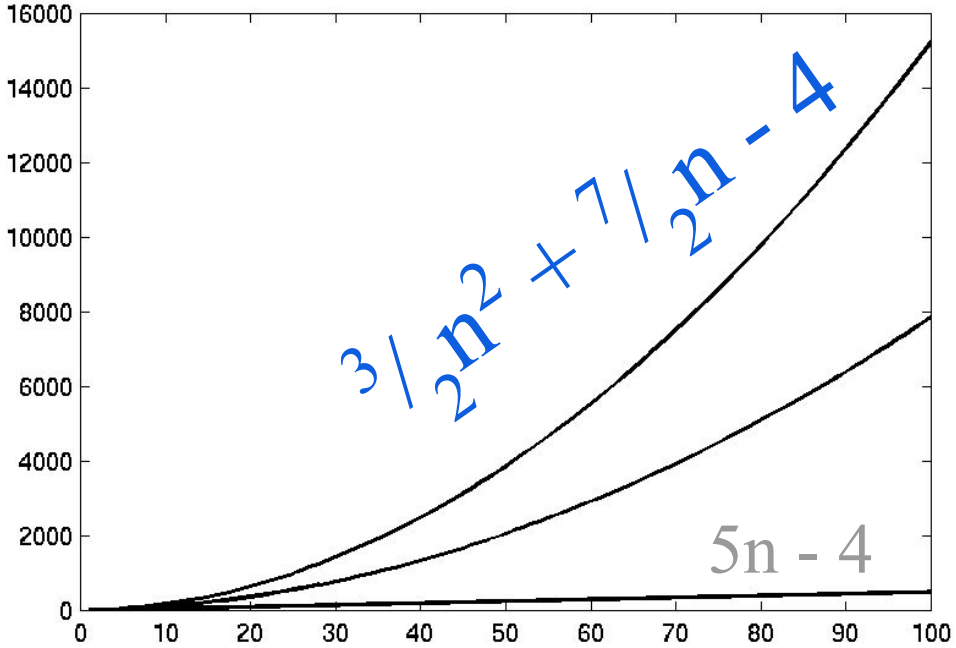
Exact Analysis: Average Case



Exact Analysis: Average Case



Exact Analysis: Average Case



We'll usually use worst case time complexity

Best combo of informative and easy to calculate

Asymptotic Notation

Exact analysis is usually more trouble than it’s worth

Asymptotic notation to the rescue! (a.k.a. big-O notation)

A formalism that we'll use to talk about the order of magnitude of how fast an equation grows

Informal definition

f(n) is O(g(n))

means

f(n) grows no faster than *g(n)*

Formal definition

f(n)* is *O(g(n)) if:

there exist positive constants, *c* and N_0 such that

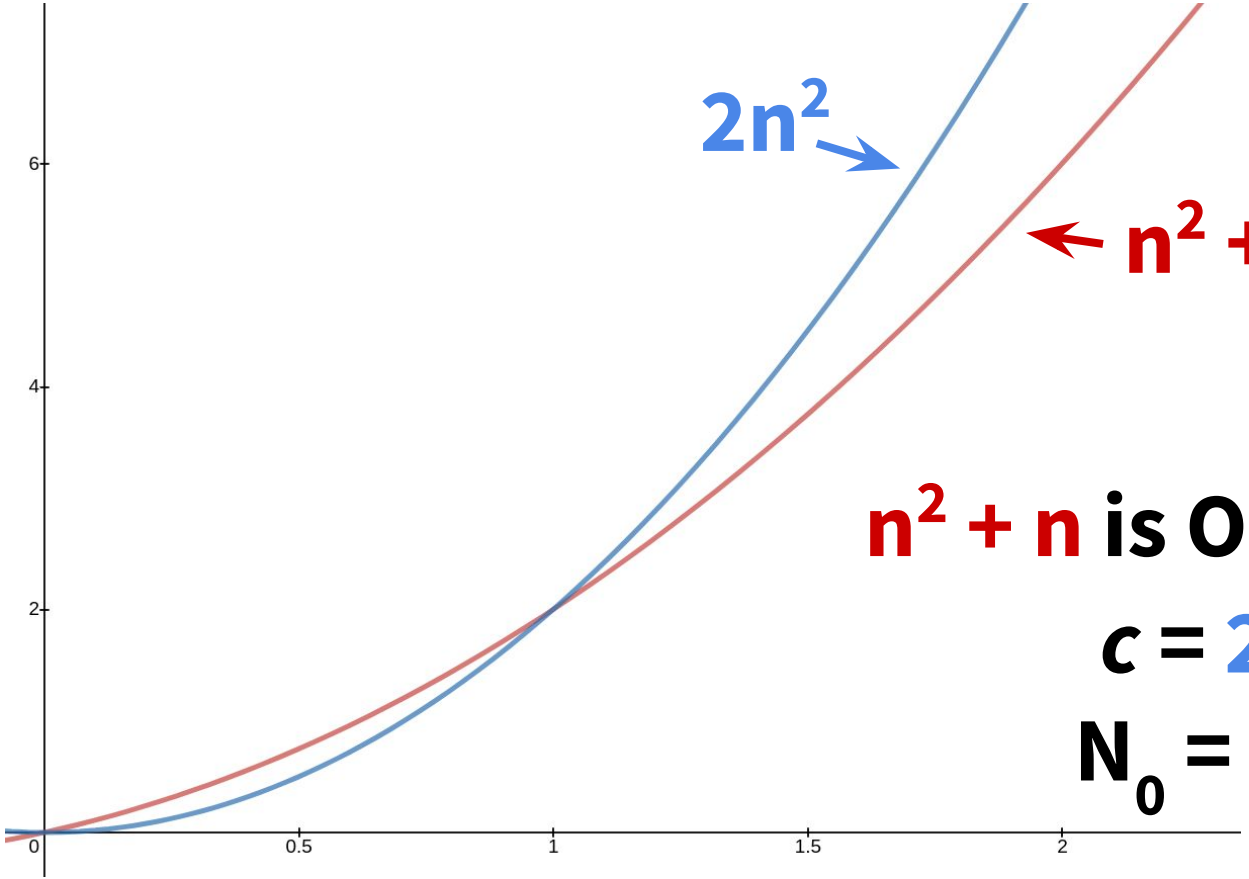
$$f(n) \leq c * g(n)$$

for all

$$N \geq N_0$$

Example

$$n^2 + n \text{ is } O(n^2)$$



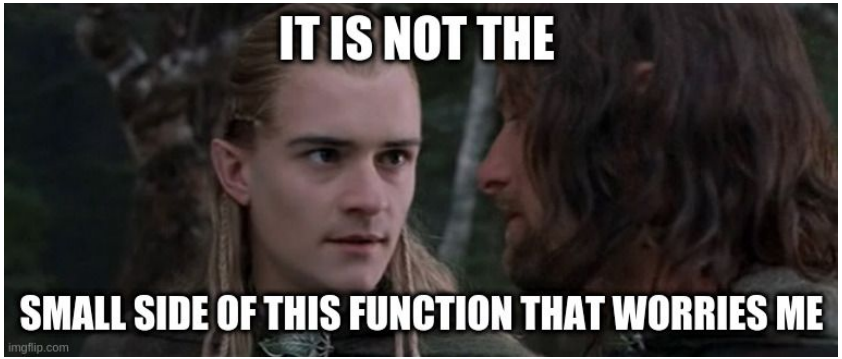
$n^2 + n$ is $O(n^2)$

$c = 2$

$N_0 = 1$

Why N_0?


LET'S NOT GO BELOW 1
'TIS A SILLY PLACE
imgflip.com


IT IS NOT THE
SMALL SIDE OF THIS FUNCTION THAT WORRIES ME
imgflip.com

Why C?

Lets us reason about rate of change

(a constant isn’t going to change, so it’s not going to affect the rate of change)

Note: f(n) is O(g(n)) does not mean that g(n) is a *tight* big-O bound on f(n)

In this class, if we want a tight big-O bound, we will ask for one (or a Θ bound)

Example: 2n is O(n)

but it’s also O(n^2), O(n!), O(nlog(n)), etc.

Other Asymptotic Notation

O(f(n))	(big Oh)	Upper bound
Ω(f(n))	(big Omega)	Lower bound
Θ(f(n))	(big Theta)	Upper and Lower Bound

Other Asymptotic Notation

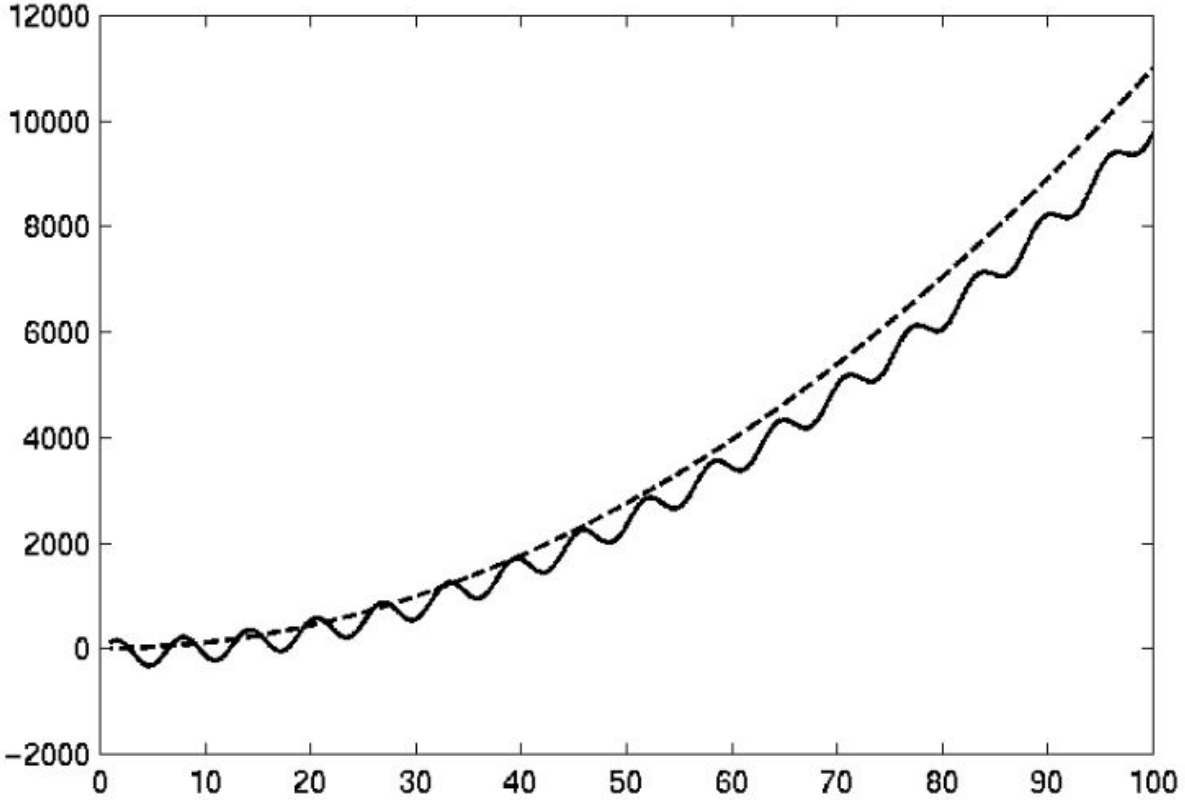
g(n) = **O(f(n))** means C × f(n) is an *Upper Bound* on g(n)

g(n) = **Ω(f(n))** means C × f(n) is a *Lower Bound* on g(n)

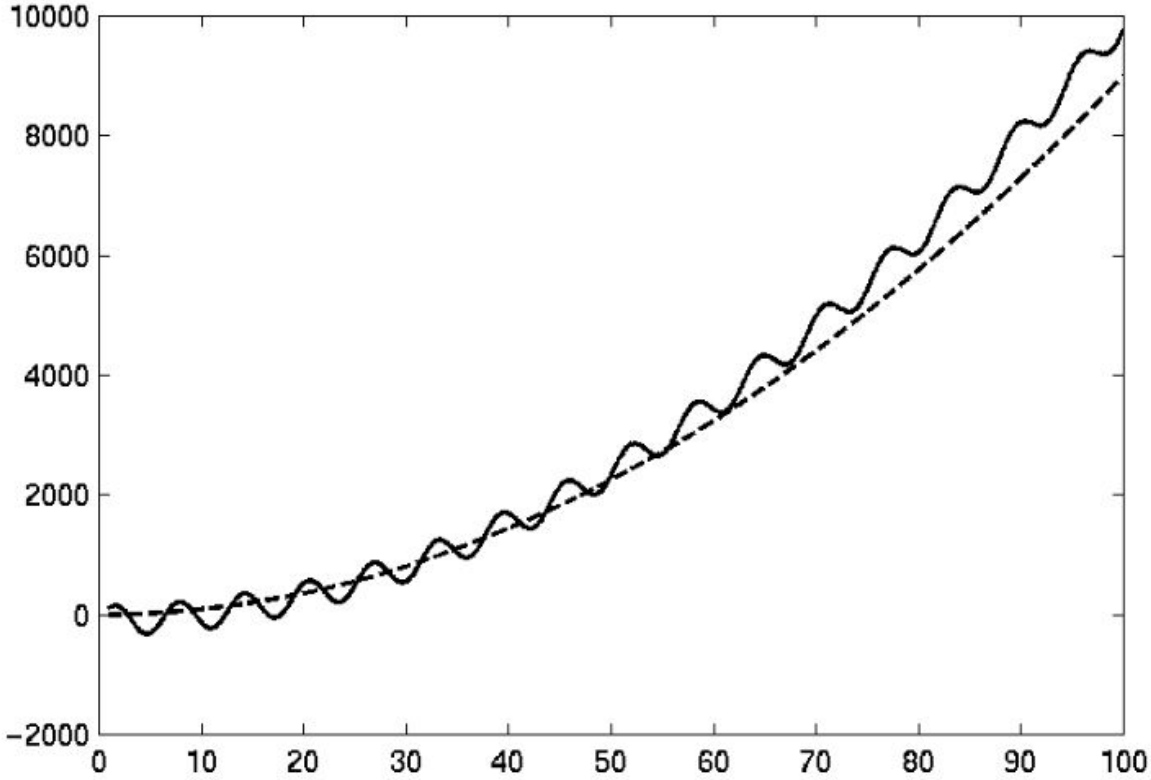
g(n) = **Θ(f(n))** means C_1 × f(n) is an *Upper Bound* on g(n)
and C_2 × f(n) is a *Lower Bound* on g(n)

These bounds hold for all inputs beyond some threshold N_0.

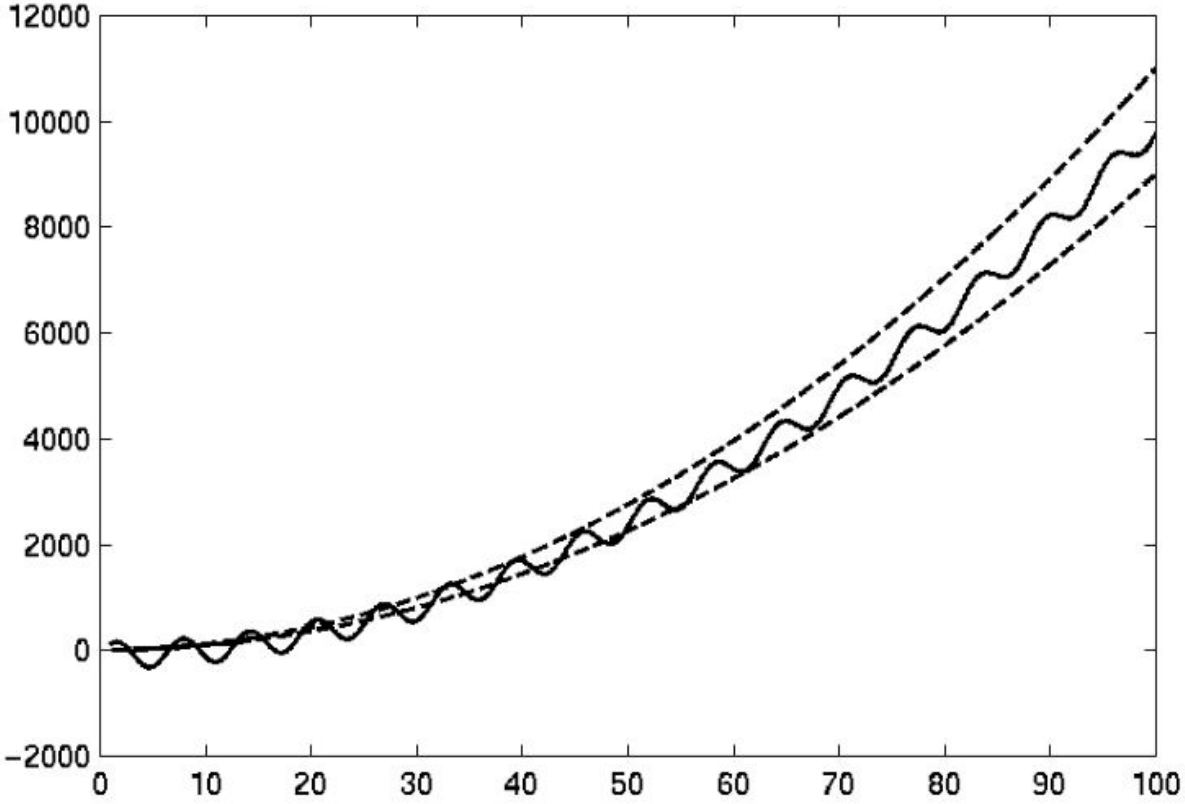
O(f(n))


12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

Ω(f(n))



Θ(f(n))


12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

Which of the following statements is True?

1. n + 3 is O(n)
2. n + 3 is O(n^2)
3. n^2 is O(n)
4. n^2 is O(2^n)

Which of the following statements is True?

1. n + 3 is Θ(n)
2. n + 3 is Θ(n^2)
3. n^2 is Θ(n)
4. n^2 is Θ(2^n)

Which of the following statements is false?

1. n + 3 is Ω(n)
2. n + 3 is Ω(n^2)
3. n^2 is Ω(n)
4. n^2 is Ω(2^n)

Common complexity classes

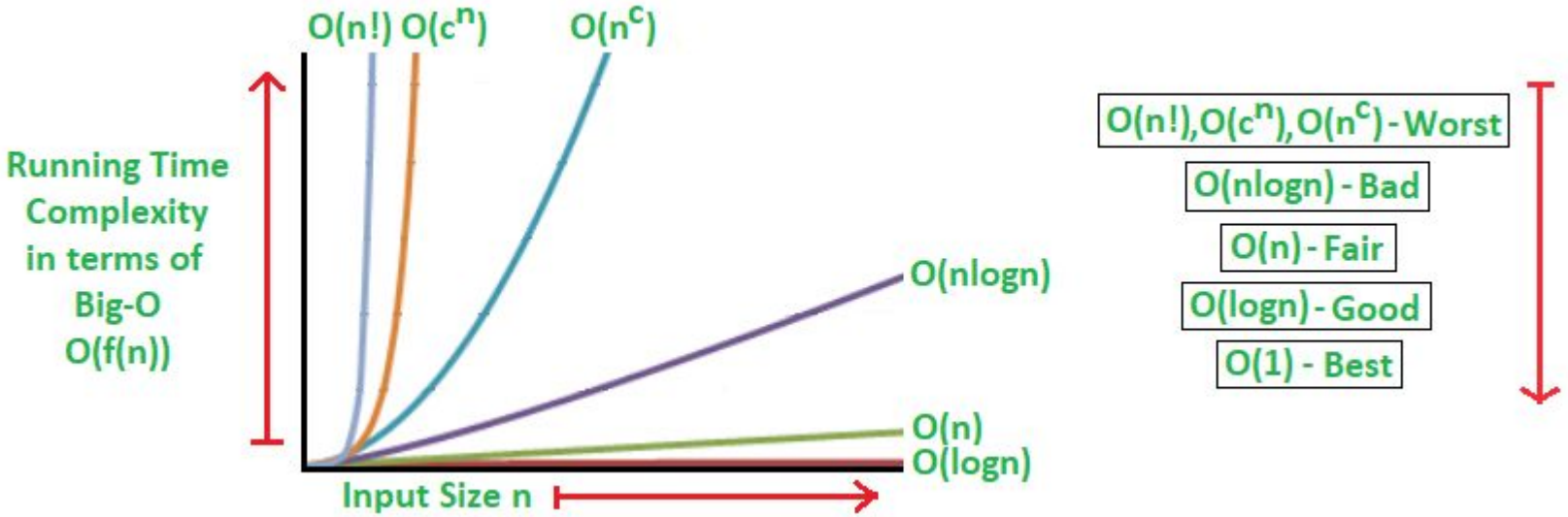
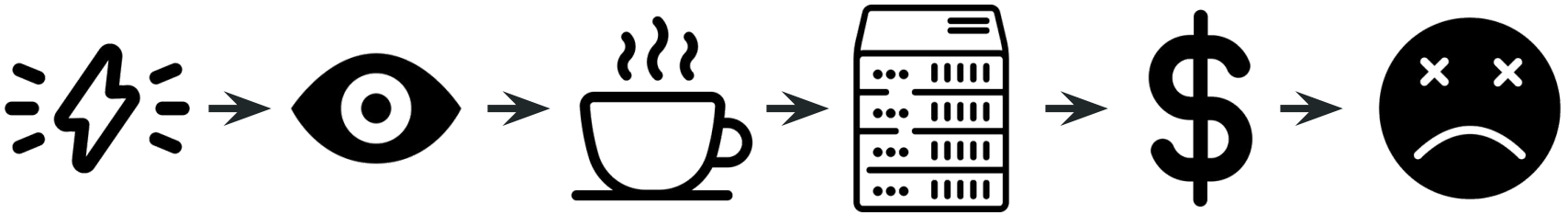


Image source: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Time differences that matter in practice



Instant

Perceptible

Long enough to go get coffee

So long that you need to throw it on a server or cluster

So long that it will be seriously expensive

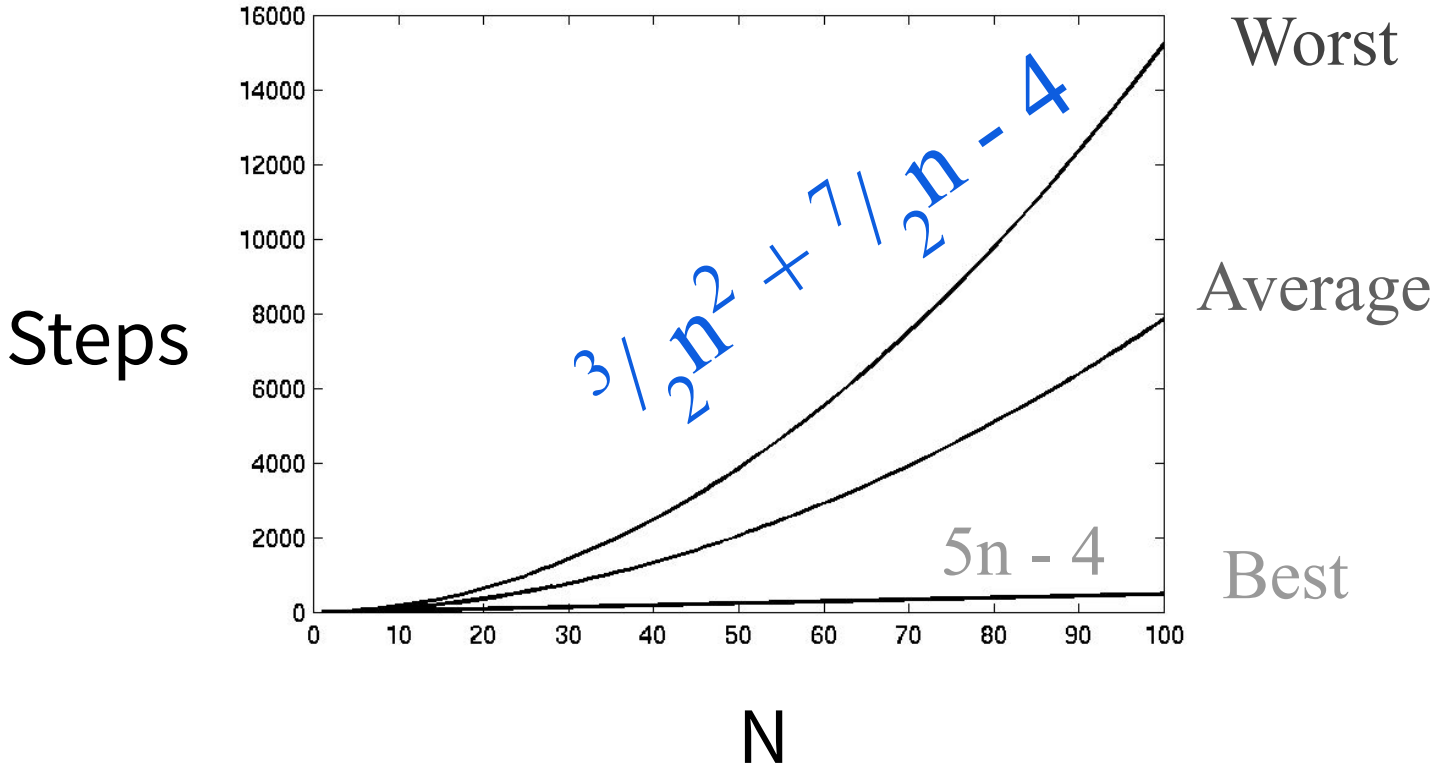
Not possible in your lifetime

Common complexity classes (if n=1 takes 1 microsecond)

Complexity	10	20	30	40	50	60
n	1×10^{-5} sec	2×10^{-5} sec	3×10^{-5} sec	4×10^{-5} sec	5×10^{-5} sec	6×10^{-5} sec
n^2	0.0001 sec	0.0004 sec	0.0009 sec	0.016 sec	0.025 sec	0.036 sec
n^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec	0.216 sec
n^5	0.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	0.001sec	1.0 sec	17.9 min	12.7 days	35.7 years	366 cent
3^n	0.59sec	58 min	6.5 years	3855 cent	2×10^{8}cent	1.3×10^{13}cent
$\log_2 n$	3×10^{-6} sec	4×10^{-6} sec	5×10^{-6} sec	5×10^{-6} sec	6×10^{-6} sec	6×10^{-6} sec
$n \log_2 n$	3×10^{-5} sec	9×10^{-5} sec	0.0001 sec	0.0002 sec	0.0003 sec	0.0004 sec

Proving asymptotic bounds

Circling back to insertion sort…



Let’s prove that...

$$\frac{3}{2}n^2 + \frac{7}{2}n - 4$$

is

$$O(n^2)$$

We must show that

$$^3/_2 n^2 + ^7/_2 n - 4 \leq c*n^2$$

for all $n > N_0$

$$^{3}/_{2}n^{2} + ^{7}/_{2}n - 4 \leq c*n^{2}$$

$$\frac{{}^{3}/_{2}n^{2} + {}^{7}/_{2}n - 4}{n^{2}} \leq c$$

$$\frac{3}{2} + \frac{7}{2} - \frac{4}{n} \leq \frac{c}{n^2}$$

What to use for C?

$$\frac{3}{2} + \frac{7}{2n} - \frac{4}{n^2} \leq c$$

It doesn't need to be the lowest possible value

$$\frac{3}{2} + \frac{\frac{7}{2}}{n} - \frac{4}{n^2} \leq c$$

It doesn't need to be the lowest possible value

$$\frac{3}{2} + \frac{7}{2} - 4 \leq c$$

$$\frac{\cdot}{n} \quad \frac{}{n^2}$$

For n > 1, the highest this term could be is 3.5

It doesn’t need to be the lowest possible value

$$3/2 + 7/2 - 4 \leq c$$

$$\overline{n^2}$$

It doesn't need to be the lowest possible value

$$3/2 + 7/2 - 4 \leq c$$

$$\frac{}{n^2}$$

The highest this term could be is 0

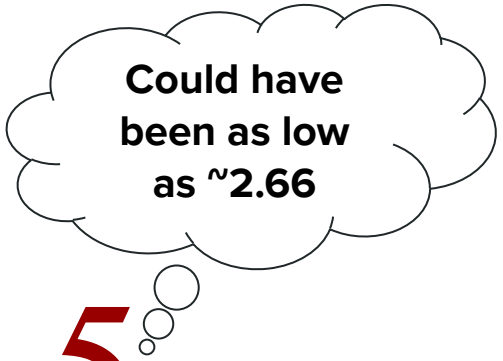
It doesn’t need to be the lowest possible value

$$^{3}/_{2} + ^{7}/_{2} \leq c$$

$$^{3}/_{2} + ^{7}/_{2} \quad \leq 5$$

$$\frac{3}{2} + \frac{7}{2} - \frac{4}{n} - \frac{}{n^2} \leq 5$$

for all n > 1



$$3/2 + \frac{7/2}{n} - \frac{4}{n^2} \leq 5$$

for all n > 1

$$\frac{3}{2} + \frac{7}{2} - \frac{4}{n} - \frac{4}{n^2} \leq 5$$

Could have been higher

for all $n > 1$.

Therefore

Worst case time complexity for insertion sort is

$$O(n^2)$$

What would have happened if we tried to prove

$$\frac{3}{2}n^2 + \frac{7}{2}n - 4$$

is

$$O(n)?$$

$$^{3}/_{2}n^{2} + ^{7}/_{2}n - 4 \leq c*n$$

$$\frac{3}{2}n + \frac{7}{2} - \frac{4}{n} \leq c$$

C is a constant so it can never be higher than all n as n goes to infinity

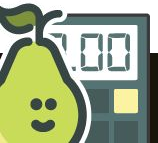
What value of C could you use to prove the following?

Best case time complexity for insertion sort $(5n - 4)$

is

$O(n)$

(use $N_0 = 1$)



Students, enter a number!

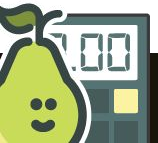
What value of C could you use to prove the following?

Worst case time complexity for insertion sort $(^3/_2 n^2 + ^7/_2 n - 4)$

is

$\Omega(n^2)$

(use $N_0 = 1$)



Students, enter a number!

Approaches to finding Big-O bounds

- For a proof, you just need valid C and N_0. Find them with:
  - **The lazy way:** Guess and check
  - **Simple algebra:** Solve for C and sum positive coefficients
  - **Calculus:** C > Lim(f(n)/g(n)) as n ➔ ∞

Approaches to finding Big-O bounds

- For a proof, you just need valid C and N_0. Find them with:
  - **The lazy way:** Guess and check
  - **Simple algebra:** Solve for C and sum positive coefficients
  - **Calculus:** $C > \text{Lim}(f(n)/g(n))$ as $n \to \infty$
- In practice, most people just eye-ball it
  - **Function:** Fastest growing term will dominate
  - **Graphing calculator:** If fastest growing term isn't obvious
  - **Code:** Look at structure (e.g. are there nested loops?)

Discussion questions

1. What does it mean if:

   $f(n) \neq O(g(n))$ *and* $g(n) \neq O(f(n))$?

2. Is $2^{n+1} = O(2^n)$?

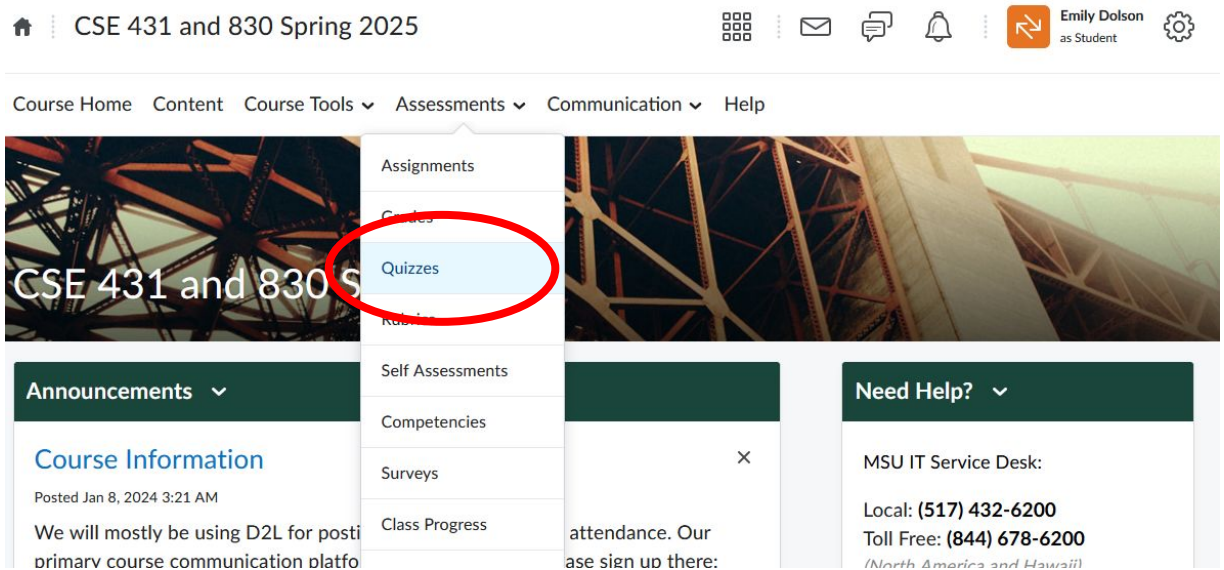
   Is $2^{2n} = O(2^n)$?

3. Does $f(n) = O(f(n))$?

4. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,

   can we say $f(n) = O(h(n))$?

Attendance break! Go to D2L, find today's quiz and answer the question



Putting it all together

Big-O, Big-Omega, and Big-Theta just describe functions.

Those functions could describe best, worst, or average case run-time of your program

Time complexities summary

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

Image credit: Robbie Weber

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

Best case: ?

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

Best case: k != 5
f(n) = ?

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

Best case: k != 5
f(n) = 4

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Best case: k != 5
f(n) = 4

Worst case: ?

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Best case: k != 5

f(n) = 4

Worst case: k == 5

g(n) = ?

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Best case: k != 5

f(n) = 4

Worst case: k == 5

g(n) = 4 + 2n

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Best case: k != 5
f(n) = 4

Worst case: k == 5
g(n) = 4 + 2n

Let’s assume k == 5 half the time.
Average case?

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Best case: k != 5
f(n) = 4

Worst case: k == 5
g(n) = 4 + 2n

Let’s assume k == 5 half the time.
Average case:
h(n) = (8 + 2n)/2 = 4 + n

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

f(n), g(n), and h(n) are just functions that model the number of steps this program will take under different circumstances

Asymptotic analysis lets us put them into complexity classes

Example

f(n) = 4 is O(?)

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

$f(n) = 4$ is $O(1)$

$C = 4, N_0 = 1$

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

$f(n) = 4$ is $O(1)$

$C = 4, N_0 = 1$

$g(n) = 4 + 2n$ is $O(?)$

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

$f(n) = 4$ is $O(1)$

$C = 4, N_0 = 1$

$g(n) = 4 + 2n$ is $O(n)$

$C = 6, N_0 = 1$

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

$f(n) = 4$ is $O(1)$

$C = 4, N_0 = 1$

$g(n) = 4 + 2n$ is $O(n)$

$C = 6, N_0 = 1$

$h(n) = 2 + n$ is $O(?)$

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

$f(n) = 4$ is $O(1)$

$C = 4, N_0 = 1$

$g(n) = 4 + 2n$ is $O(n)$

$C = 6, N_0 = 1$

$h(n) = 2 + n$ is $O(n)$

$C = 3, N_0 = 1$

Example

```
def sum_numbers(n, k):
    total = 0
    if k == 5:
        for num in range(n):
            total += num
    else:
        total = n * 5
    return total
```

We have now proved that the best case performance of this function is O(1) and the average and worst cases are both O(n)

Practice problem

Imagine you have an algorithm with a worst-case time complexity that is Θ(n^2).

Could the best case time complexity be Θ(n)?

Practice problem

Imagine you have an algorithm with a worst-case time complexity that is Θ(n^2).

Could the best case time complexity be O(n^2)?

Practice problem (be careful with this one)

Imagine you have an algorithm with a worst-case time complexity that is O(n^2).

Could the best case time complexity be O(n)?

Students choose an option

What if problem size depends on multiple variables?

What is problem size? (a.k.a. N)

- The aspect of the input that will cause algorithmic complexity to increase
  - Array length
  - Size of a number
  - etc.
- If ambiguous, define

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

n + 1 steps

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

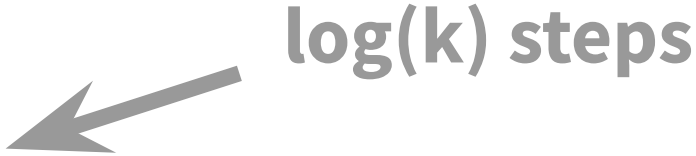
Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

log(k) + 1 steps

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```



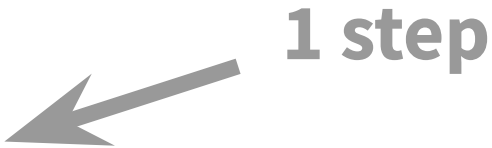
Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

log(k) steps

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```



Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

Total: 4 + 2n + 3log(k)

Example

```
def sum_numbers(n, k):
    total = 0
    for num in range(n):
        total += num
    while k > 1:
        total += k
        k /= 2
    return total
```

We can simplify this to O(n + log(k)) using asymptotic analysis

But we can’t go any farther

Practice problem - what is the time complexity of this code?

```
def sum_numbers(n, k):
    total = 0
    for num in range(n - 5):
        total += num
        curr = k
        while curr > 1:
            total += curr
            curr /= 4
    return total
```

Students, write your response!

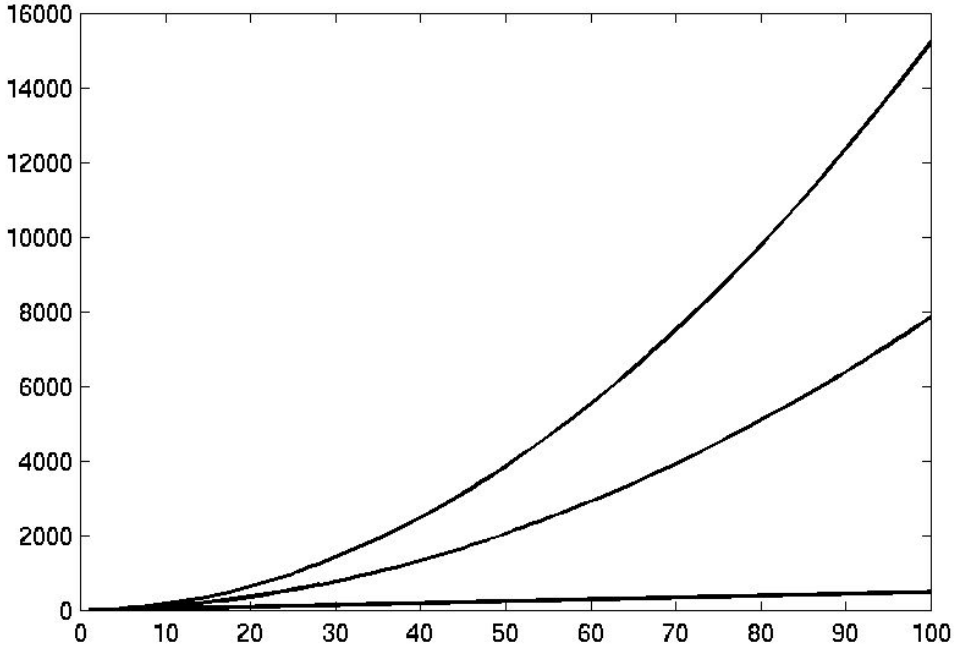
Divide and Conquer Algorithms (time complexity analysis in action!)

Divide & Conquer

- **Divide** the problem into smaller problems, often even if they are all the same.
- **Conquer** the individual pieces, recursively if they are just smaller versions of the main problem.
- **Combine** the results into a solution for the main problem.

Insertion Sort

Worst



Steps

Average

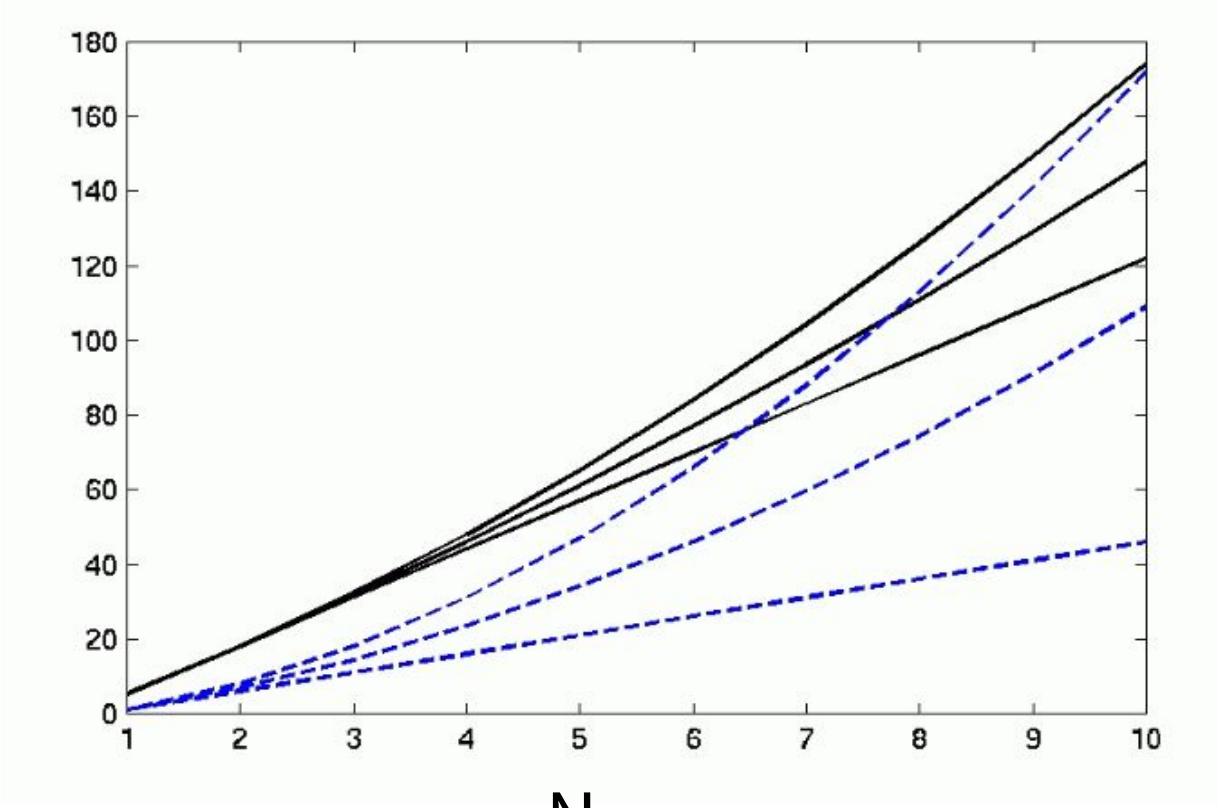
Best

N

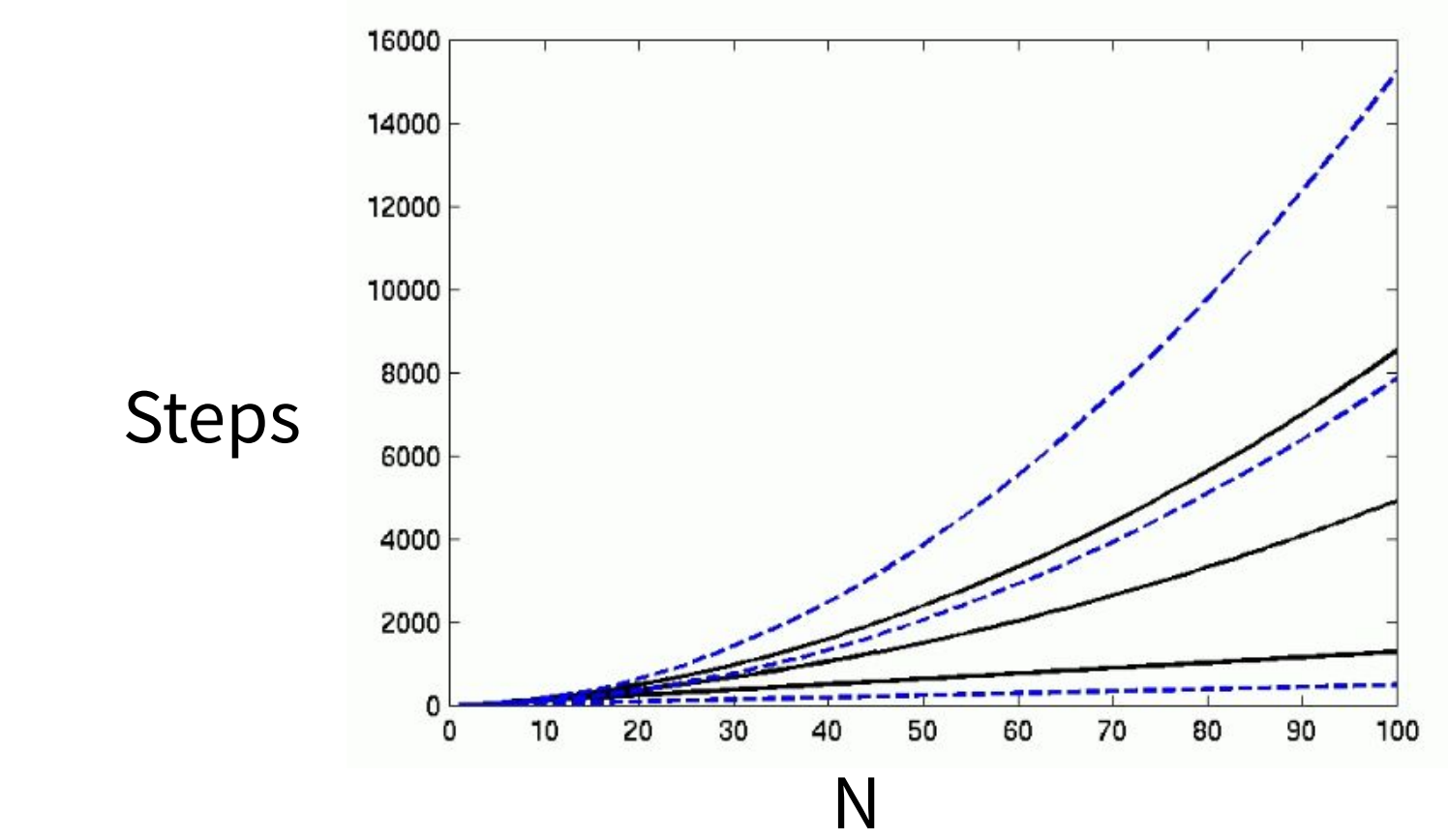
What if we split the list in half, ran insertion sort on each half, and then merged them?

Divided Insertion Sort

Steps



Divided Insertion Sort: zoomed out



Can we do better?
What if we kept
dividing in half?

Merge Sort

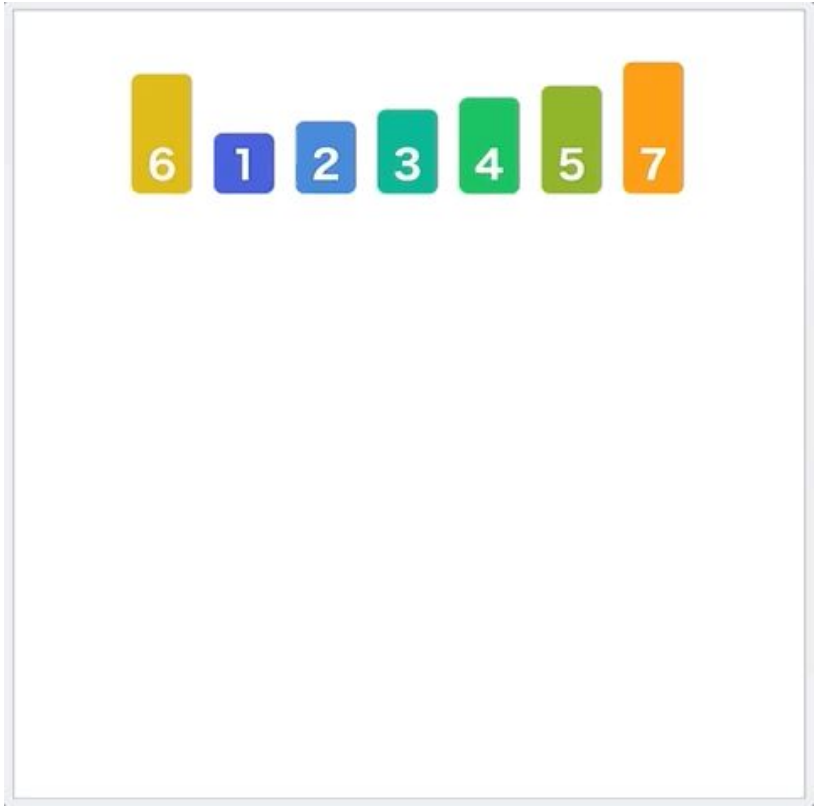
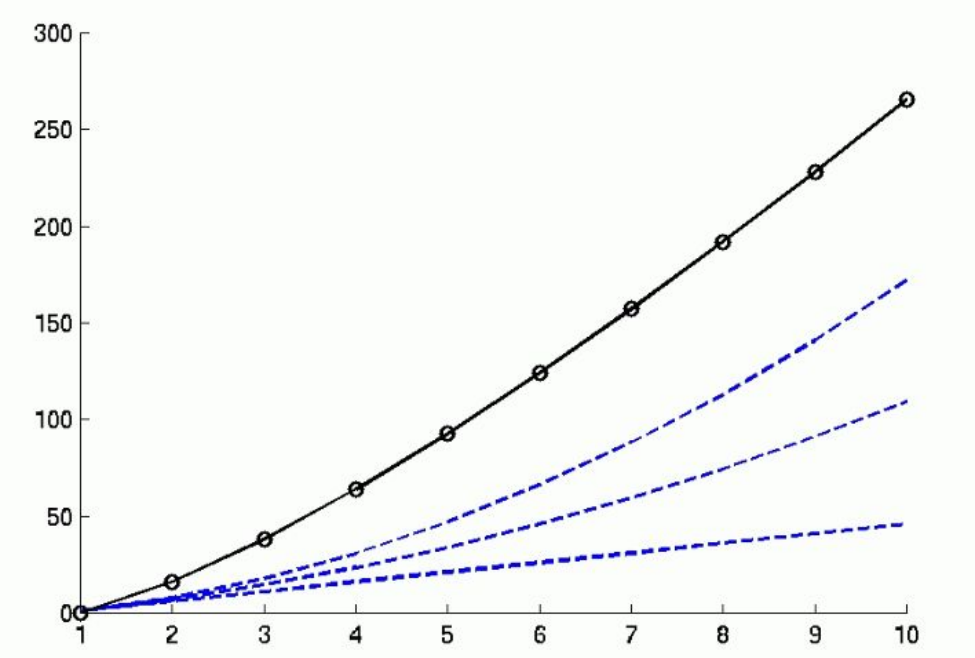


Image credit: Mahmud Ahsan

Merge sort

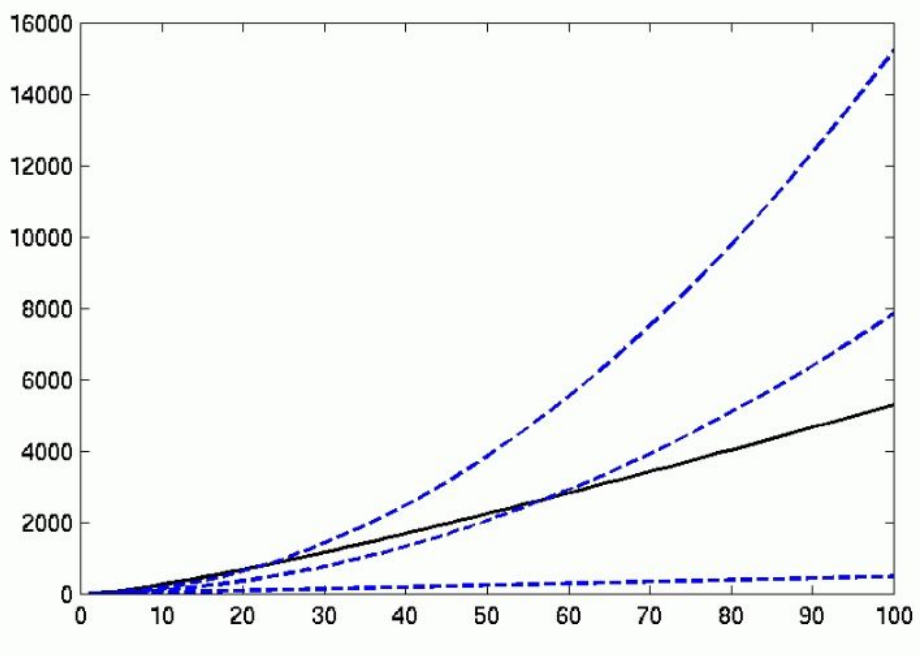
Steps



N

Merge sort

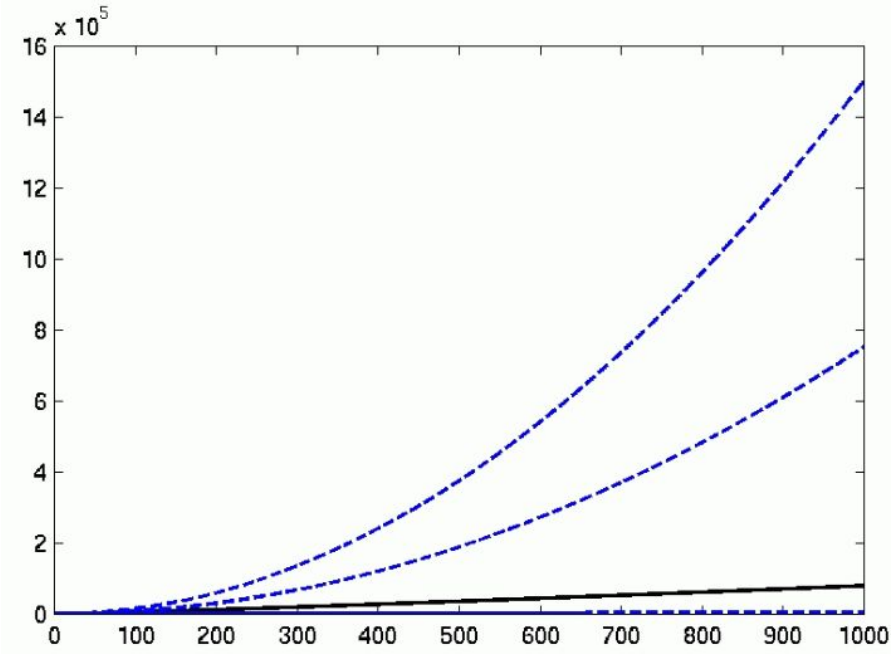
Steps



N

Merge sort

Steps



N

Merge-sort is **Θ (nlog(n))**

We'll discuss this more on Tuesday

Activities

Identify the worst-case run-times of the following code chunks

A:

```
bool HasSum100_A(std::vector<int> vals) {
  for (int i = 0; i < vals.size(); i++) {
      for (int j = 0; j < i; j++) {
        if (vals[i] + vals[j] == 100) return true;
      }
  }
  return false;
}
```

B:

```
bool HasSum100_B(std::vector<int> & vals, int i=1,
int j=0) {
  if (i >= vals.size()) return false;
  if (vals[i] + vals[j] == 100) return true;
  ++j;
  if (j == i) { ++i; j = 0; }
  return HasSum100_B(vals, i, j);
}
```

C:

```
bool HasSum100_C(std::vector<int> vals, int i=1, int
j=0) {
  if (i >= vals.size()) return false;
  if (vals[i] + vals[j] == 100) return true;
  ++j;
  if (j == i) { ++i; j = 0; }
  return HasSum100_C(vals, i, j);
}
```

D:

```
bool HasSum100_D(std::vector<int> vals) {
  std::sort(vals.begin(), vals.end());
  for (int i = 0; i < vals.size(); i++) {
      int val_needed = 100 - vals[i];
      bool found =
         std::binary_search(vals.begin(), vals.end(),
val_needed);
      if (found) return true;
  }
  return false;
}
```

Testing asymptotic notation

Use this website (https://rithmschool.github.io/function-timer-demo/) to test out how asymptotic notation works in the real world!

Choose one of the 7 functions along the top and do the following:

1. From looking at the code, try to figure out what complexity class you expect it to be in
2. Try gradually increasing the problem size and running the code to see how long it takes (be careful to increase N gently, as the website will hang if you give it too large a value)
3. Does the time complexity you observe match what you were expecting based on the code?
4. If you have time, try another function and see how the results compare!

Note: I recommend starting with a function other than logAllBinaries - it is substantially more challenging than the others.