CSCI 241: Data Structures

Lecture 2

Runtime Analysis Continued

Quiz time!!!

- On review topics.
- **Not** graded - participation credit only.
- 10 minutes

Announcements

- Course webpage link sent via Canvas.
- Slides will be posted on the webpage after each lecture.
- Lab attendance policy has been refined.
- Action item: make a GitHub account by next week’s lab.

Last Time

- We care how fast things run.
- Trade-offs between operations:
  - FilingCabinet vs PilingCabinet
- Runtime analysis: counting “primitive operations”.

“Primitive” Operations

Things the computer can do in a “fixed” amount of time.

“fixed” - doesn’t depend on the input size (n)

A non-exhaustive list:

- Get or set the value of a variable or array location
- Evaluate a simple expression
- Return from a method

```
findMax(A, n):
input: an array A of n integers
output: the maximum value in A

  currentMax = a[0] ....................1     1
  for i in 1..n:
    if currentMax < A[i]: ......1           N
      currentMax = A[i] ..........1         N

  return currentMax; ..................1    1
                                        ------
                                        2 + 2N
```

N times

```
sillyFindMax(A, n)
input: an array A of n integers
output: the maximum value in A

  for i in 0..n:
    isMax = true ......................1                        N
    // search for an element bigger than A[i]
    for j in 0..n:
      if A[j] > A[i]:...........1                               N²
        isMax = false ..........1                               N²

    if isMax:
      return A[i] ..................1                            1
                                                          ___________
                                                          1+N+2N²
```

N times (outer loop)

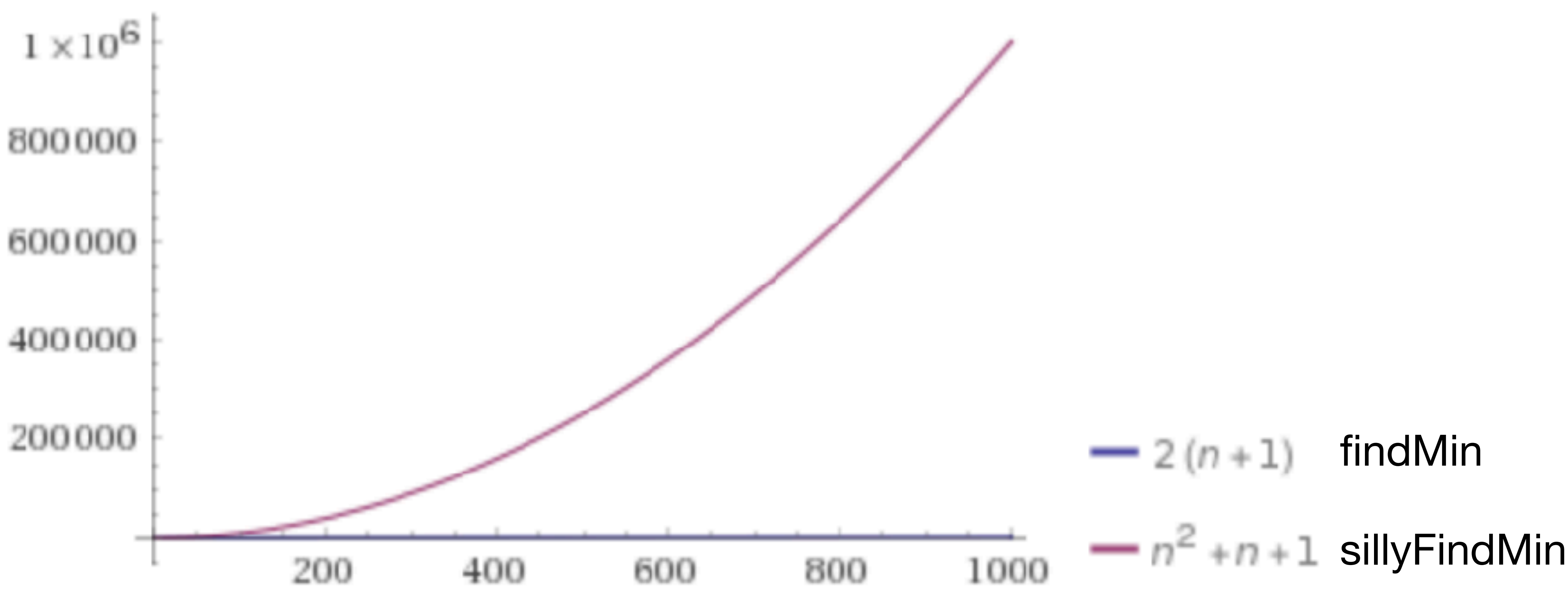
N times (inner loop)

$1+N+2N^2$

Input interpretation:

plot $2 + 2n$, $1 + n + n^2$ $\quad n = 1$ to 1000

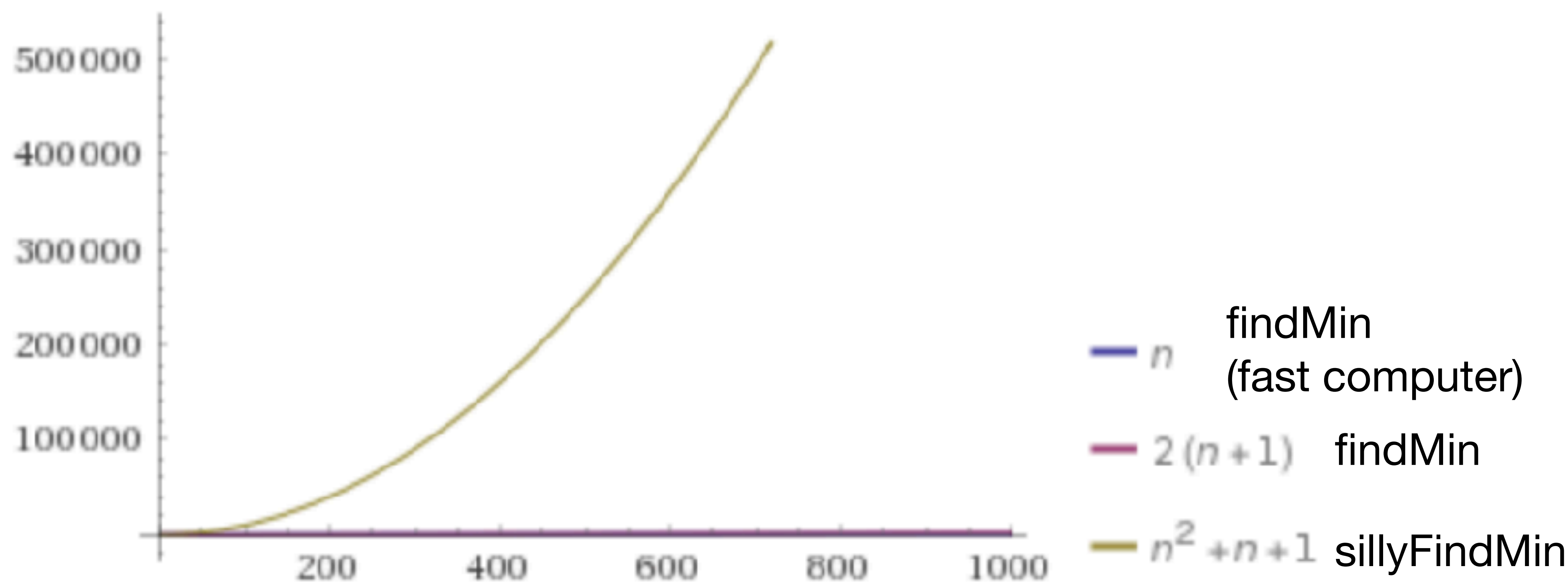
Plot:



Input interpretation:

	n	
plot	$2+2n$	$n = 1$ to 1000
	$1+n+n^2$	

Plot:



Asymptotic Runtime Complexity

- As the problem size (n) gets large:

  the difference **between** complexity classes dwarf the differences **within** them.

- To go from a count of operations to a big-O class:
  - Keep only the fastest-growing term
  - Drop any constants

4 is O(1)
600 is O(1)
n-2 is O(n)
$n^4 + 2n + 4$ is $O(n^4)$
$n! + n^{256}$ is $O(n!)$

Big-O, Informally

- "is O(n)" means "is in the same complexity class as n"
- Because constants get ignored, we can often use simple shortcuts:
    - Single loop: often O(n)
    - Two nested loops: often $O(n^2)$
    - When loop iteration variable increases as a factor of b: $O(f(\log_b N))$

Really? *any* constant?

A practical argument:

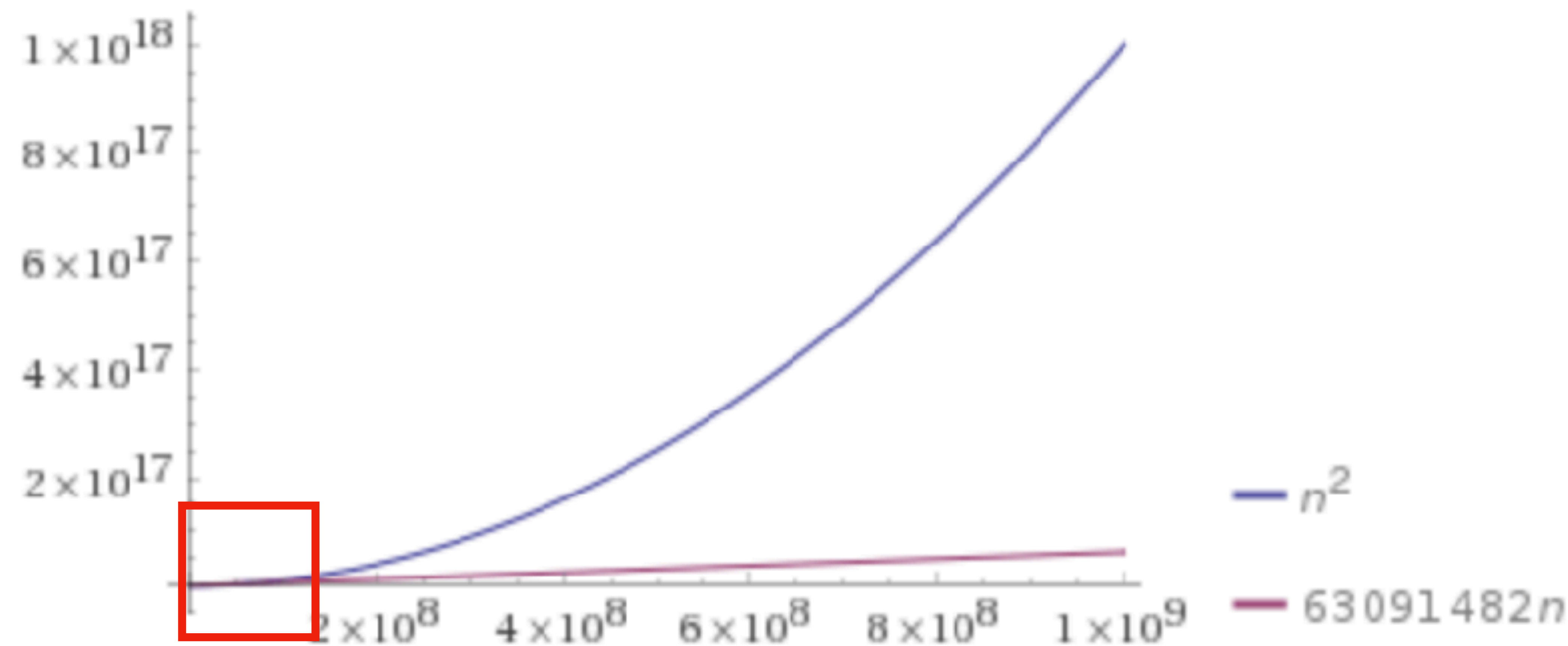
- My MacBook Pro from 2013: 3.17 **giga**FLOPs
- Fastest supercomputer as of June 2018: 200 **peta**FLOPs
- Supercomputer is 63,091,482 times faster.

Input interpretation:

plot	n^2	$n = 0$ to 1 000 000 000
	63 091 482 n	

Enlarge | Data | Customize | Plaintext | Interactive

Plot:



n^2 algorithm may be faster here!

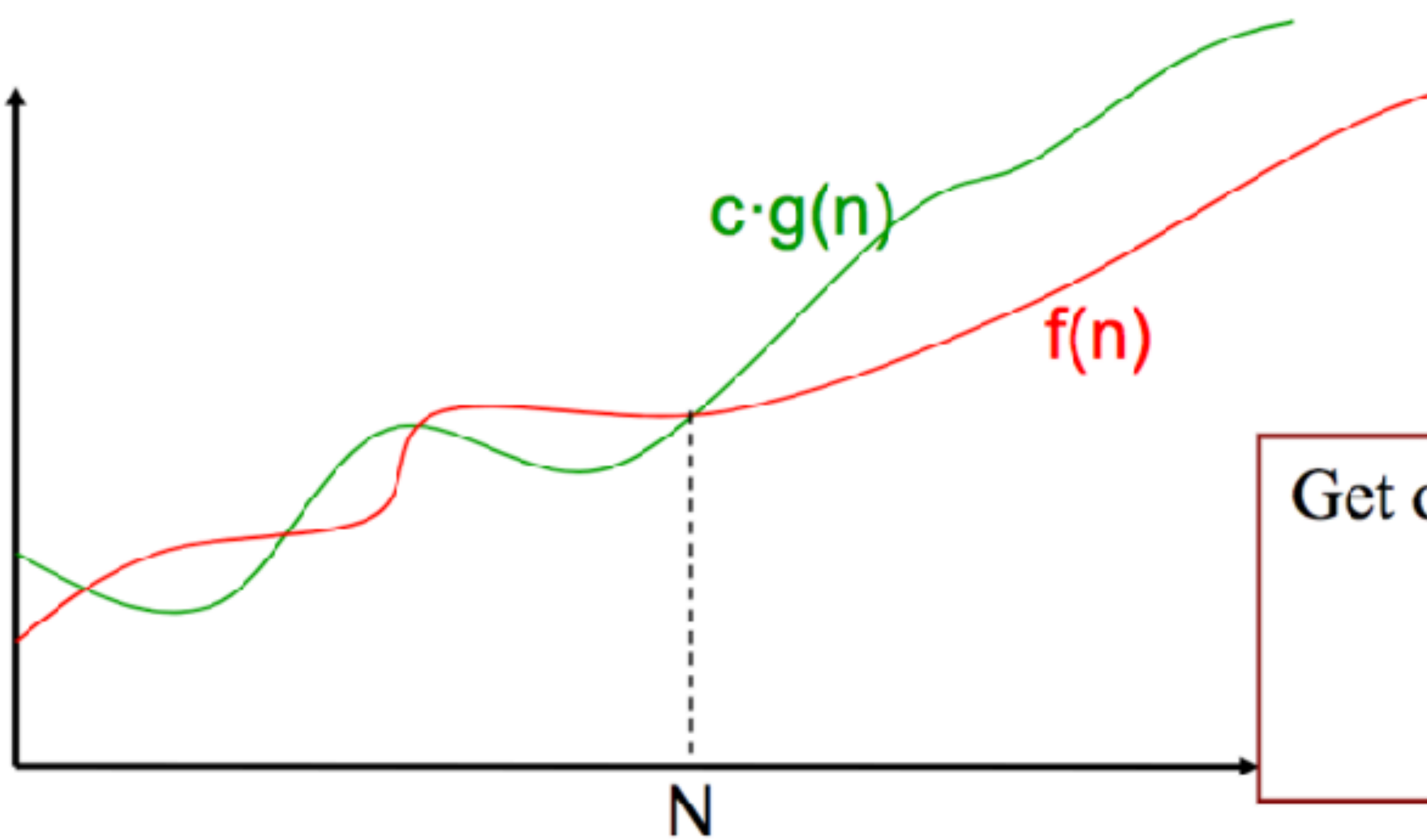
Really? *any* constant?

A theoretical argument:

Formal definition of big-O:

A function f(n) is O(g(n)) IF there exist constants c and N such that for all n larger than N, f(n) < c*g(n).

Graphical view

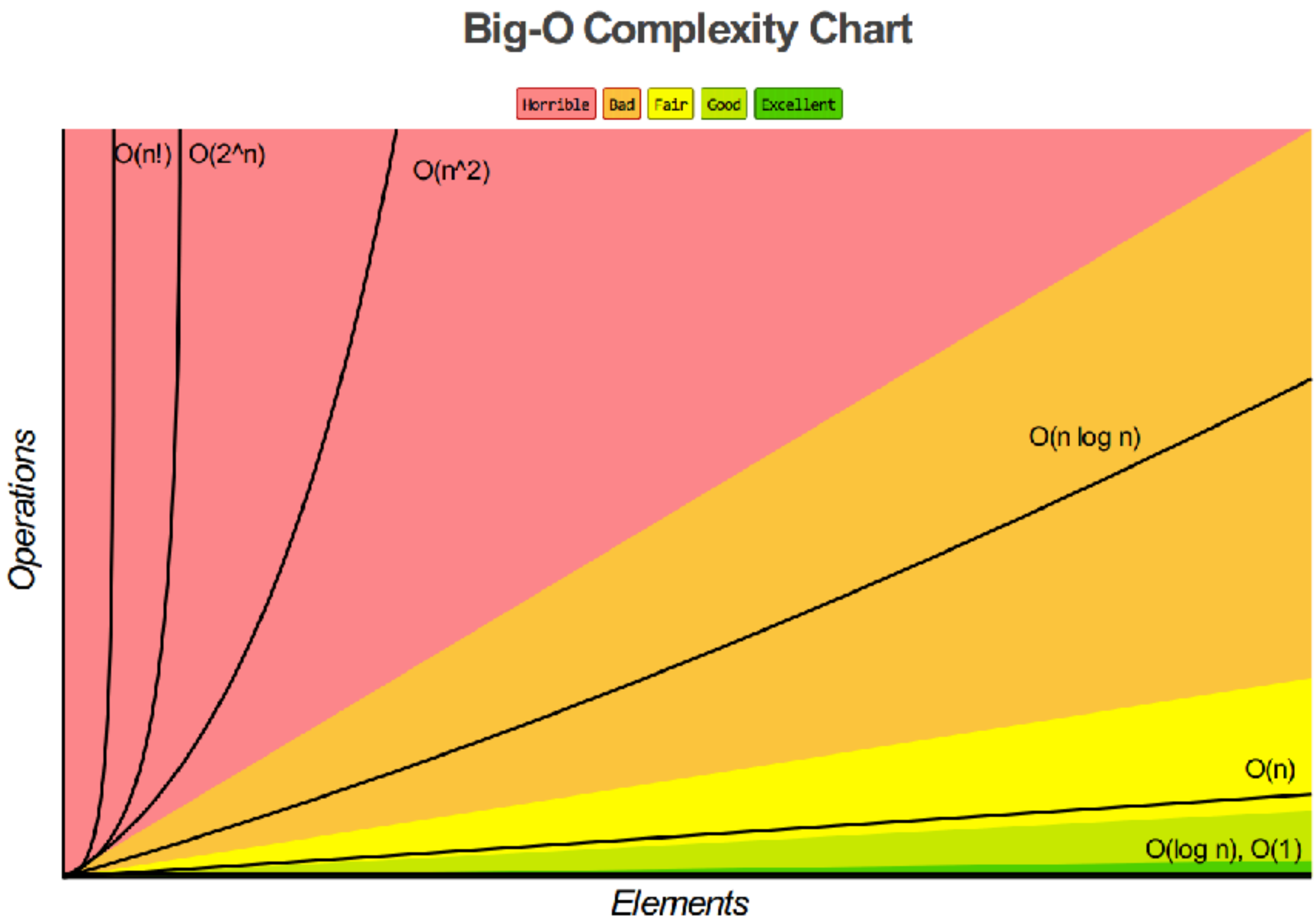


Get out far enough (for $n \geq N$) f(n) is at most c·g(n)

O(1)

- Primitive operations:
  - Get or set the value of a variable or array location
  - Evaluate a simple expression
  - Return from a method
- Why are these all O(1)?

Common Complexities



Big-O: Example

```
Alg1(n):
   sum = 0;
   for i = 0..n:
      for j = 1..100:
         sum += i*j
   return sum;
```

A: O(1)
B: O(n)
C: $O(n^2)$
D: $O(n^3)$