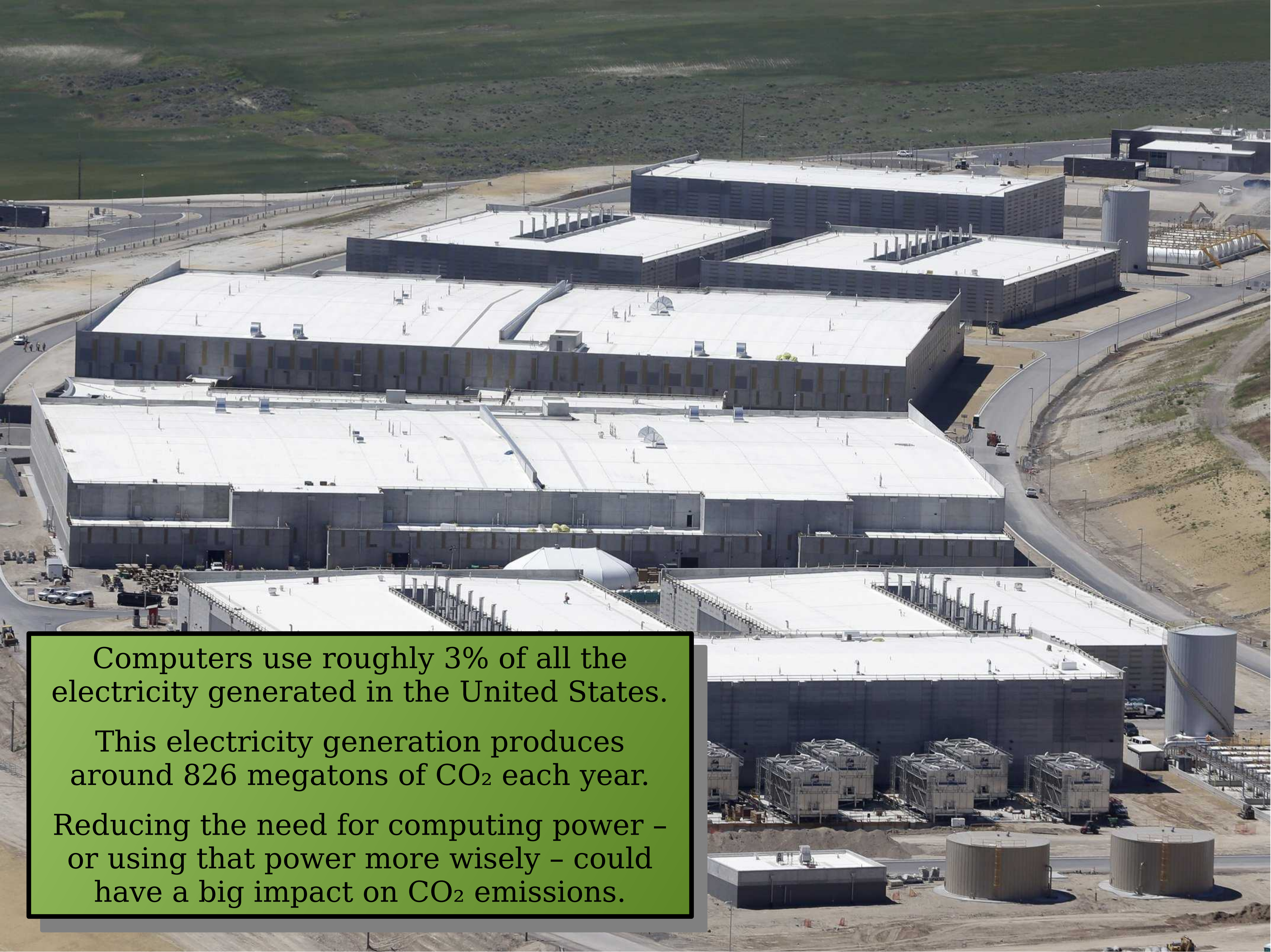
# Algorithmic Analysis and Sorting

Part One



Computers use roughly 3% of all the electricity generated in the United States.

This electricity generation produces around 826 megatons of $CO_2$ each year.

Reducing the need for computing power – or using that power more wisely – could have a big impact on $CO_2$ emissions.

## ***Fundamental Question:***

How can we compare solutions to problems?

One Idea: ***Runtime***

# Runtime is Noisy

- Runtime is highly sensitive to ***which computer you're using***.
- Runtime is highly sensitive to ***which inputs you're testing***.
- Runtime is highly sensitive to ***external factors***.

```
bool linearSearch(const string& str, char ch) {
  for (int i = 0; i < str.length(); i++) {
    if (str[i] == ch) {
      return true;
    }
  }
  return false;
}
```

Work Done: At most $k_0 n + k_1$

# Big Observations

- Don't need to explicitly compute these constants.
  - Whether runtime is $4n + 10$ or $100n + 137$, runtime is still proportional to input size.
  - Can just plot the runtime to obtain actual values.
- Only the dominant term matters.
  - For both $4n + 1000$ and $n + 137$, for very large $n$ most of the runtime is explained by $n$.
- Is there a concise way of describing this?

# Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$

For the mathematically inclined:

$$f(n) = O(g(n)) \text{ if}$$
$$\exists n_0 \in \mathbb{R}.\ \exists c \in \mathbb{R}.\ \forall n \geq n_0.\ f(n) \leq c|g(n)|$$

# Algorithmic Analysis with Big-O

```
double average(const Vector<int>& vec) {
  double total = 0.0;
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }

  return total / vec.size();
}
```

$$\mathbf{O}(\boldsymbol{n})$$

# A More Interesting Example

```
bool linearSearch(const string& str, char ch) {
  for (int i = 0; i < str.length(); i++) {
    if (str[i] == ch) {
      return true;
    }
  }
  return false;
}
```

How do we analyze this?

# Types of Analysis

- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for “sleeping well at night.”
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!

# Types of Analysis

- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for “sleeping well at night.”
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!

# A More Interesting Example

```
bool linearSearch(const string& str, char ch) {
  for (int i = 0; i < str.length(); i++) {
    if (str[i] == ch) {
      return true;
    }
  }
  return false;
}
```

**O(*n*)**

# Determining if a Character is a Letter

```
bool isAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
           (ch >= 'a' && ch <= 'z');
}
```

**O(1)**

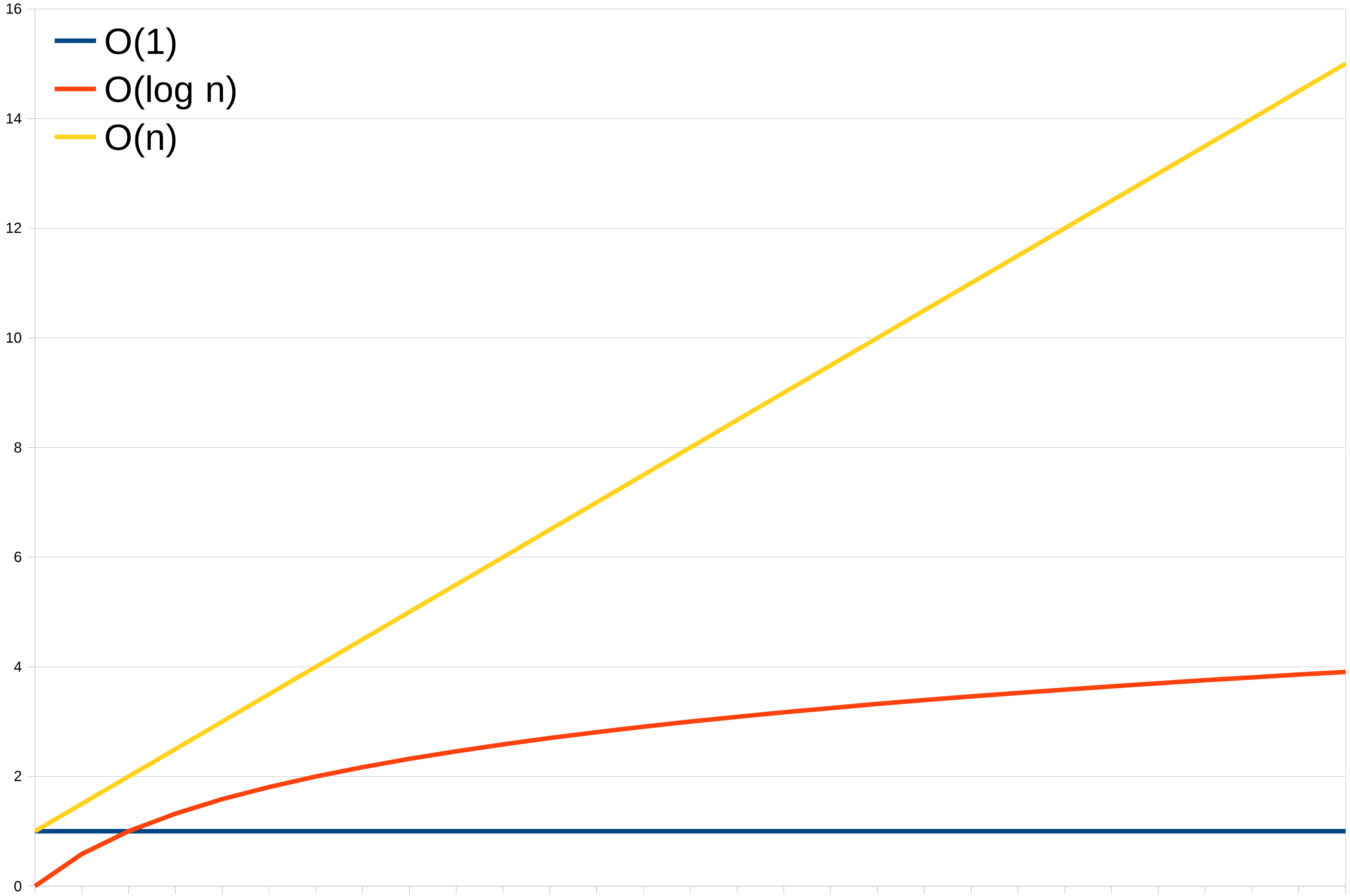
# What Can Big-O Tell Us?

- Long-term behavior of a function.
  - If algorithm A has runtime O($n$) and algorithm B has runtime O($n^2$), for very large inputs algorithm A will always be faster.
  - If algorithm A has runtime O($n$), for large inputs, doubling the size of the input doubles the runtime.

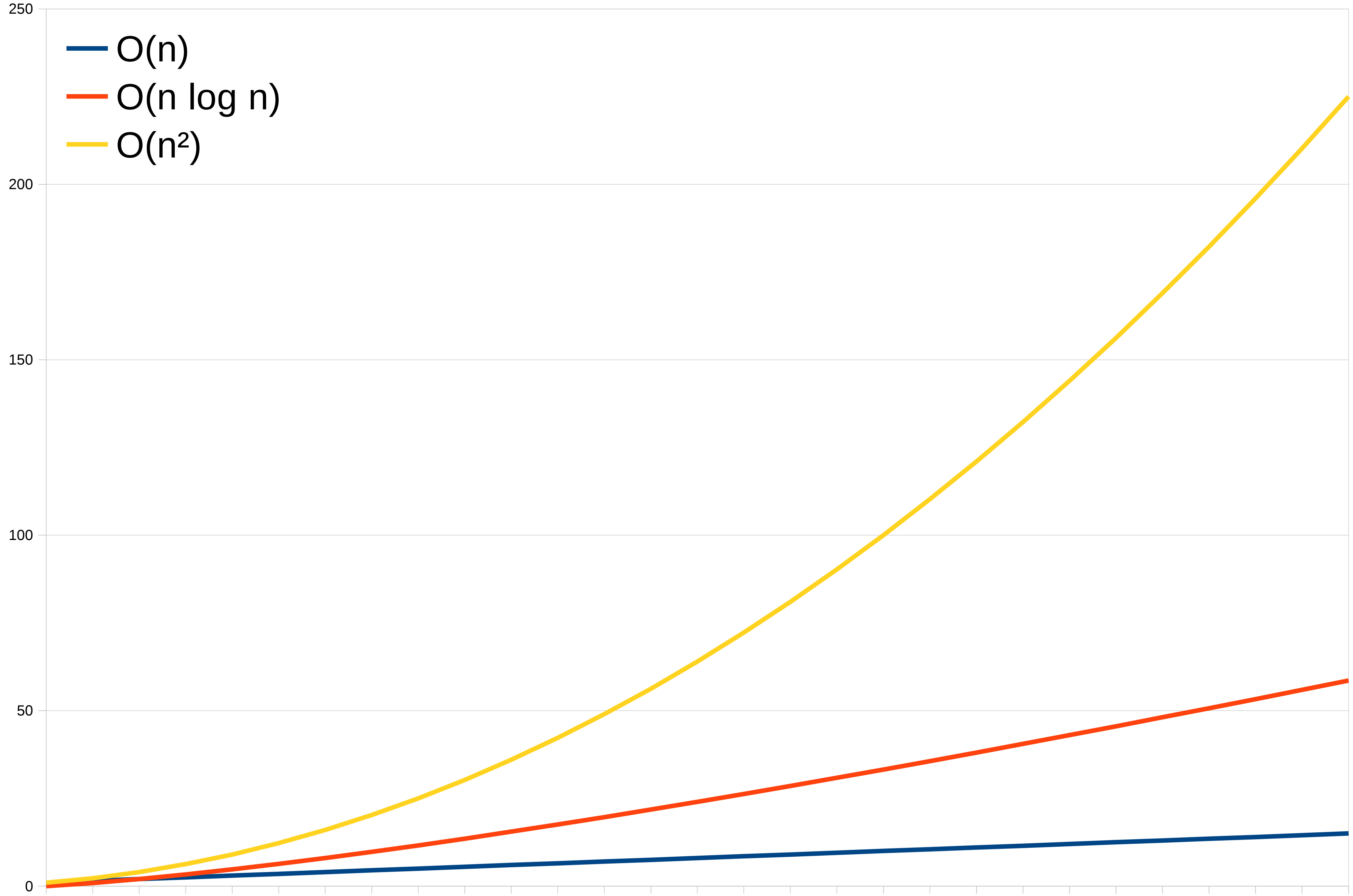
# What *Can't* Big-O Tell Us?

- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$

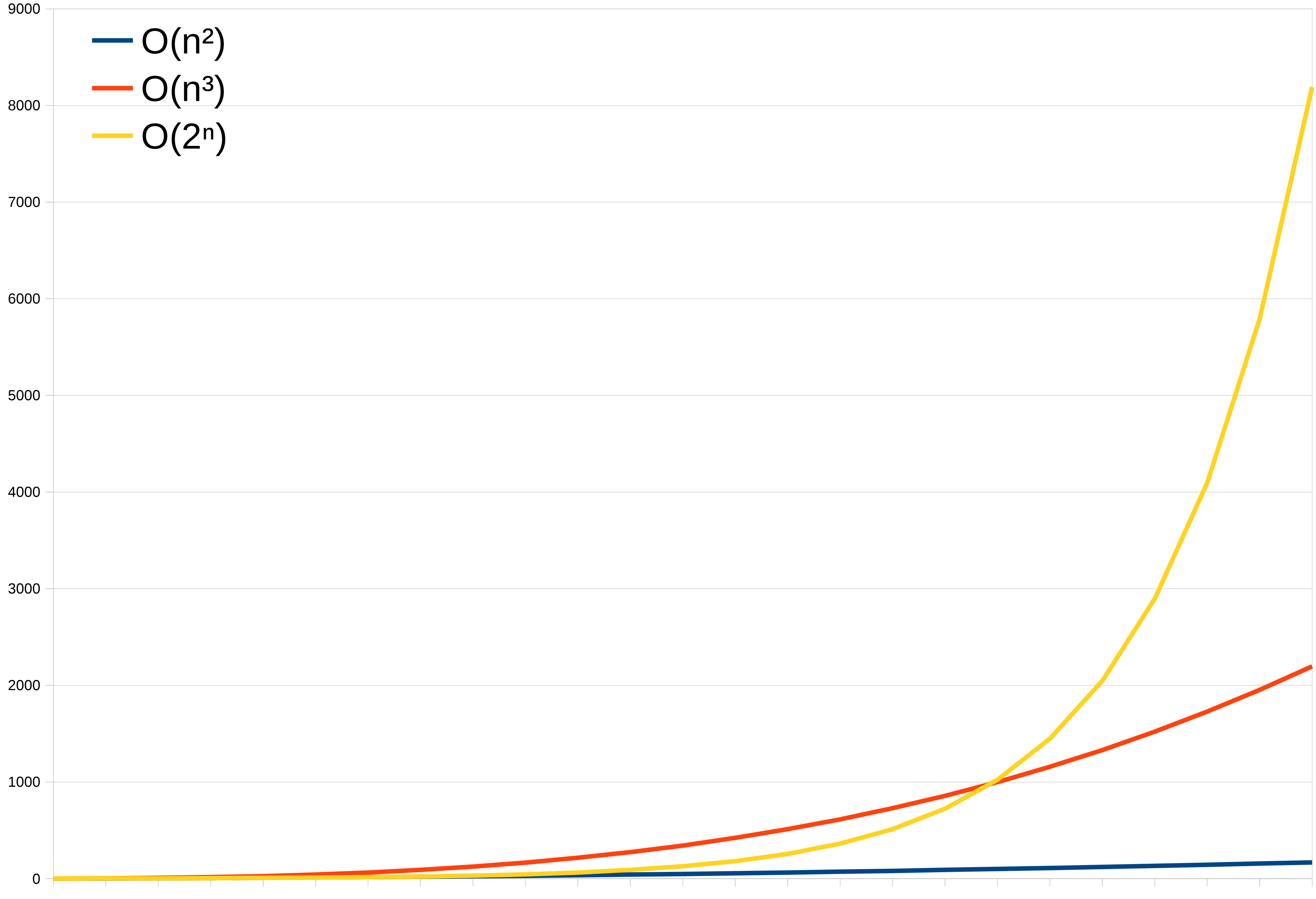
# Growth Rates, Part One



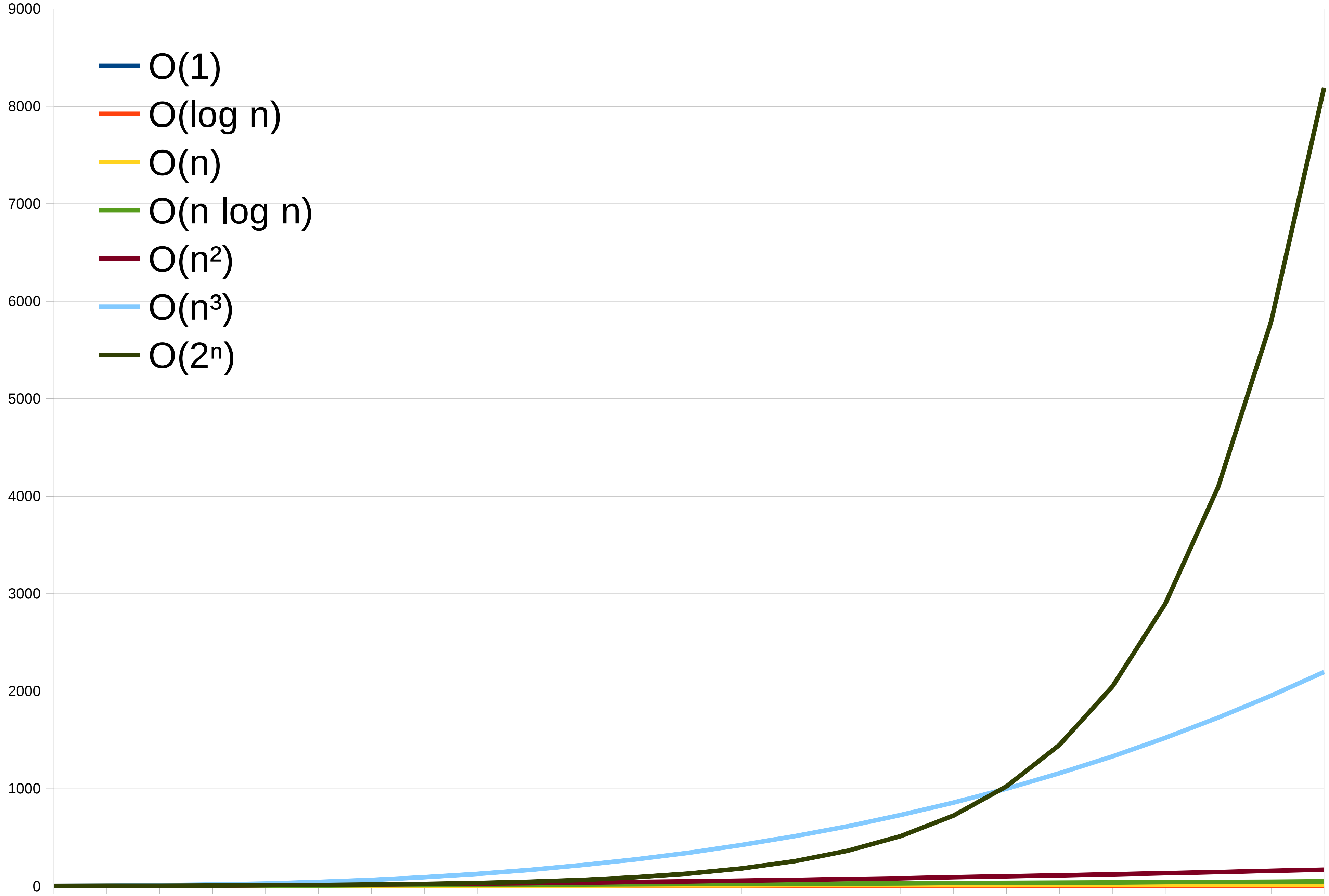
# Growth Rates, Part Two



# Growth Rates, Part Three



# To Give You A Better Sense...



# Comparison of Runtimes

***(assuming 1 operation = 1 nanosecond)***

| Size | 1 | $\log_2 n$ | n | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|---|
| 1000 | 1ns | 9.966ns | 1μs | 9.966μs | 1ms | 1s | $3.4\times10^{284}$ yr |
| 2000 | 1ns | 10.966ns | 2μs | 21.932μs | 4ms | 8s | Just… wow. |
| 3000 | 1ns | 11.551ns | 3μs | 34.652μs | 9ms | 27s | |
| 4000 | 1ns | 11.966ns | 4μs | 47.863μs | 16ms | 1.067min | |
| 5000 | 1ns | 12.288ns | 5μs | 61.439μs | 25ms | 2.083min | |
| 6000 | 1ns | 12.551ns | 6μs | 75.304μs | 36ms | 3.6min | |
| 7000 | 1ns | 12.773ns | 7μs | 89.412μs | 49ms | 5.717min | |
| 8000 | 1ns | 12.966ns | 8μs | 103.726μs | 64ms | 8.533min | |
| 9000 | 1ns | 13.136ns | 9μs | 118.221μs | 81ms | 12.15min | |
| 10000 | 1ns | 13.288ns | 10μs | 132.877μs | 100ms | 16.667min | |
| 11000 | 1ns | 13.425ns | 11μs | 147.677μs | 121ms | 22.183min | |
| 12000 | 1ns | 13.551ns | 12μs | 162.609μs | 144ms | 28.8min | |
| 13000 | 1ns | 13.666ns | 13μs | 177.661μs | 169ms | 36.617min | |
| 14000 | 1ns | 13.773ns | 14μs | 192.824μs | 196ms | 45.733min | |

# Summary of Big-O

- A means of describing the growth rate of a function.
- Ignores all but the leading term.
- Ignores constants.
- Allows for quantitative ranking of algorithms.
- Allows for quantitative reasoning about algorithms.

Time-Out for Announcements!

# Assignment 4

- Assignment 4 (***Recursion to the Rescue!***) goes out today. It's due next Friday, February 17th at the start of class.
  - We've pushed the due date for this assignment back a class period to give you a little more breathing room.
  - You're encouraged to work in pairs on this assignment. These problems are great to discuss in a group.
  - ***Start early!*** There's a suggested timetable on the front of the assignment handout that we think will help you keep on track.
  - ***Be careful about taking late days here***. The midterm is on the Tuesday after this assignment is due.
- Anton will be holding YEAH hours tonight from 7PM – 8PM in room 420-040. Highly recommended, as always!
- Assignment 3 was due today at 11:30. Feel free to use a late day if you need to, though keep in mind that you'll want to get a jump on Assignment 4.

# Girl Code @Stanford

- This summer, I'll be running our fifth iteration of Girl Code @Stanford from July 10th – July 21st.
- We invite high-school girls (primarily from low- to middle-income schools in majority-minority areas) to come to campus for two weeks to learn CS, meet researchers, and talk to folks from industry.
- We're looking for Stanford students to serve as "Workshop Assistants" during the program. We pay competitively (roughly $3,000 over two weeks).
- Interested? Learn more and apply using this link:

***https://goo.gl/forms/icYcRiX8PTgoVJ0n1***

All current Stanford students are invited to apply. Feel free to forward this link around!

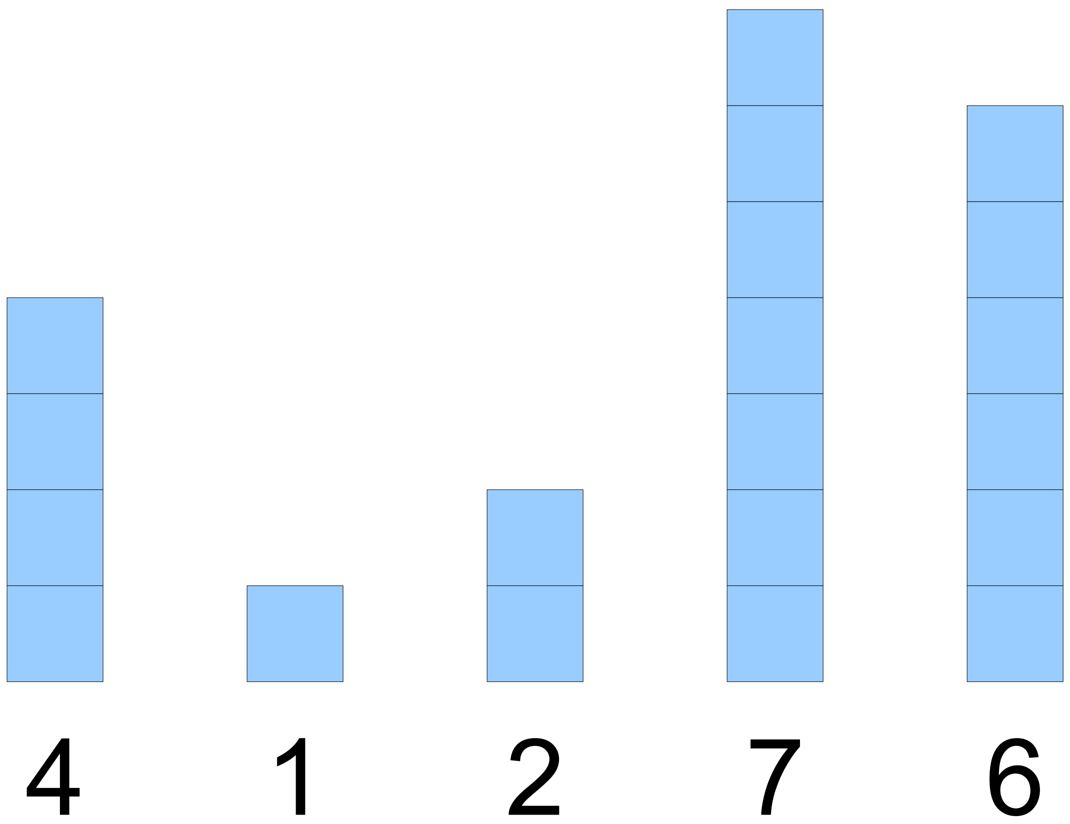
Back to CS106B!

# Sorting Algorithms

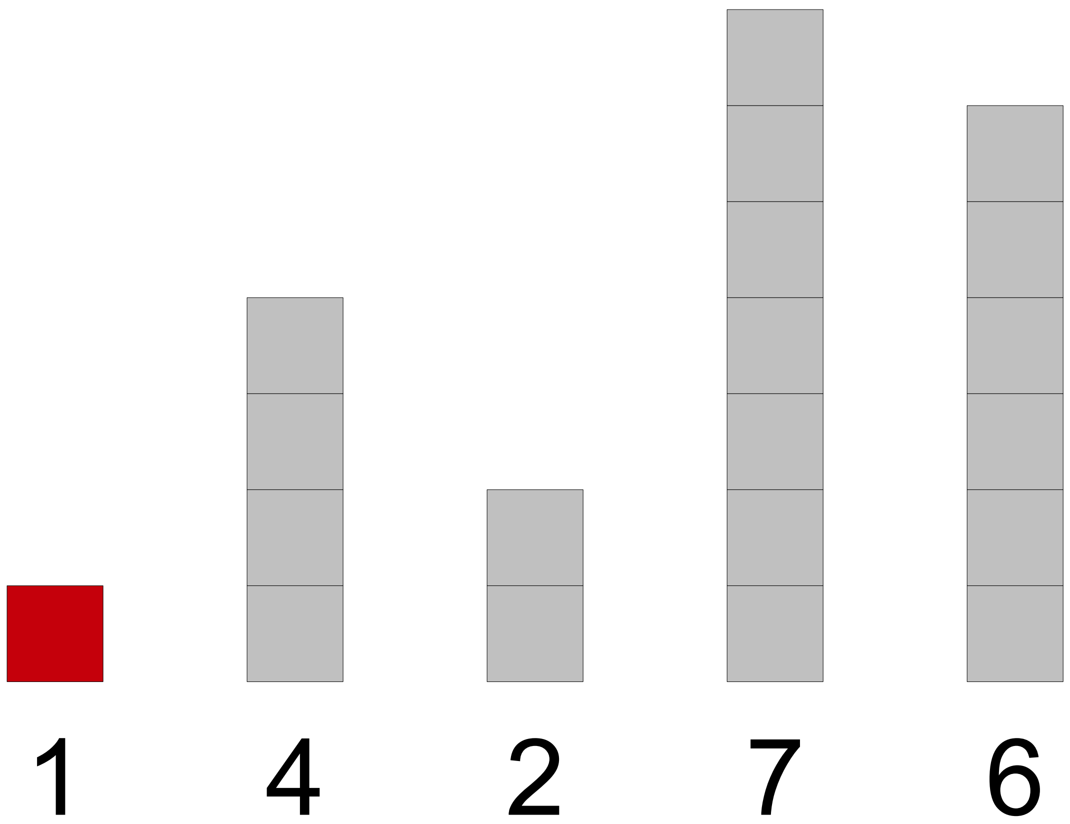
# The Sorting Problem

- Given a list of elements, sort those elements in ascending order.
- There are ***many*** ways to solve this problem.
- What is the ***best*** way to solve this problem?
- We'll use big-O to find out!

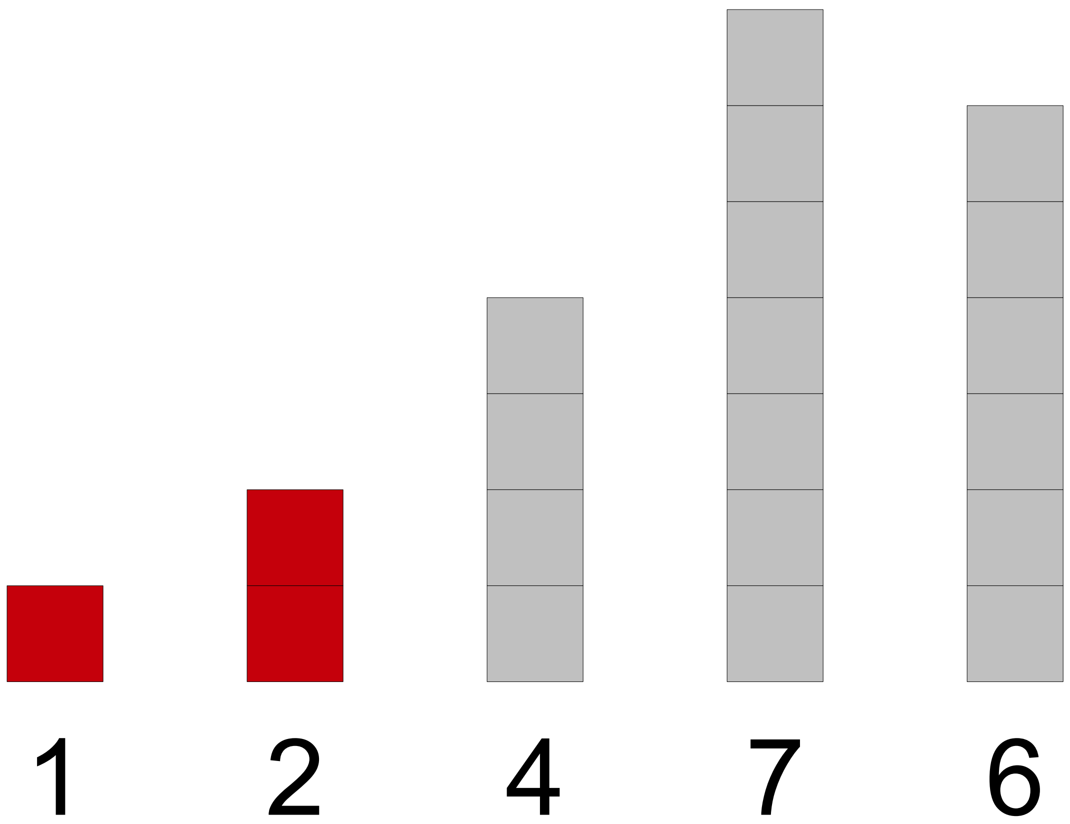
# An Initial Idea: ***Selection Sort***



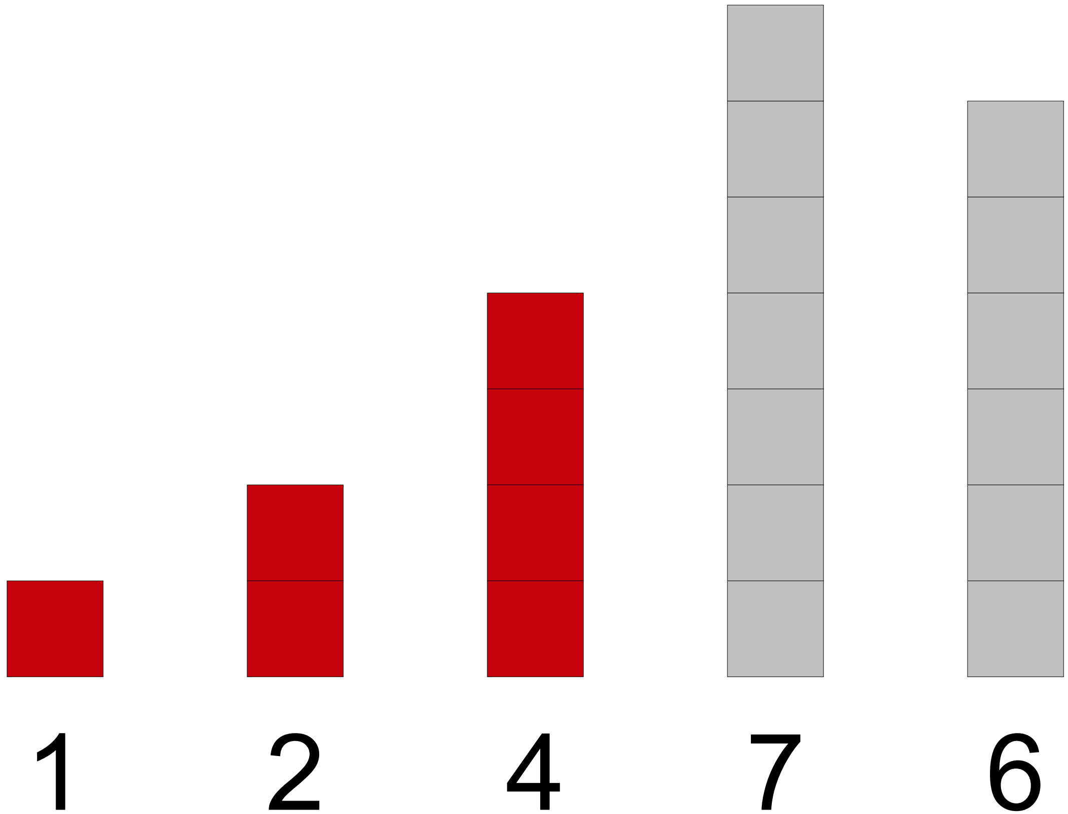
# An Initial Idea: ***Selection Sort***



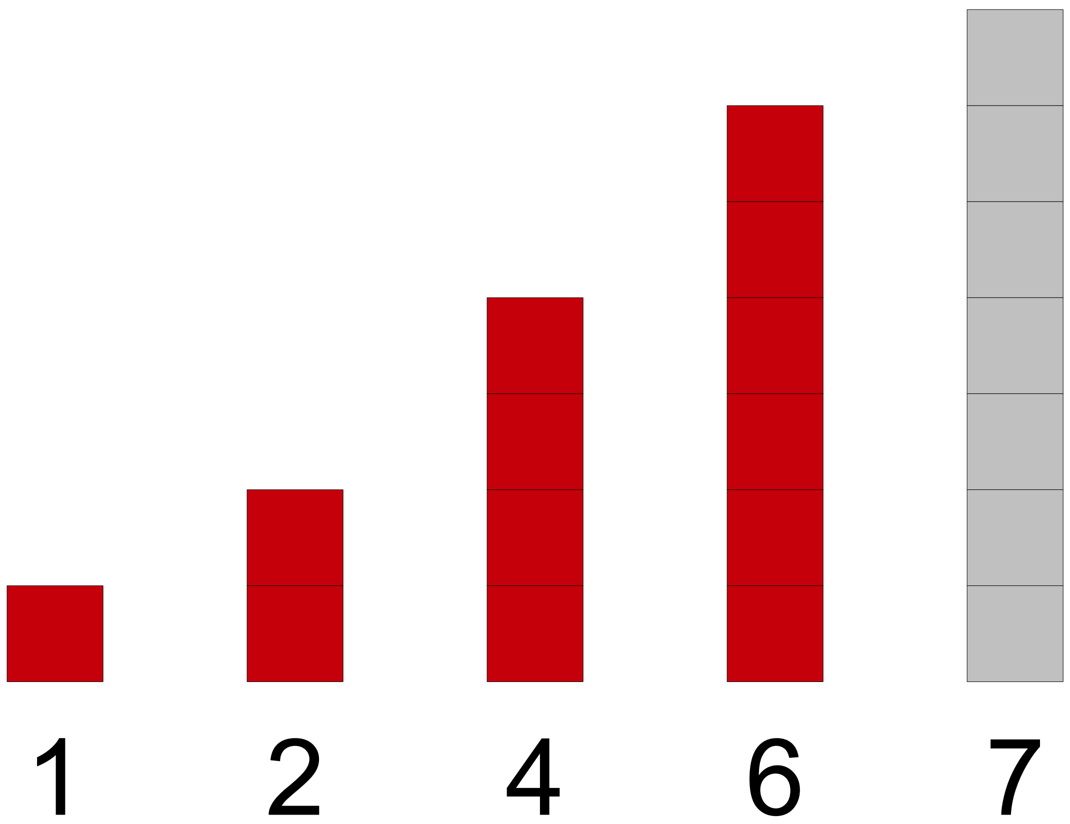
# An Initial Idea: ***Selection Sort***



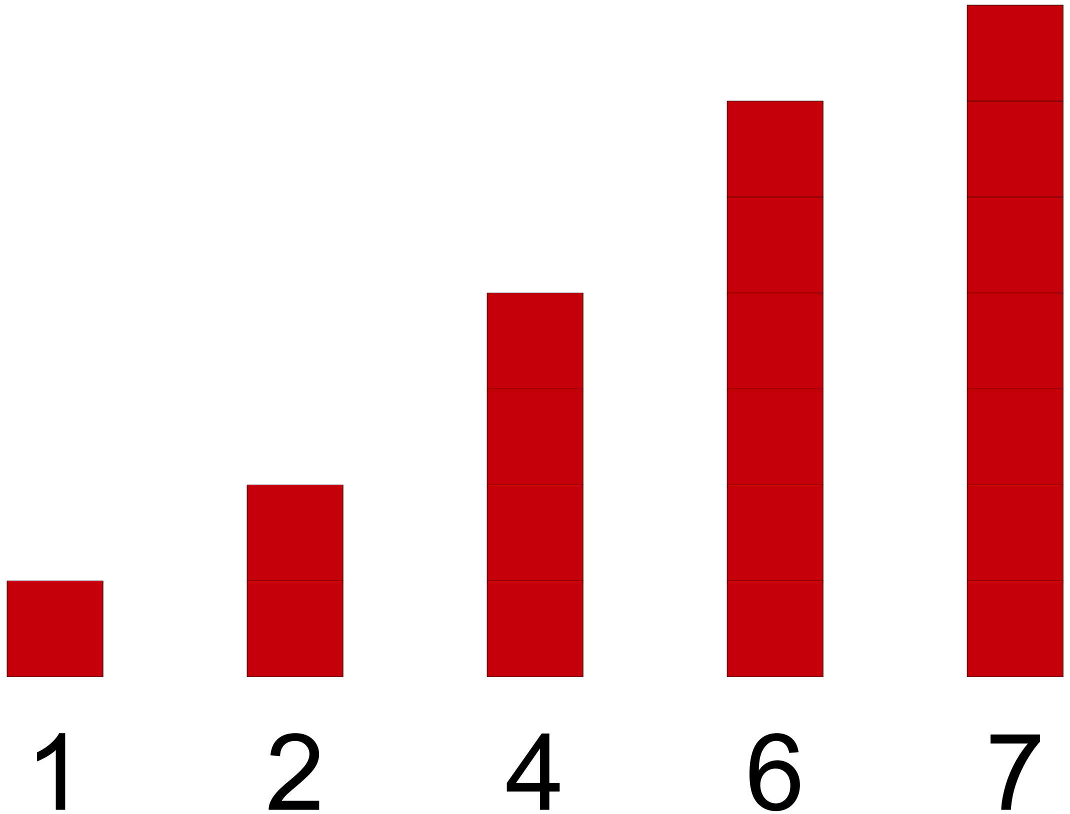
# An Initial Idea: ***Selection Sort***



# An Initial Idea: ***Selection Sort***



# An Initial Idea: ***Selection Sort***



# Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position.
- (etc.)

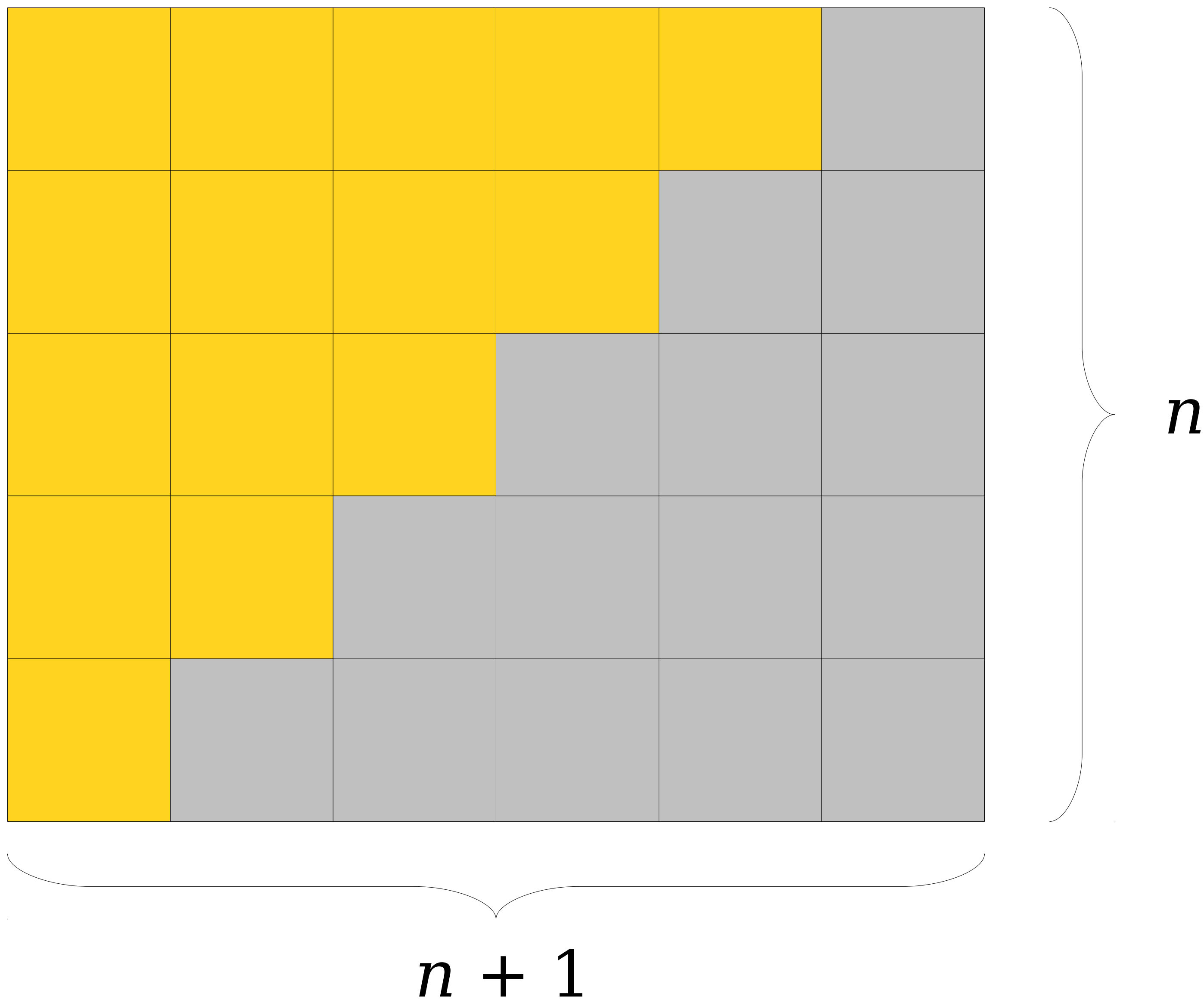
```cpp
/**
 * Sorts the specified vector using the selection sort algorithm.
 *
 * @param elems The elements to sort.
 */
void selectionSort(Vector<int>& elems) {
  for (int index = 0; index < elems.size(); index++) {
    int smallestIndex = indexOfSmallest(elems, index);
    swap(elems[index], elems[smallestIndex]);
  }
}

/**
 * Given a vector and a starting point, returns the index of the smallest
 * element in that vector at or after the starting point
 *
 * @param elems The elements in question.
 * @param startPoint The starting index in the vector.
 * @return The index of the smallest element at or after that point
 *         in the vector.
 */
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
  int smallestIndex = startPoint;
  for (int i = startPoint + 1; i < elems.size(); i++) {
    if (elems[i] < elems[smallestIndex])
      smallestIndex = i;
  }
  return smallestIndex;
}
```

# Analyzing Selection Sort

- How much work do we do for selection sort?
- To find the smallest value, we need to look at all $n$ array elements.
- To find the second-smallest value, we need to look at $n - 1$ array elements.
- To find the third-smallest value, we need to look at $n - 2$ array elements.
- Work is $n + (n - 1) + (n - 2) + \ldots + 1$.

$$n + (n\text{-}1) + \ldots + 2 + 1 = n(n{+}1) \,/\, 2$$



# The Complexity of Selection Sort

$$O(n\,(n+1)/2)$$

$$= O(n\,(n+1))$$

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

$$= O(n^2)$$

So selection sort runs in time **$O(n^2)$**.

# Thinking About O($n^2$)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 14 | 6 | 3 | 9 | 7 | 16 | 2 | 15 |

$\mathrm{T}(n)$

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 6 | 3 | 9 | 7 | 16 | 2 | 15 | 5 | 10 | 8 | 11 | 1 | 13 | 12 | 4 |

$\mathrm{T}(2n) \approx 4\mathrm{T}(n)$

# Selection Sort Times

| Size | Selection Sort |
|---:|---:|
| 10000 | 0.304 |
| 20000 | 1.218 |
| 30000 | 2.790 |
| 40000 | 4.646 |
| 50000 | 7.395 |
| 60000 | 10.584 |
| 70000 | 14.149 |
| 80000 | 18.674 |
| 90000 | 23.165 |

# Next Time

- ***Faster Sorting Algorithms***
  - Can you beat O($n^2$) time?
- ***Hybrid Sorting Algorithms***
  - When might selection sort be useful?