Overview of the Coming Week: May 4-8

Overview of this Week

- We've made it so far! Yay!
- Second last week of classes
- What days of the week are we in?   **May 4 - May 8**
- Things that are coming up:
  - HW 8 (Recursion) due on Monday May 4
  - HW 9 will be released Wed May 6, due May 11
  - Lab 10 (Oracle) is due May 7 **(Extra credit & Optional)**
  - No quiz this Friday (May 8)
  - Quiz 3 and 4 will be held May 15 and 22
- One stop shop for all course information: GLOW course homepage!

Lecture Topics

- So far in the course we have focused on solving various problems computationally
- The sequence of steps we follow to solve the problem (the recipe of our program) is called an **algorithm**
- How do we know if a particular algorithm is any good?
- Topic 1. **Efficiency.** How do we measure efficiency and performance of an algorithm?
    - "Big Oh" notation

Designing and analyzing algorithms:

- Topic 2. **Searching.** Exploring and analyzing efficient algorithms for searching in a sorted sequence
- Topic 3. **Sorting.** Exploring different sorting algorithms and comparing their performance

Any Questions, Come See Us!

Measuring Efficiency

Measuring Efficiency

- How do we measure how efficiency of our program?
  - We want programs that run "fast"
  - But what do we mean by that?
- **One idea:** use a stopwatch to see how long it takes.
  - Is this a good method?
  - What is the stop watch really measuring?
  - How long does this piece of code takes **on this machine** **on this particular input**
- Machine dependent
  - We want to evaluate our program not the machine's speed
- Cannot make any general conclusion
  - Doesn't tell us how fast the program will be different inputs

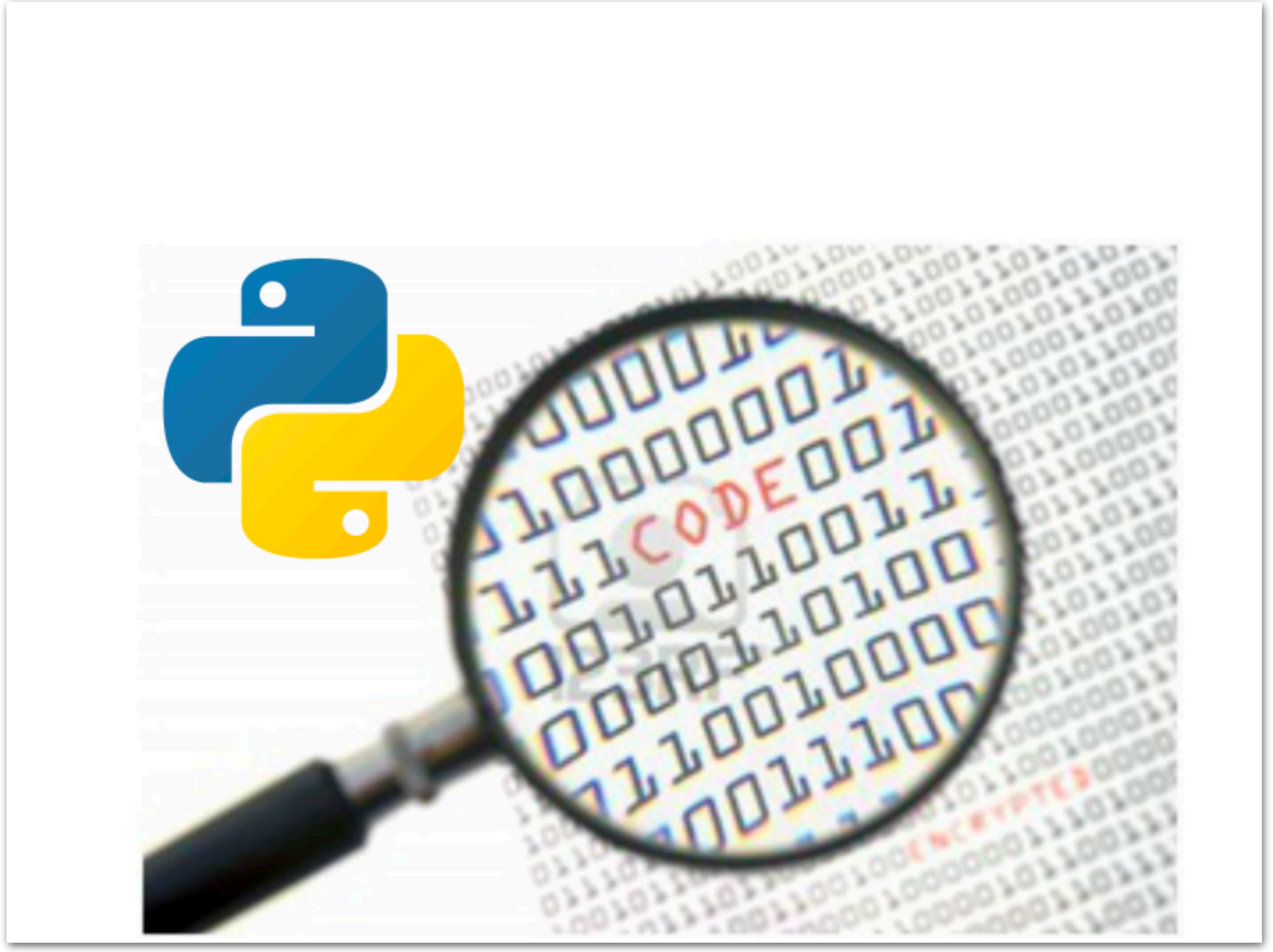
Efficiency Metric: Goals

We want a method to evaluate efficiency that:

- **Platform independent.** Is independent of the machine
- **Implementation independent.** Is independent of the implementation details
- **Guarantees that hold for different types of inputs.** Can help us make general conclusions about how the program will do on any arbitrary input
- **Dependence on input size.** Captures how the performance will "scale" when the input gets bigger
- *"Has the right level of specificity"*.
    - We don't want to be too specific (cumbersome)
    - Measure things that matter, ignore what doesn't

Platform Independent

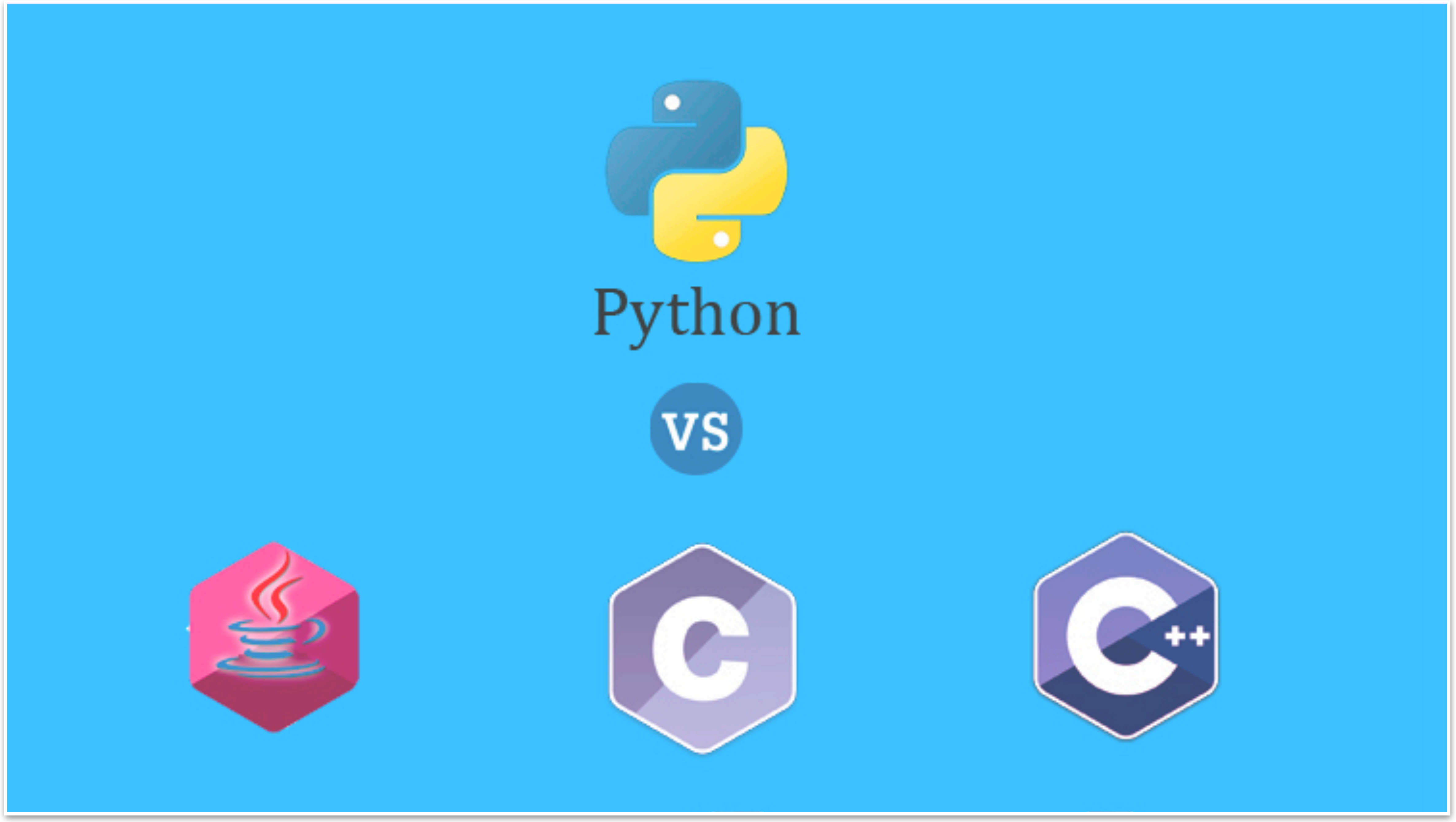
Evaluating the program, rather than the speed of the machine its run on.

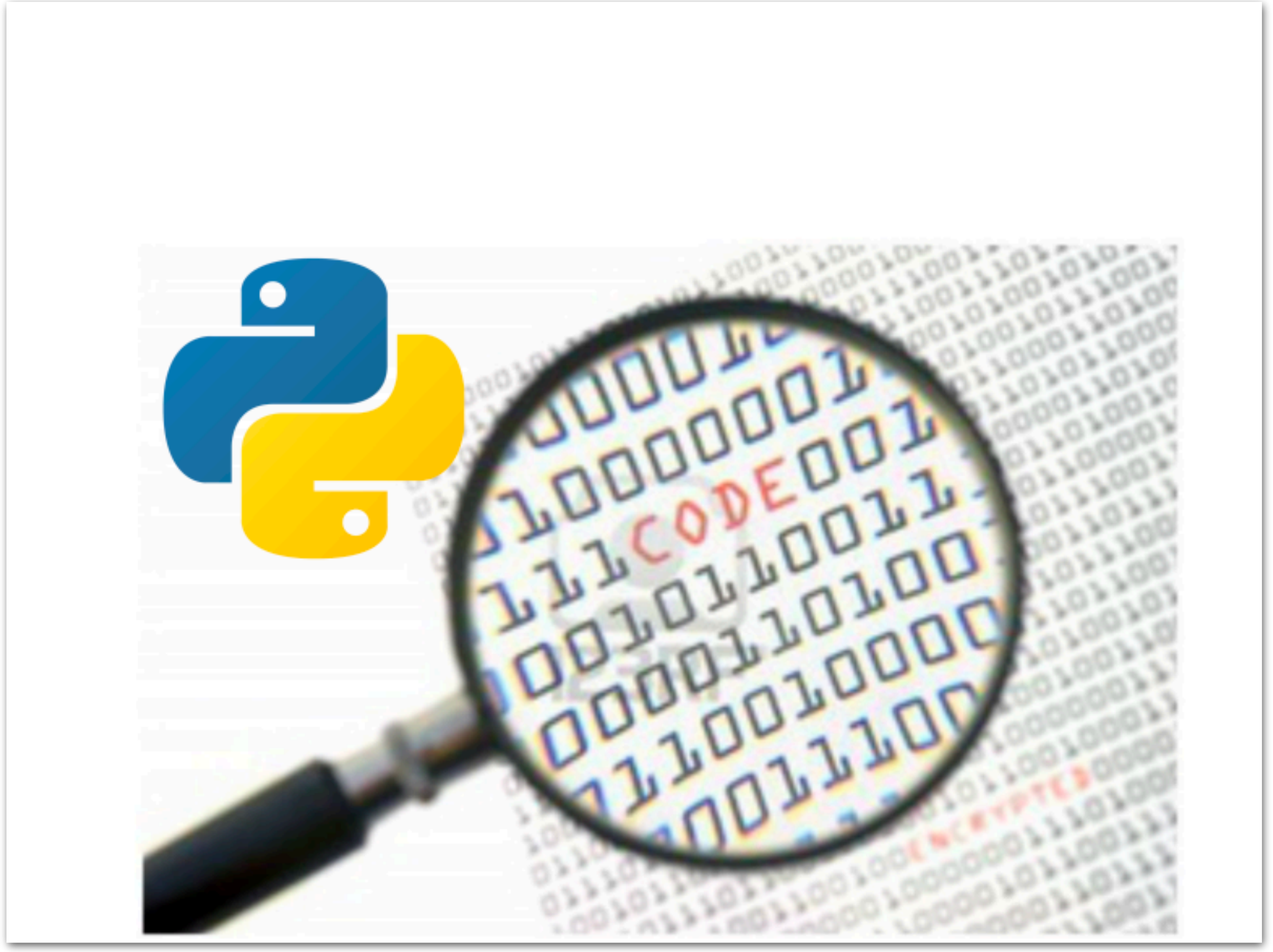


Implementation Independent

Actually, we want to evaluate the problem solving strategy (the algorithm) rather than the "program" itself.

- Count *number of steps* taken by the algorithm
- Sometimes referred to as "running time" (abusing term)





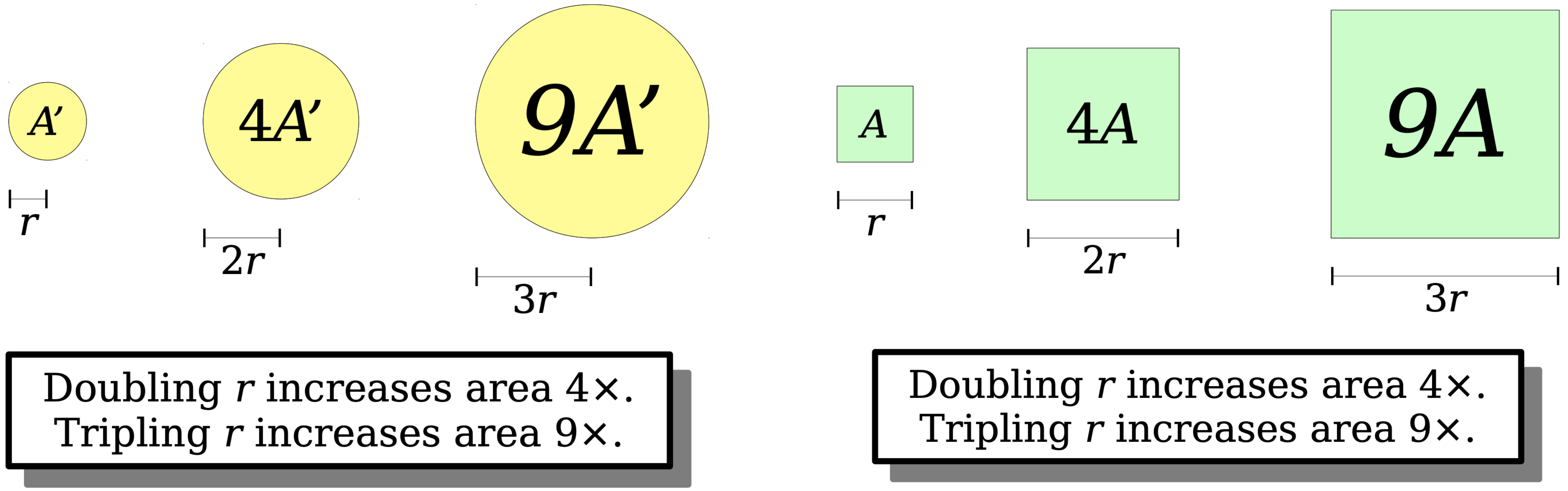
Worst-Case Guarantees

- We can't just analyze our algorithm on few inputs and declare victory
- **Best case.** Minimum number of steps taken over all possible inputs of a given size
- **Average case.** Average number of steps taken over all possible inputs of a given size
- **Worst case.** Maximum number of steps taken all possible inputs of a given size.

Takeaway. *We want provable guarantees, regardless of the input.*

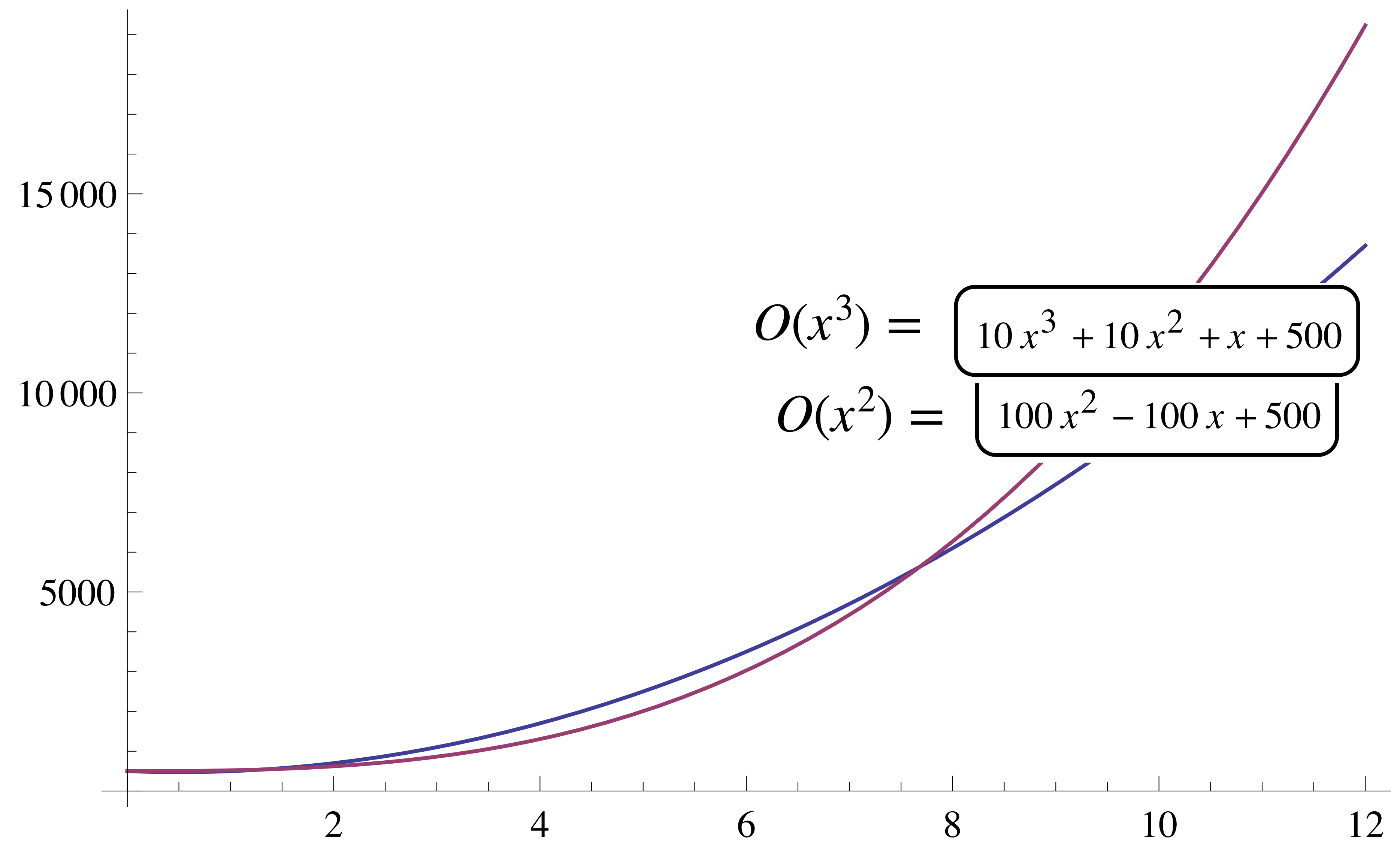
Dependence on Input Size

- Don't care about performance on "small inputs"
- Instead we care about "the rate at which it grows" with respect to the input size
- **Big-O notation** is a way of quantifying (in fact, upper bounding) how the function grows wrt input size
- For example:
    - A square of side length r has area $O(r^2)$.
    - A circle of radius r has area $O(r^2)$.



Big Oh: Level of Specificity

- Big Oh notation is designed to capture the rate at which which the number of steps taken by the algorithm grows wrth size of input n, *"as n gets large"*
- Not precise by design, it ignores information about
  - Multiplication constants, e.g. $100n = O(n)$
  - Lower-order terms: terms that contribute to the growth but are not dominant, so they get glossed over $O(n^2 + n + 10) = O(n^2)$
- Powerful tool for predicting performance behavior: focuses on what matters, ignores the rest
- Separates fundamental improvements from smaller optimizations



https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1204/lectures/11/Slides11.pdf

Example Analysis

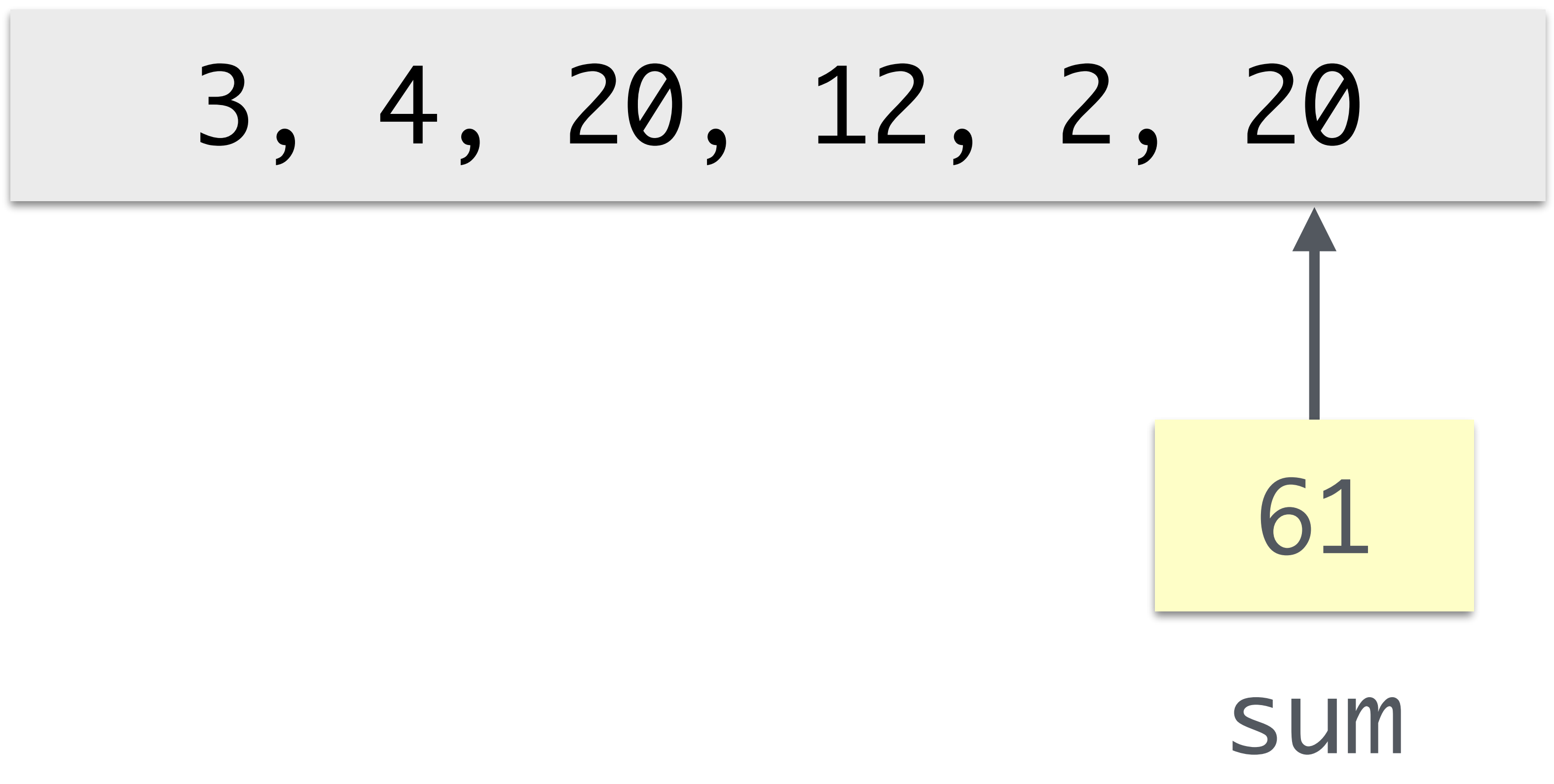
- Summing up items in a list of numbers
- Assume basic operations such as variable assignments, addition, multiplication represent a "single step"

- ```
  # assume numList is a list of integers
  sum = 0    # line 1
  for item in numList:
        sum = sum + item
  ```

  `sum = 0` → 1 step

  `sum = sum + item` → Single step executed $n =$ `len(numList)` times

- Total time steps $1 + n = O(n)$


  ```

Searching in an Unsorted List

- Okay to overestimate: we are computing an upper bound
- Worst case analysis: doesn't matter if faster on some inputs

```
def linearSearch(e, L):
    for elem in L:
        if elem == e:
            return True
    return False
```

Might not always run, but assume it does: **overestimate**

Might return early if e is first item in list
but interested in the **worst case**;
happens if e is not in the list or last item

- Statements in loop body execute $n =$ `len(L)` times and other operations take constant number of steps
- Thus, overall takes $O(n)$ time

Searching in an Unsorted List

- Okay to overestimate: we are computing an upper bound
- Worst case analysis: doesn't matter if faster on some inputs

```
def linearSearch(e, L):
    for elem in L:
        if elem == e:
            return True
    return False
```

2 steps, executed $n =$ `len(L)` times

1 step

- Overall we have $2 \cdot n + c = O(n)$ steps

Linear Algorithms: $O(n)$

- Algorithms that take steps proportional to the size of the input, are called linear time algorithms or $O(n)$ time
- Examples: $n =$ length of input sequence
  - Summing up a list of numbers
  - Searching in an unsorted list
- Any algorithm where we iterate over a sequence of length n and do constant number of operations within the loop
- Any algorithm that "touches" all input items
- "Simple loops" are usually $O(n)$
- But what about "*nested loops*"?

Quadratic Steps: $O(n^2)$

- Usually occurs when we have a loop within a loop ("nested")
- Example: determining if a list L_1 is a subset of another list L_2 (that is, every item in L_1 is in L_2) e.g. $L_1 = [2,4,6]$ is a subset of $L_2 = [1,2,3,4,5,6]$

```
def subsetOf(L1, L2):
    matched = False
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                matched = True
        if not matched:
            return False
    return True
```

Found e1 in list L2

If e1 not in L2, can return False

If we reach this line, we have found a match for all elements

Quadratic Steps $O(n^2)$

- Usually occurs when we have a loop within a loop ("nested")
- Example: determining if a list L_1 is a subset of another list L_2 (that is, every item in L_1 is in L_2) e.g. $L_1 = [2,4,6]$ is a subset of $L_2 = [1,2,3,4,5,6]$

```
def subsetOf(L1, L2):
    matched = False
    for e1 in L1:
        matched = linearSearch(e1, L2)
        if not matched:
            return False
    return True
```

Executes len($L1$) $\leq n$ steps, where $n =$ len(L2)

$O(n)$ steps

$O(n^2)$ steps

Quadratic Steps $O(n^2)$

- Usually when you have a nested loop, e.g.

```
for i in range(n):
    for j in range(n):
        print('something')
```

- When you are iterating over two sequences and comparing items, e.g.,
    - checking if a string is a substring of another string
    - finding common elements between two sequences

Two loops vs Nested Loops

- Sequential loops vs nested loops are like addition vs multiplication when it comes to big Oh

- For example

```
for i in range(n):
    print('a')
for j in range(n):
    print('a')
```

$O(n)$ steps

$O(n)$ steps

$O(n) + O(n) = O(n)$ steps

- Versus

```
for i in range(n):
    for j in range(n):
        print('a')
```

$O(n)$ steps

$O(n) * O(n) = O(n^2)$ steps

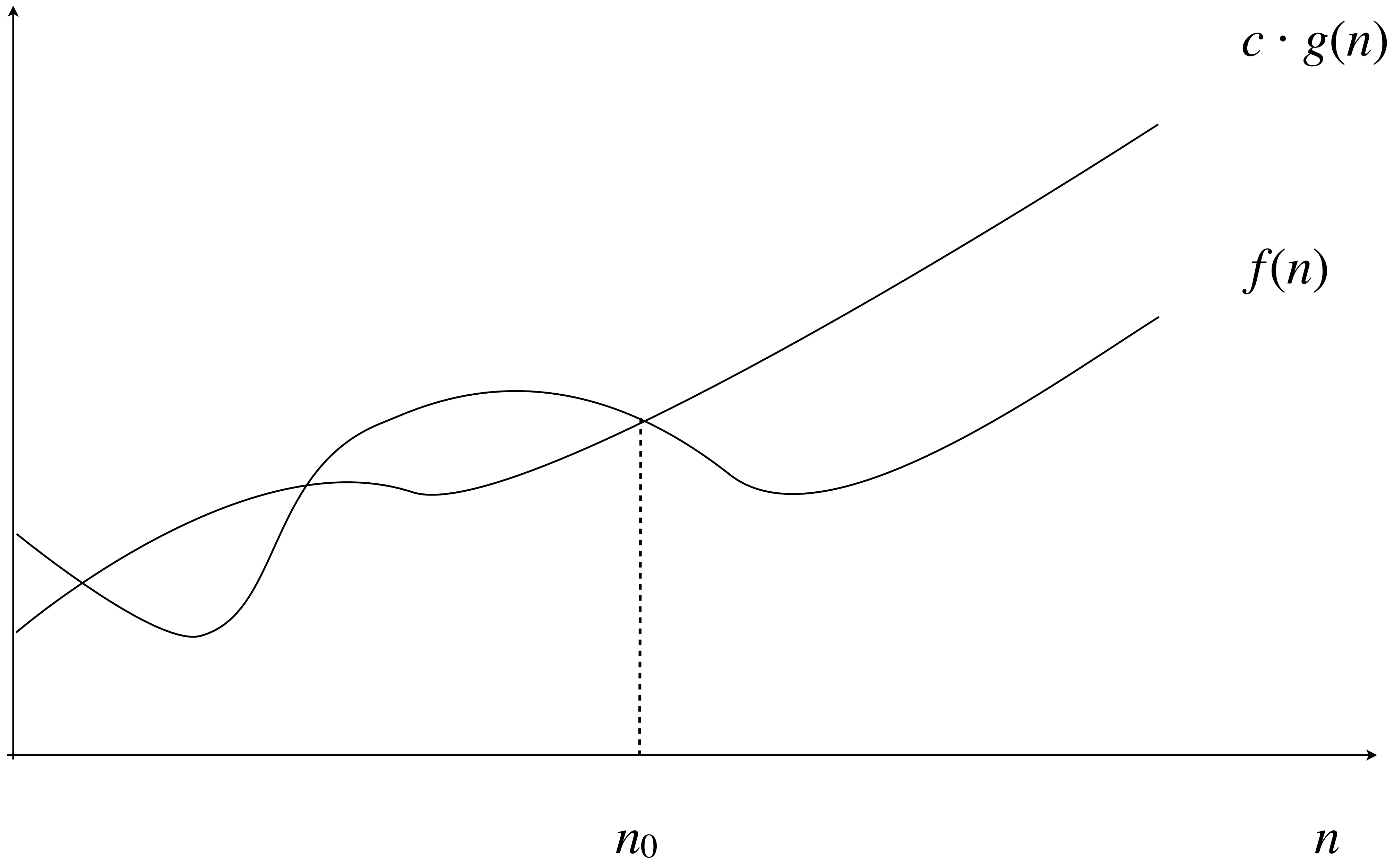
Why We Aren't Stating the Definition

Definition: $f(n)$ is $O(g(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

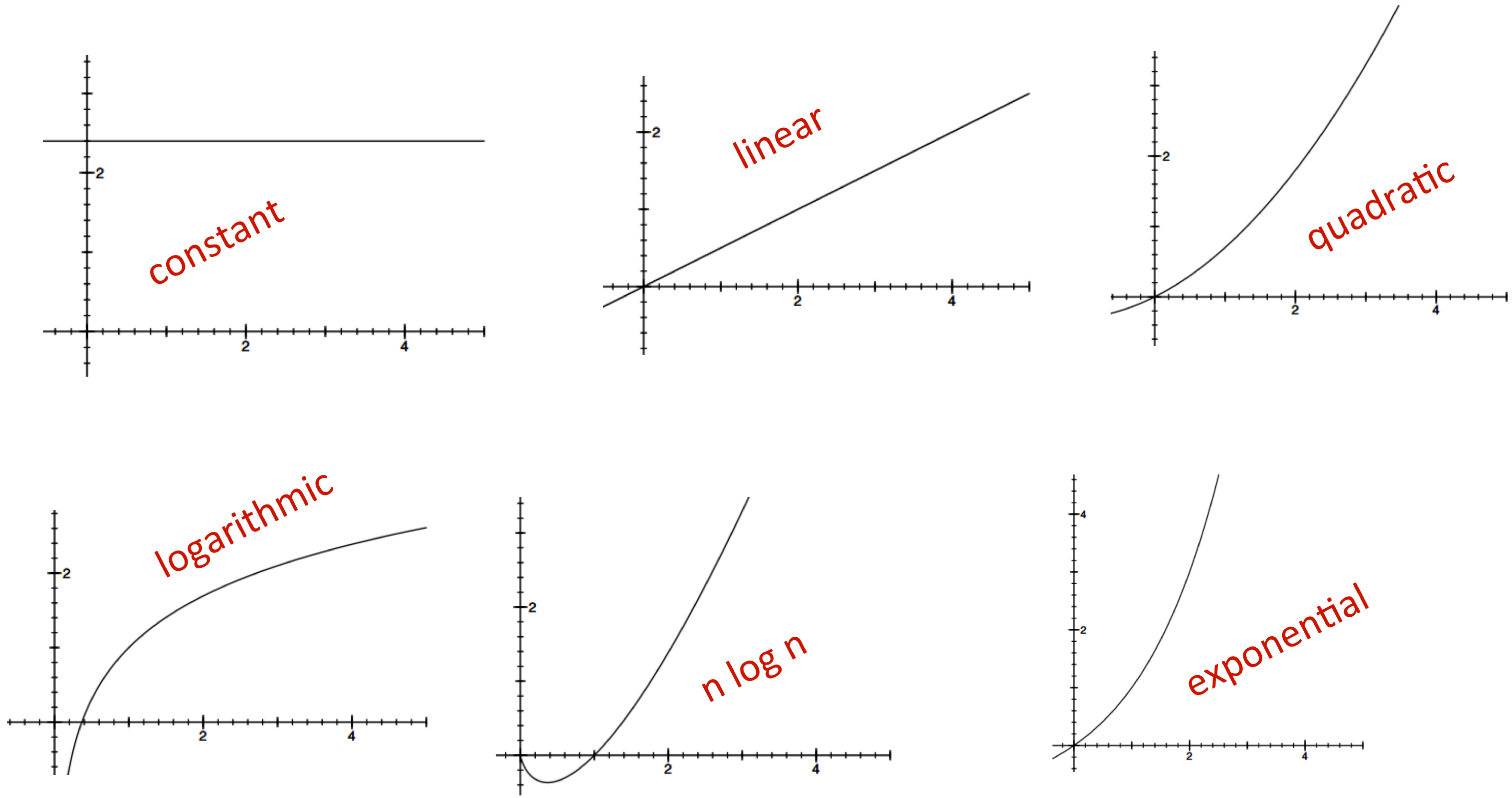
In other words, for sufficiently large n, $f(n)$ is asymptotically bounded above by $g(n)$

Examples

- $100n^2 = O(n^2)$
- $n \log n = O(n^2)$
- $5n^3 + 2n + 1 = O(n^3)$



Common Big Oh Functions



What's Next

- What on earth is $\log n$
- Searching in a sorted list:
    - Can we search faster than $O(n)$ if the list is sorted?
    - Binary search: algorithm that takes $O(\log n)$ steps
- Sorting algorithms:
    - We have used Python's in-built sorting methods
    - How do we design our own sorting algorithm?
    - How long does sorting a list of n takes?
    - Example of an $O(n \log n)$ algorithm
- Example of an exponential time algorithm

Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/