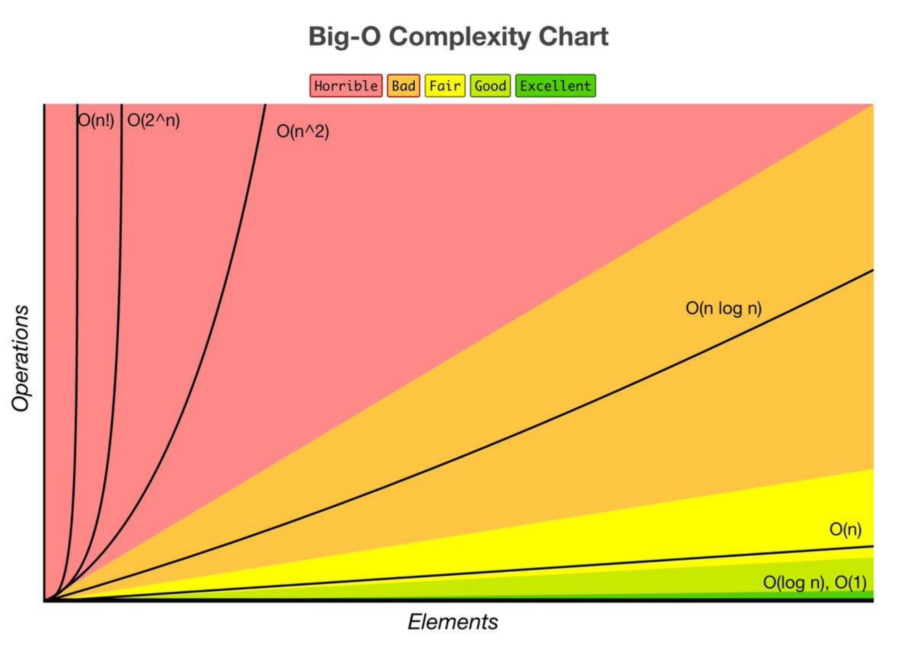
Algorithmic Analysis (Big-O) Sorting

Frank Burkholder
Taryn Heilman
Joe Gartner

galvanize



Learning Objectives

galvanize

- **Understand** Big-O notation and how to use it to describe algorithms
- Be ready to **write implementations** of
    - Insertion Sort
    - Bubble Sort
    - Heap Sort
- **Optional: Python Review (or maybe New!)** Define a decorator

Scenario

Algorithmic Time Complexity

galvanize

“Big-O” Notation

- Big O Notation - Used to describe how the runtime (time complexity) and size (space complexity) of an algorithm increases as the size of the input array of **length *n*** increases. We’ll be focusing on time complexity.

- This is an order of magnitude approximation, meaning we only worry about the leading term. In big-O, a process that requires *n* computations is the same as one that requires *3*n* compuations, both are *O(n)*.

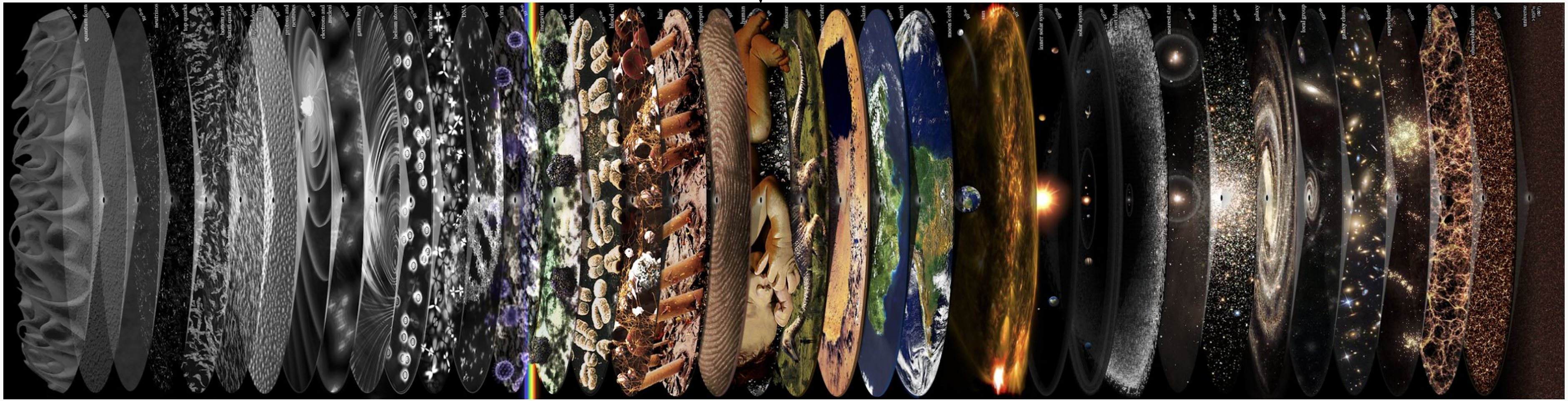
- Expressed as a function of *n* (e.g, C^n, n^3, n^2, $n \log n$, n, $\log n$)

Order of magnitude approximation examples

Number N	Expression in $N = a \times 10^b$	Order of magnitude b
0.2	2×10^{-1}	-1
1	1×10^{0}	0
5	0.5×10^{1}	1
32	0.32×10^{2}	2
999	0.999×10^{3}	3

Person
(L~10^0 m)

Source: Wikipedia

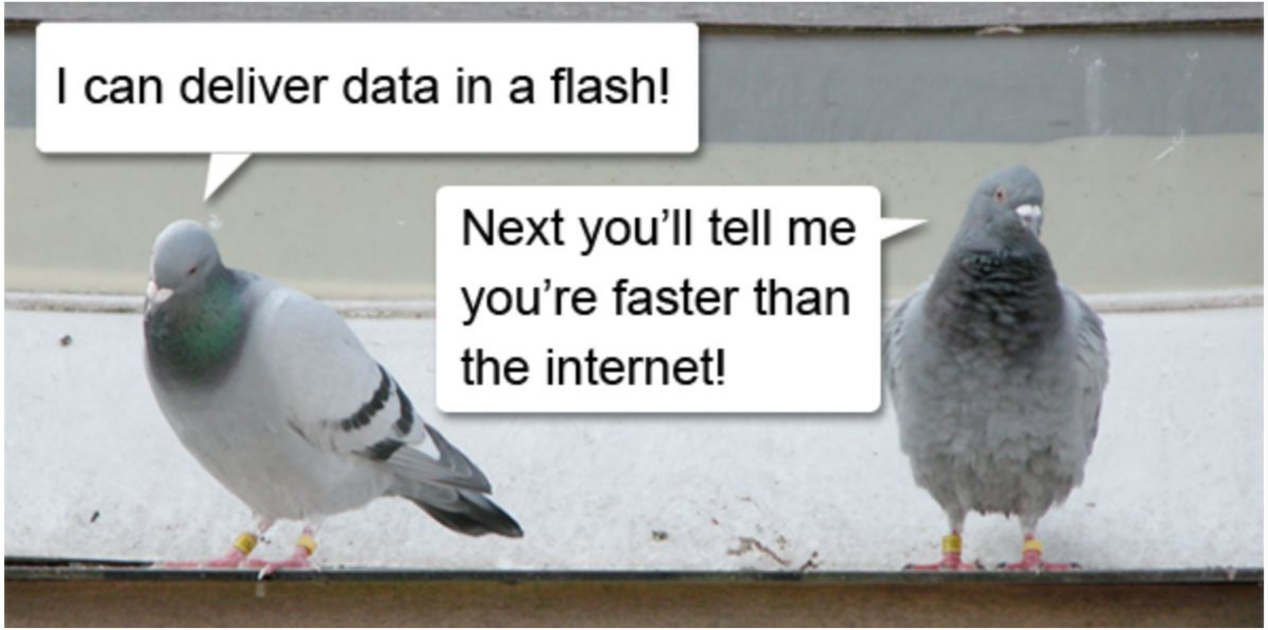


“Big-O” Notation - why?

This topic often shows up in software interviews, as it is a good test of several things:

- Estimating the growth in time for an algorithm as the input size grows
- Evaluate your knowledge of data structures
- They test your ability to think about what you are doing

Pigeon vs Broadband

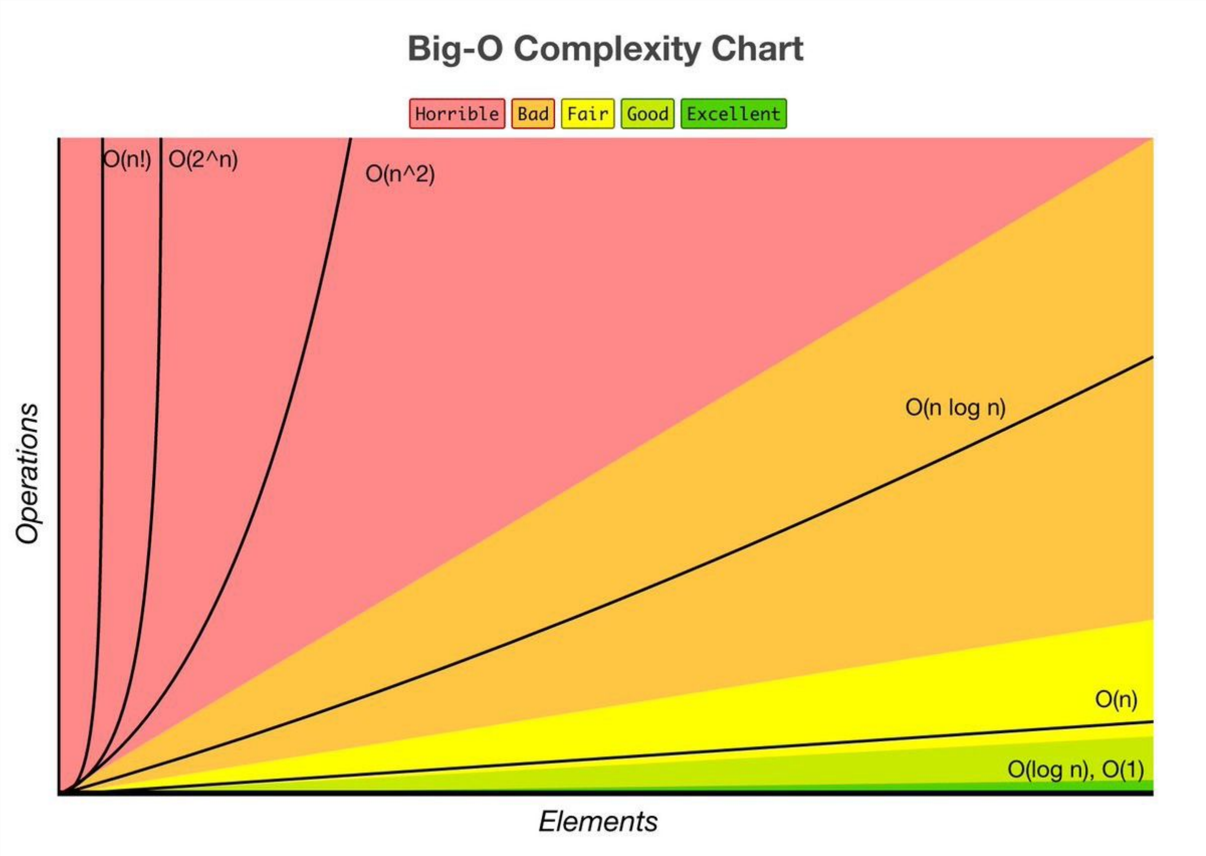


[Source](#)
[another example](#)

O(N) of common routines (more complete on Wikipedia)

Time Complexity (best to worst)	Name	Description - Given an input of size *n*….	Example operation or algorithm
$O(1)$	Constant	Only a single step required to complete the task.	Lookup in a set. Appending to a list.
$O(\log n)$	Logarithmic	The number of steps it takes to accomplish the task are decreased by some factor with each step.	Binary search
$O(n)$	Linear	The number of of steps required is directly related to *n*.	Lookup in a list. Kmeans algorithm.
$O(n \log n)$	Log-linear	The number of of steps required is directly related to *n* multiplied by some factor that is a factor of *n* (but much less than *n*).	Merge sort
$O(n^2)$	Quadratic	The number of steps it takes to accomplish a task is square of *n* (**bad**).	Double `for` loop. Create covariance matrix. Hierarchical clustering.
$O(C^n)$	Exponential	The number of steps it takes to accomplish a task is a constant to the *n* power (**very bad**).	Graph colouring.

O(n) rough comparison of number of operations



What is the O(n) complexity of this algorithm?

```python
def check_list_elements(x, lst):
    for i in lst:
        if element==x:
            return x
```

Assume there are *n* items in `lst`.

How many comparisons does this function make?

What is the run time complexity?

What is a way to reduce the complexity?

What is the O(n) complexity of this algorithm?

```python
def check_list_elements(x, lst):
    for i in lst:
        if element==x:
            return x
```

Assume there are *n* items in `lst`.

How many comparisons does this function make? **Each item `lst` with x.**

What is the run time complexity? **`lst` is `n` elements, so `n`, or O(`n`)**

What is a way to reduce the complexity? **`Maybe use a set instead of a list?`**

What is the O(n) complexity of this algorithm?

galvanize

```
def print_pairs(lst):
    for x in lst:
        for y in lst:
            print(x,y)
```

Assume there are *n* items in `lst`.
How many pairs does this function make?
What is the run time complexity?

What is the O(n) complexity of this algorithm?

```
def print_pairs(lst):
    for x in lst:
        for y in lst:
            print(x,y)
```

Assume there are *n* items in `lst`.

How many pairs does this function make?**Each `item` in `lst` with every other `item` in `lst`.**

What is the run time complexity? **`lst` is `n` elements, so `n` * `n`, or O(n^2)**

What is the O(n) complexity of this algorithm?

```
def print_pairs_two_lists(lst1, lst2):
    for a in lst1:
        for b in lst2:
            print(a, b)
```

Assume there are *n* items in `lst1 and m items in lst2`

How many pairs does this function make?

What is the run time complexity?

What is the O(n) complexity of this algorithm?

galvanize

```
def print_pairs_two_lists(lst1, lst2):
    for a in lst1:
        for b in lst2:
            print(a, b)
```

Assume there are *n* items in `lst1 and m items in lst2`

How many pairs does this function make?**Each `item` in `lst1` with each item in `lst2`.**

What is the run time complexity? **so `n` * `m`, or O(`nm`)**

What is the O(n) complexity of this algorithm?

```
def max_min(lst):
    current_max = 0
    current_min= 0
    for i in lst:
        current_max = max(current_max, i)
    for i in lst:
        current_min = min(current_min, i)
    print(current_max, current_min)

def max_min2(lst):
    current_max = 0
    current_min= 0
    for i in lst:
        current_max = max(current_max, i)
        current_min = min(current_min, i)
    print(current_max, current_min)
```

Assume there are *n* items in `lst`.

What is the run time complexity for each function?

What is the O(n) complexity of this algorithm?

galvanize

```python
def max_min(lst):
    current_max = 0
    current_min= 0
    for i in lst:
        current_max = max(current_max, i)
    for i in lst:
        current_min = min(current_min, i)
    print(current_max, current_min)

def max_min2(lst):
    current_max = 0
    current_min= 0
    for i in lst:
        current_max = max(current_max, i)
        current_min = min(current_min, i)
    print(current_max, current_min)
```

Assume there are *n* items in `lst`.

What is the run time complexity for each function? **O(n) for both**

What is the O(n) complexity of this algorithm?

```
def find_anagrams(lst):
    result = []
    for word1 in lst:
        for word2 in lst:
            if word1 != word2 and sorted(word1) == sorted(word2):
                if word1 not in result:
                    result.append(word1)
                if word2 not in result:
                    result.append(word2)
    return result
```

Assume there are *n* words in `lst`.

How many comparisons does this function make? **Each `word` in `lst` with each `word` in `lst`.**

What is the run time complexity? **`lst` is `n` elements, so `n` * `n`, or O(n^2)**

What about this way?

```
def find_anagrams(lst):
    result = []
    d = defaultdict(list)
    for word in lst:
        d[tuple(sorted(word))].append(word)
    for key, value in d.iteritems():
        if len(value) > 1:
            result.extend(value)
    return result
```

How many comparisons does this function make?

What is the run time complexity? *O(n)*

What about this way?

galvanize

```
def find_anagrams(lst):
    result = []
    d = defaultdict(list)
    for word in lst:
        d[tuple(sorted(word))].append(word)
    for key, value in d.iteritems():
        if len(value) > 1:
            result.extend(value)
    return result
```

How many comparisons does this function make? **Doesn't make any. Checks for more than one word for a sorted character key in `d`, which is much smaller than length *n*. The `lst` is iterated through only once.**

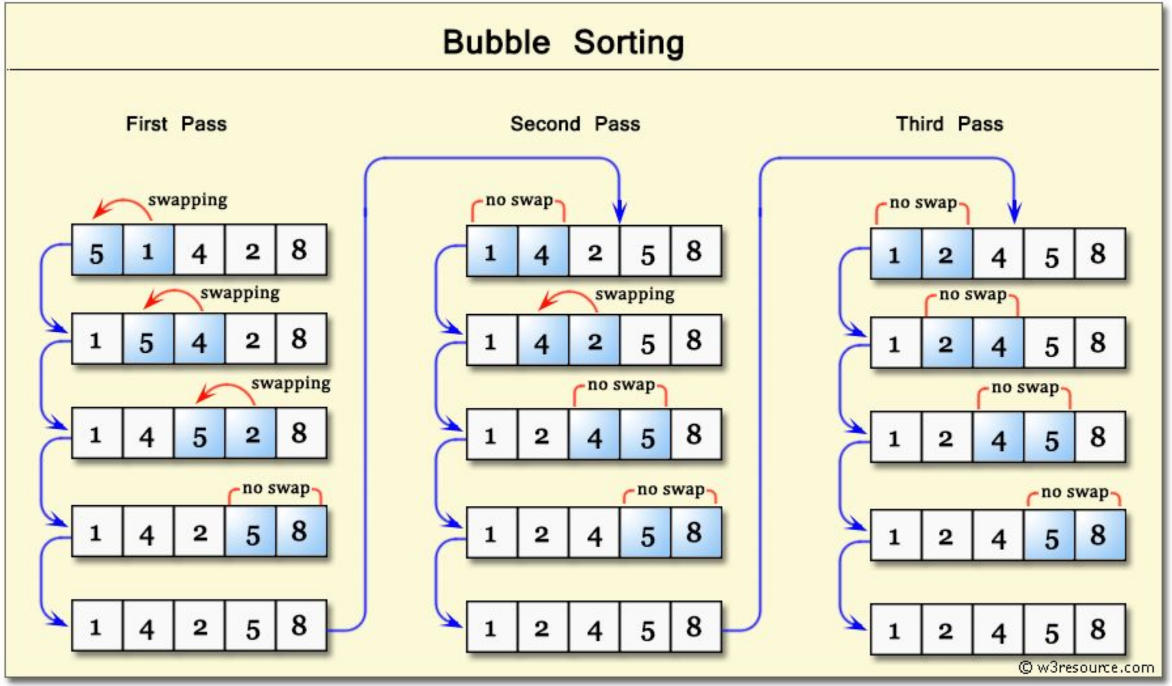
What is the run time complexity? ***O(n)***

Investigate

`find_anagrams.py`

Sorting Algorithms

Bubble Sort



[Wikipedia visualization]()

https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-4.php

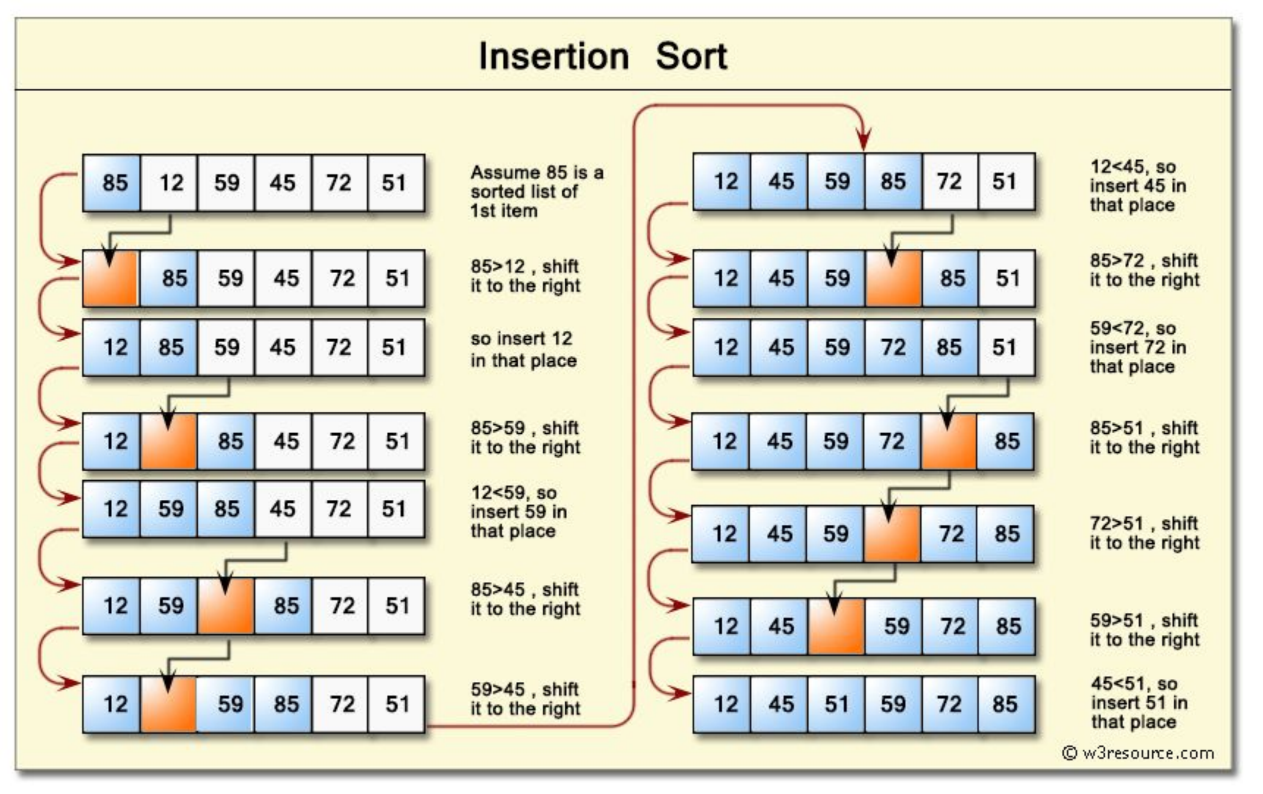
Bubble Sort

```
# psuedocode
def bubbleSort(lst):
    for i in range(len(lst) - 1, 1, -1): # loop backwards
        for j in range(i):
            if alist[j] > alist[j+1]:
                swap alist[j] and alist[j+1]
```

What is the run time complexity?

Insertion Sort

galvanize



[Wikipedia visualization.]

https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-6.php

Insertion Sort

```
# psuedocode
def insertionSort(lst):
    for i in range(1, len(lst)):

        val = lst[i]

        while i > 0 and lst[i - 1] > val:
            move lst[i - 1] to the right
            decrement i

        assign val to lst[i]
```

What is the run time complexity?

Break-out

Pair up and sort a shuffled suit from a deck of cards.

Part I:
Try Bubble Sort and Insertion Sort something at home (5-6 playing cards or maybe some coins). Can you explain and demonstrate each sorting algorithm to your partner?

Part II:
There are two text files in the lecture repo:
`unsorted_num_1.txt` and `unsorted_num_2.txt`

Write a Python script that you run from the command line, using argument parsing, to sort the numbers in the text file. Use Bubble Sort.

You should be able to run your script like this:

```
$ python bubblesort.py --in unsorted_num_1.txt --out sorted_num_1.txt
```

There are better sorting algorithms!

- Heap Sort
- Merge Sort
- Quick Sort
- All of the above have complexity O(n log n)

- Read more at https://wikipedia.org/wiki/Sorting_algorithm

Learning Objectives

High level takeaways:

- Why data structures matter (do you need to use a list? maybe tuple, dict, set is better?)
- When referring to time it takes to run code… our favorite answer, **it depends!!!!** (size of data, computational power, memory, etc.) so we use Big O notation instead
- One common place to talk about Big O is in sorting algorithms
- Keep in mind **‘premature optimization is the root of all evil’**

Review: Decorators

```python
class SortTester(ABC):
    """
    Abstract base class for our sorting classes

    Allows for the calling of a sort, a stack count sort, and a timed sort
    """
    def __init__(self):
        pass

    @classmethod
    def this_sort(cls, input_list):
        pass

    @timeit
    def timed_sort(self, input_list):
        pass
```

Decorators

See `decorators.ipynb`

Learning Objectives

galvanize

- **Understand** Big-O notation and how to use it to describe algorithms
- Be ready to **write implementations** of
    - Insertion Sort
    - Bubble Sort
    - Heap Sort
- **Python Review (New!)** Define a decorator