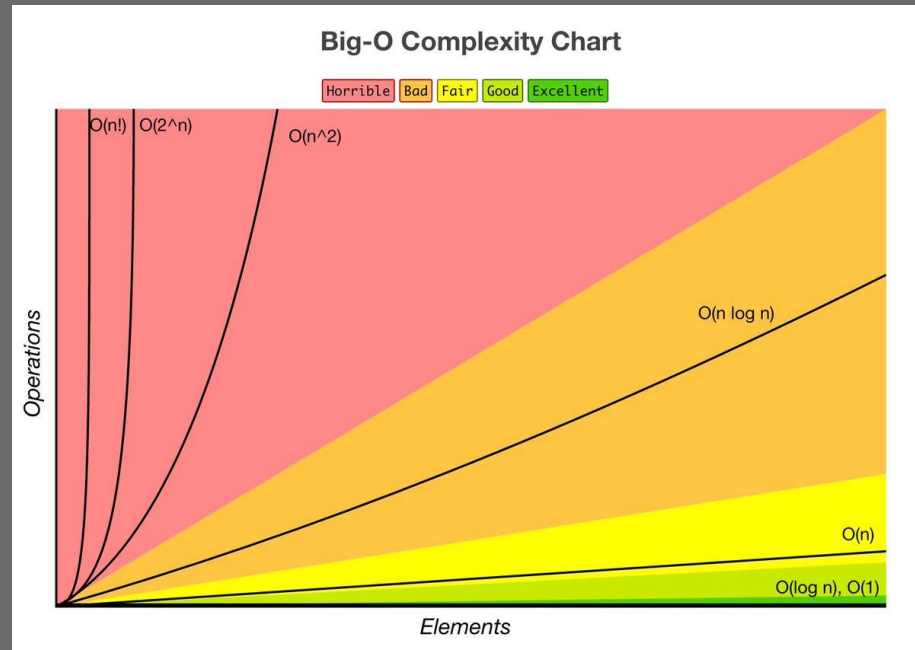


Algorithmic Analysis (Big-O) Sorting

Frank Burkholder
Taryn Heilman
Joe Gartner



- **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



Algorithmic Time Complexity



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

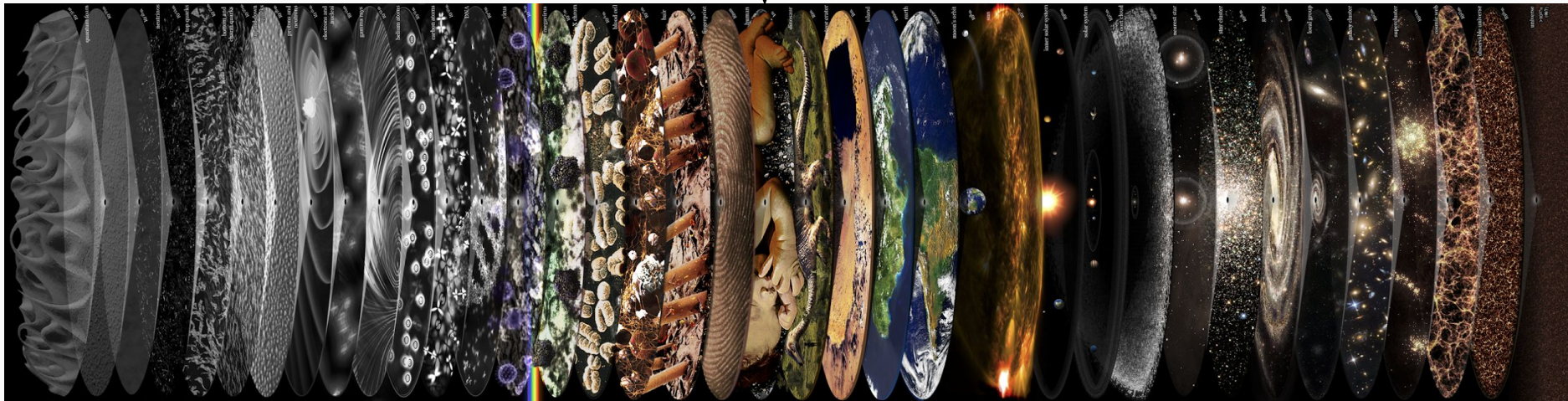
Order of magnitude approximation examples

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
($L \sim 10^0$ 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

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?

What is the run time complexity?

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?

```
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.**

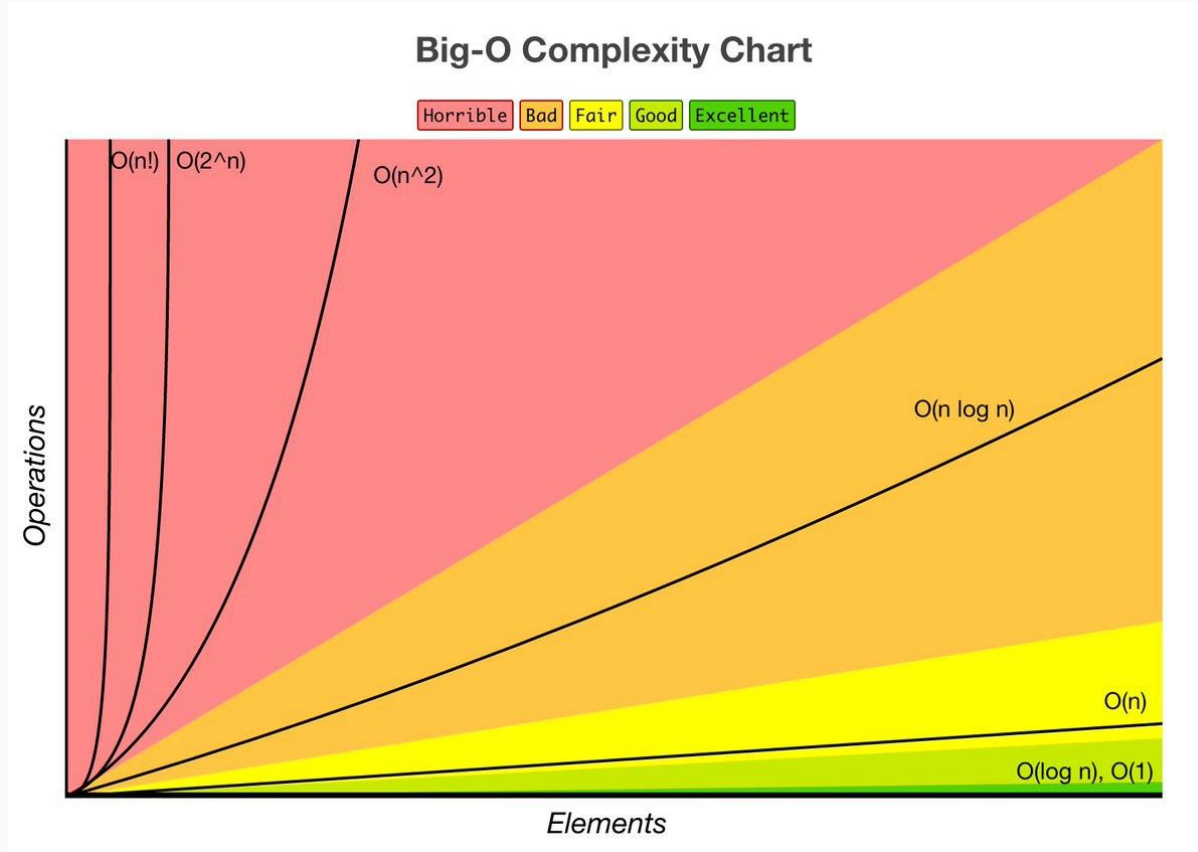
What is the run time complexity? **$O(n)$**

```
find_anagrams.py
```

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 <code>for</code> 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 .

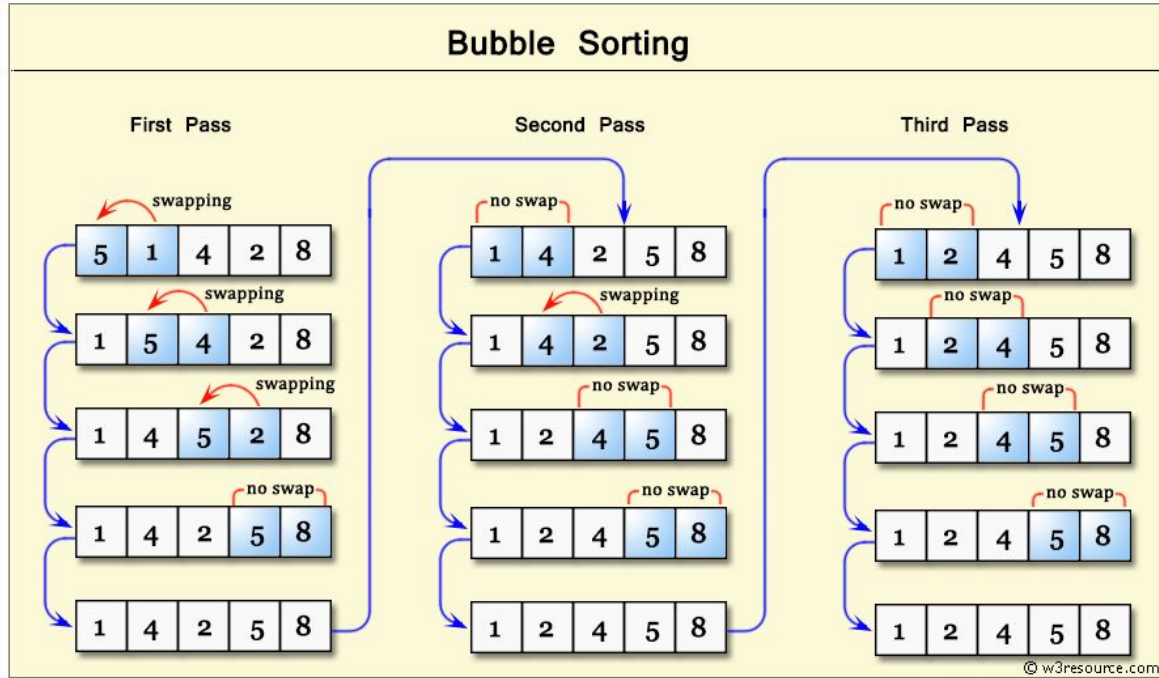
$O(n)$ rough comparison of number of operations



Sorting Algorithms



Bubble Sort

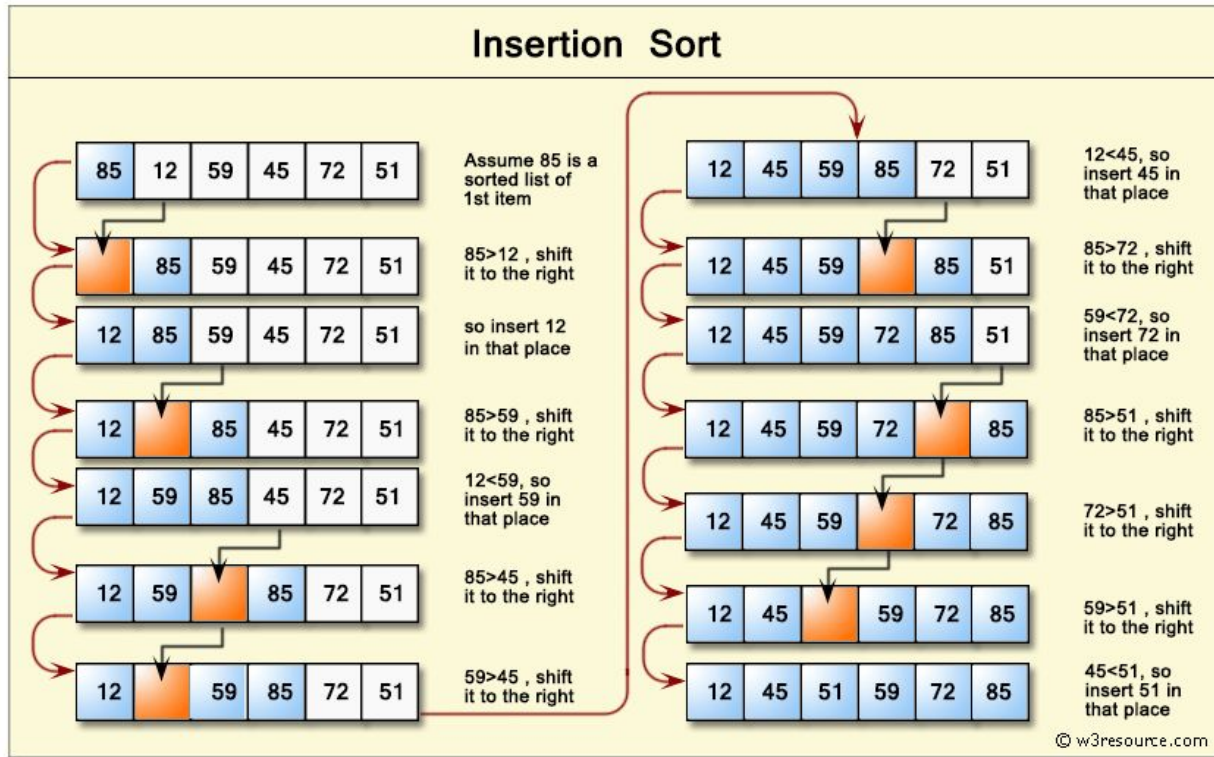


[Wikipedia
visualization](https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-4.php)


```
# 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



[Wikipedia visualization.](https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-6.php)

```
# 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?

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

Part I:

Try Bubble Sort and Insertion Sort with the deck of cards. 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

```
16 class SortTester(ABC):
17     """
18     Abstract base class for our sorting classes
19
20     Allows for the calling of a sort, a stack count sort, and a timed sort
21     """
22     def __init__(self):
23         pass
24
25     @classmethod
26     def this_sort(cls, input_list):
27         pass
28
29     @timeit
30     def timed_sort(self, input_list):
31         pass
32
33
```



See `decorators.ipynb`

- **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