

Sorting & Running Time

BUAD 5042

Topics

- Big-Oh running time
- Broad perspective:
 - Getting code to run faster
 - Running time and micro-optimization
 - Optimizing programs
- Sorting methods and running time
 - **Bubble sort**
 - **Selection sort**
 - **Insertion sort**
 - **Merge sort**
 - Heap sort
 - Quick sort

Big-Oh Running Time

- We are concerned with computing how long programs will take to run
 - ... particularly for large problems
 - Interesting and important problems are large

Big-Oh Examples

- Compute the sum of an n element array

```
9 n = 3
10 array = list(range(n))
11 array_sum = 0
12 for element in array:
13     array_sum += element
14 print('array_sum = ' + str(array_sum))
```

Big-Oh Examples

- How many computations are required to find the closest DC for each store?

```
8 """
9 The j-th element in the i-th sub list of dist represents the distance
10 from store i to distribution center j
11 """
12 dist = [[1,5,11,9],[3,7,14,21],[14,9,7,6],[23,15,9,17]]
13 answer = []
14
15 """ Find the closest DC for each store """
16 for store in dist:
17     dist_min = 999999999999999999
18     for j in range(len(store)):
19         if store[j] < dist_min:
20             dist_min = store[j]
21             dc_num = j
22     answer.append((dist_min,dc_num))
23 print('The minimum distances are as follow: ' + str(answer))
```

Big-Oh Examples

- How many computations are required for this matrix multiplication program?

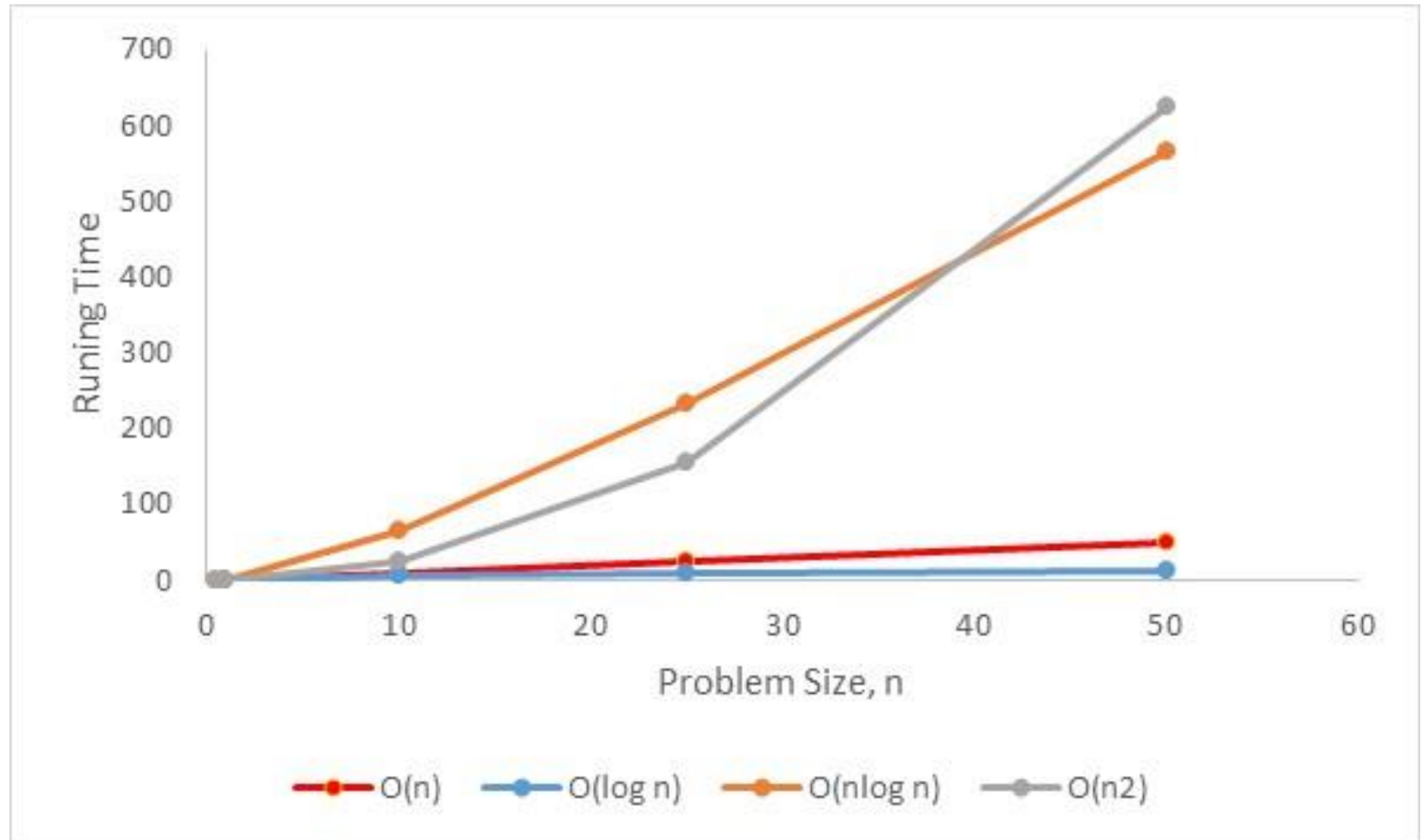
```
8 A = [[2,5,9],[1,5,3],[7,1,4]]
9 I = [[1,0,0],[0,1,0],[0,0,1]]
10 AI = [[0,0,0],[0,0,0],[0,0,0]]
11
12 for i in range(len(A)):
13     for k in range(len(AI)):
14         for j in range(len(A[i])):
15             AI[i][k] += A[i][j] * I[j][k]
16 print('AI = ' + str(AI))
```

Big-Oh Examples

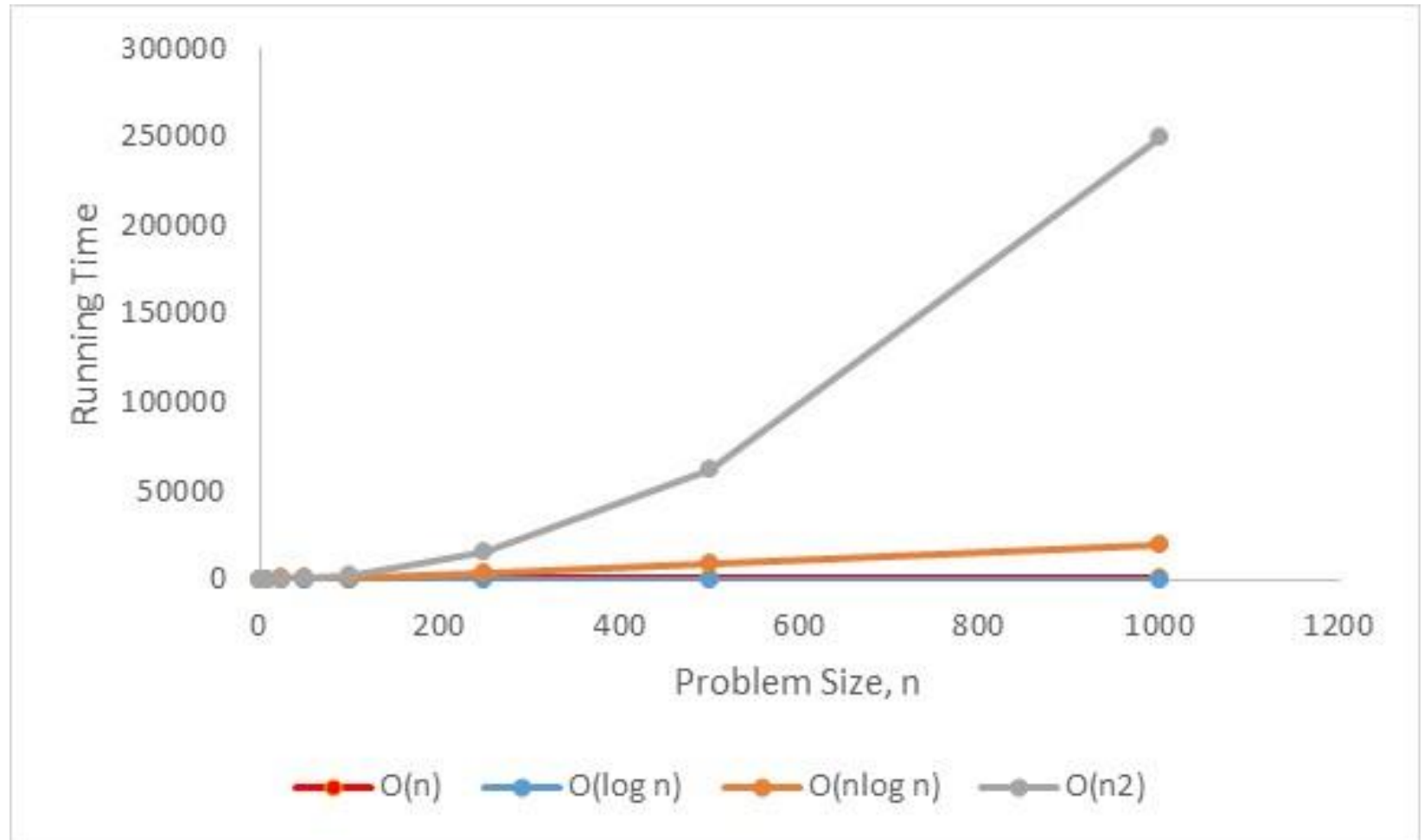
- How many calculations for distances between each pair of n locations?

```
7 import math
8
9 R = 6371.0 * 0.621371
10 def hav_dist(lat1, lon1, lat2, lon2):
11     """ latitude and longitude inputs are in degrees """
12
13     """ convert latitude and longitude to radians """
14     lat1 = lat1 * math.pi / 180.0
15     lon1 = lon1 * math.pi / 180.0
16     lat2 = lat2 * math.pi / 180.0
17     lon2 = lon2 * math.pi / 180.0
18
19     a = math.sin((lat2-lat1)/2)**2 + math.cos(lat1) * math.cos(lat2) * (math.sin((lon2-lon1)/2))**2
20     return R * 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
21
22 """ Store location data: [id, latitude, longitude] """
23 stores = [[0, 36.8176, -76.26684],[1, 36.8368239286, -76.3111960714],[2, 36.784765, -76.25588],[3, 36.80
24 R = 6371.0 * 0.621371
25
26 for i in range(len(stores)):
27     for j in range(i+1, len(stores)):
28         thisDist = hav_dist(stores[i][1], stores[i][2], stores[j][1], stores[j][2])
29         print('Distance Store ' + str(i) + ' to Store ' + str(j) + ': ' + str(thisDist))
```

Why is Big-Oh Important?



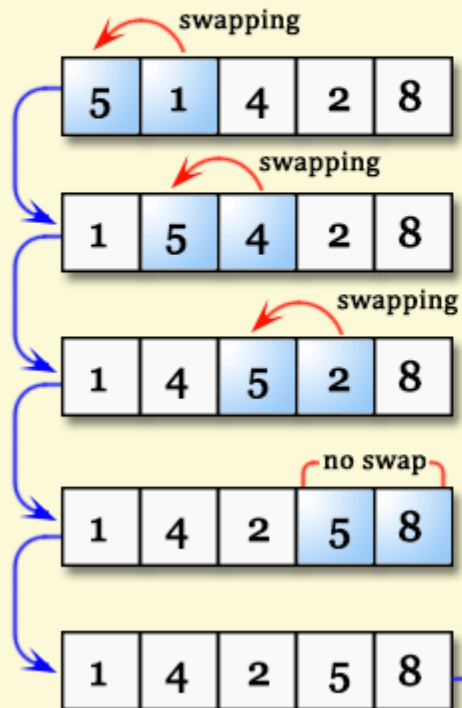
Why is Big-Oh Important?



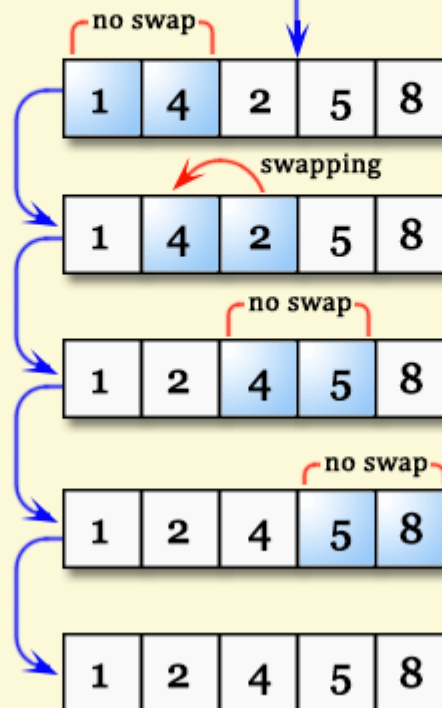
Bubble Sort

Bubble Sorting

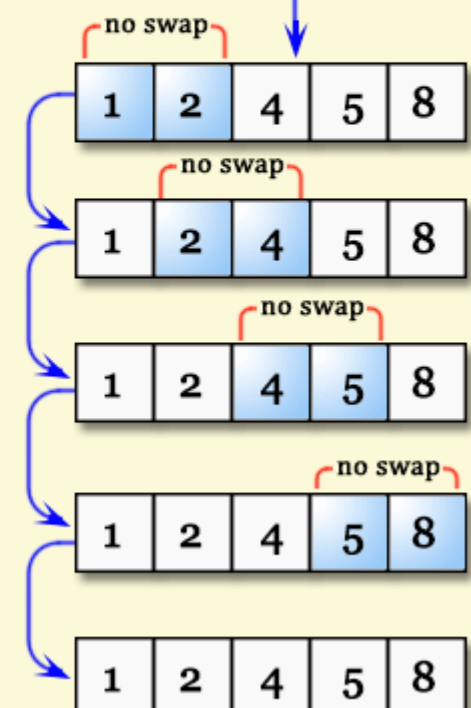
First Pass



Second Pass



Third Pass



Bubble Sort

- Calculate running time = $O(?)$
- What is worst case?
 - Putting an list in descending order into ascending order

8	5	4	2	1
---	---	---	---	---

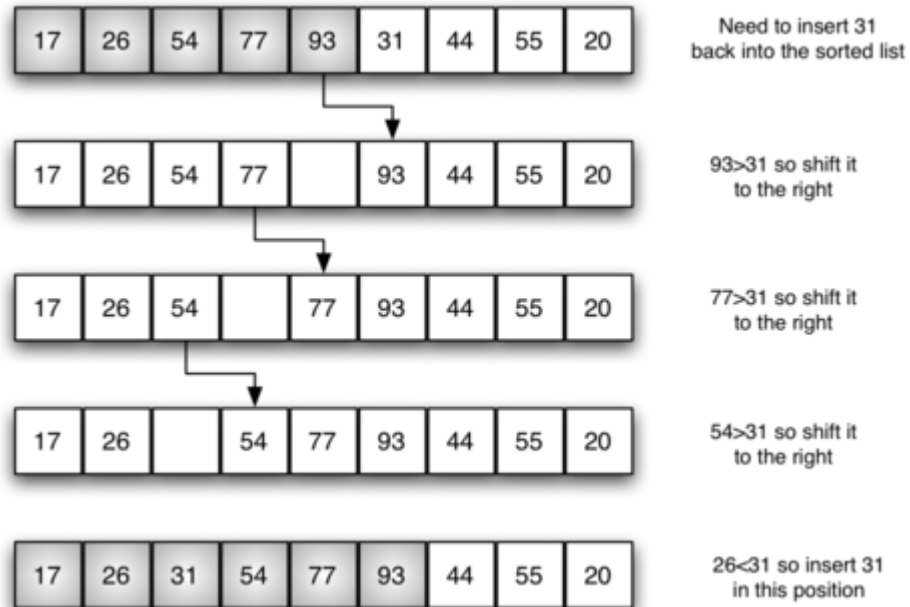
- How many passes are required to sort?
- How many comparisons/swaps are made in each pass?

Insertion Sort

54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

<http://interactivepython.org/XikcZ/courselib/static/pythonds/SortSearch/TheInsertionSort.html>

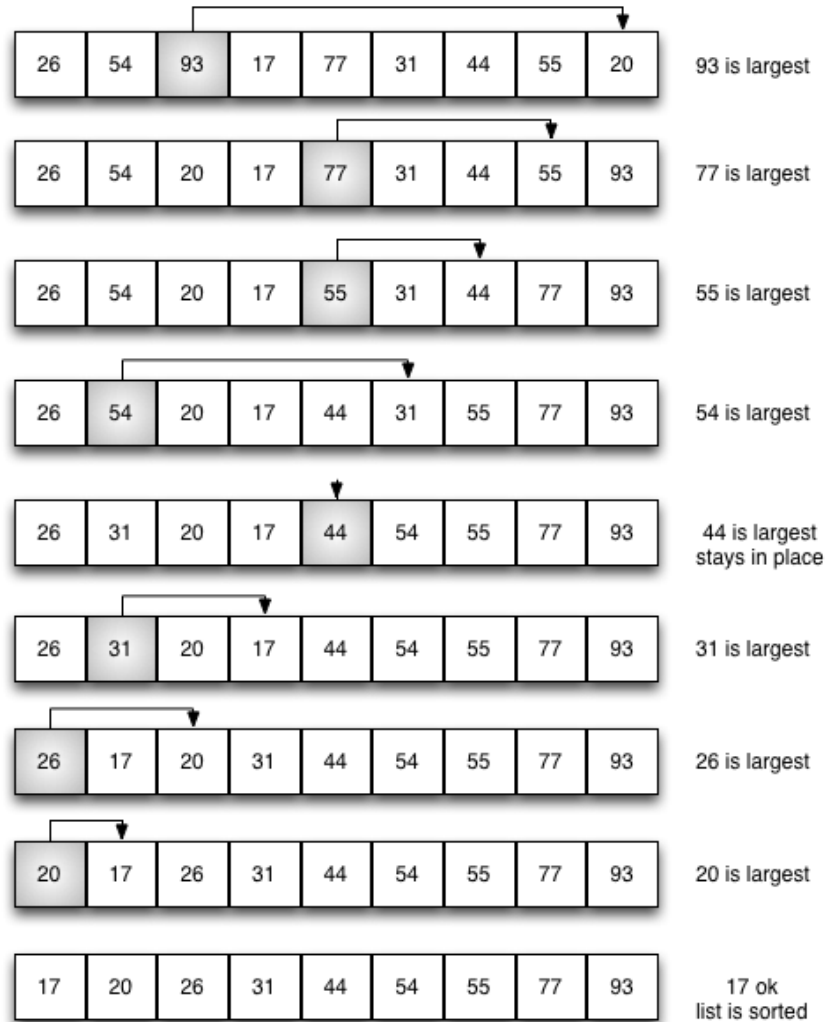
Insertion Sort



Insertion Sort

- Let's calculate the running time, $O(?)$
- How many insertions?
- Worst case number of comparisons/shifts for each insertion?

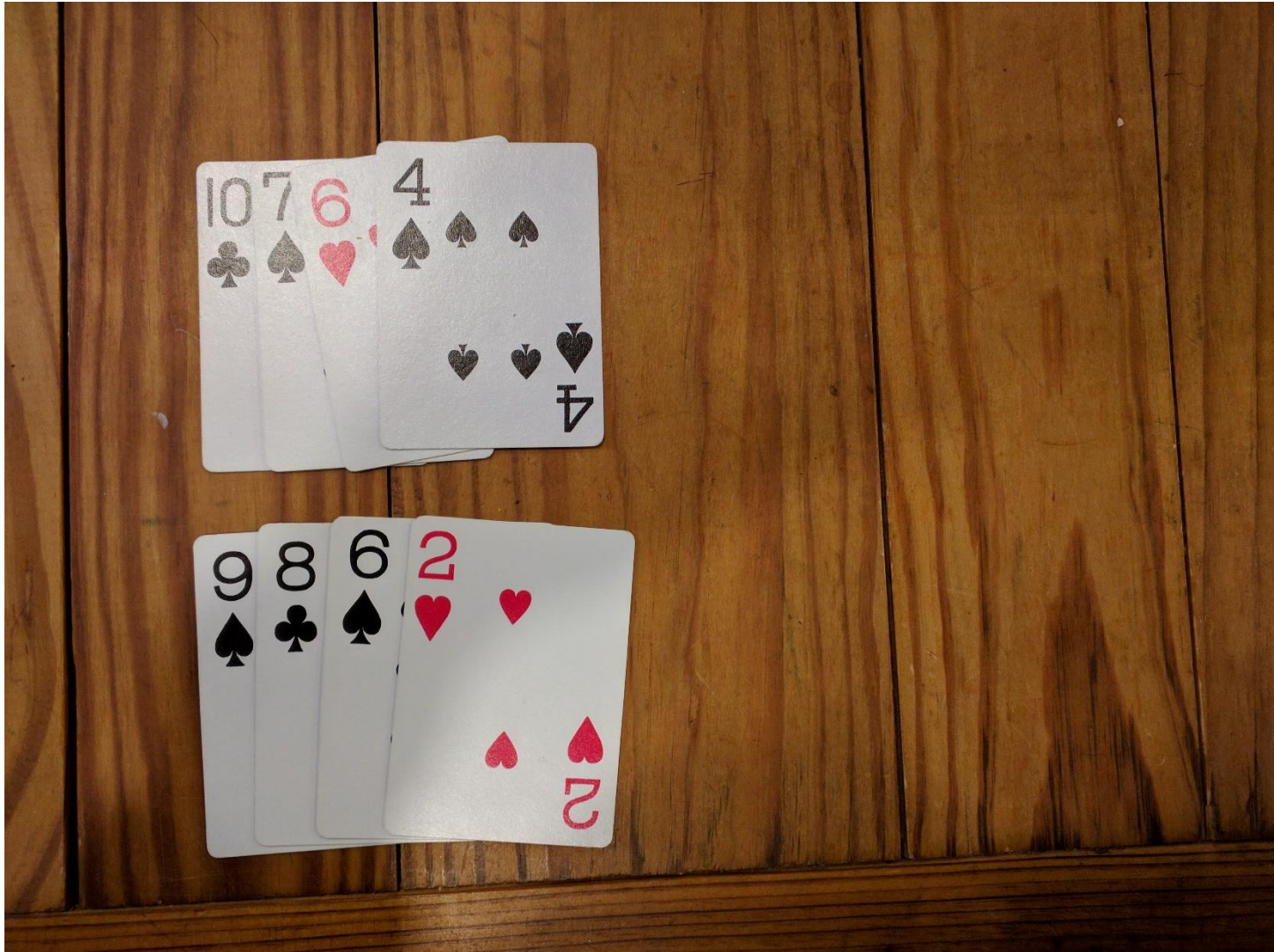
Selection Sort



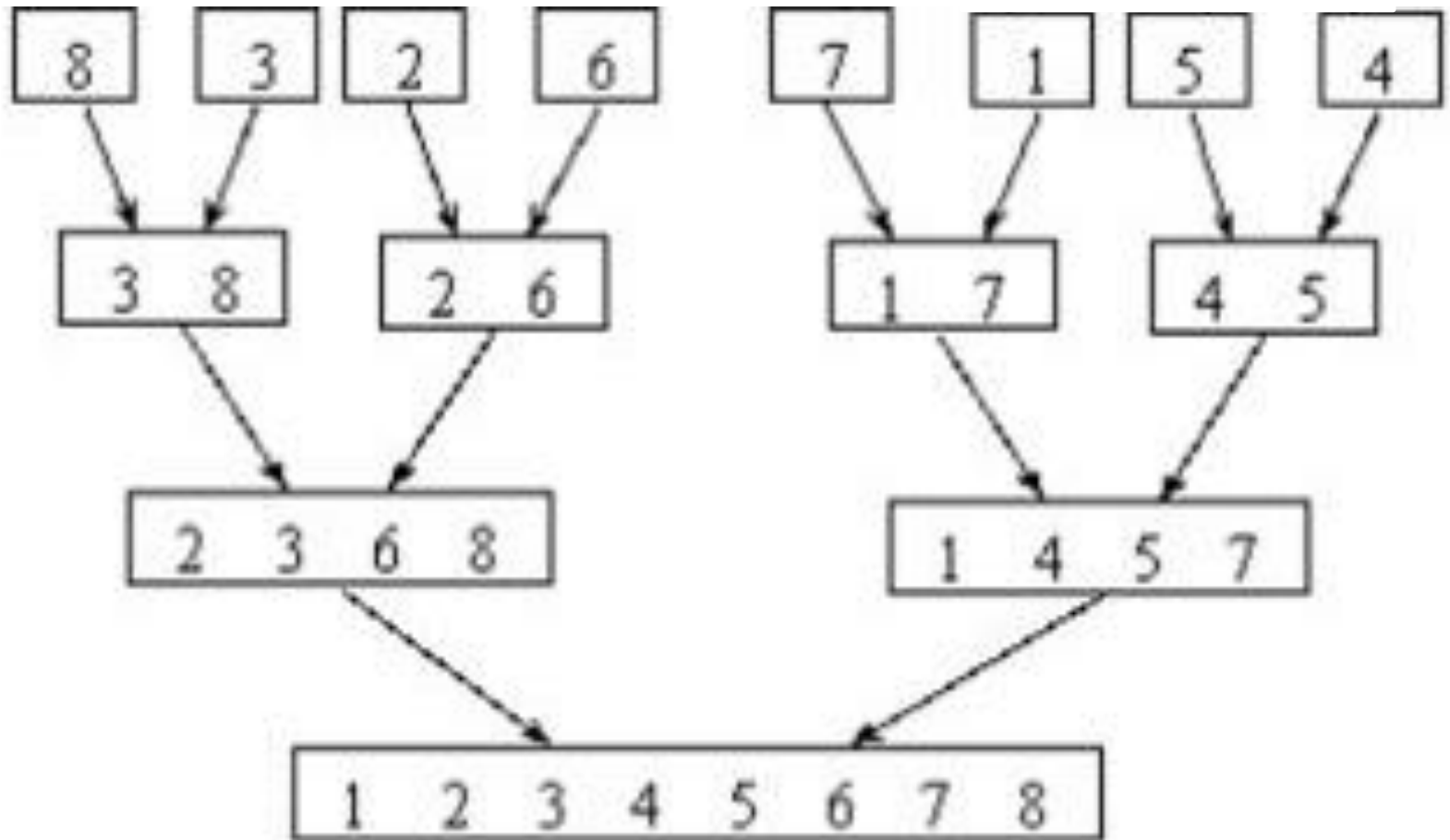
Selection Sort

- Let's calculate the running time, $O(?)$
- How many maximum computations?
 - How many comparisons for each one?
- How many memory changes to move the maximum to its position toward the right of the array?

Merge Sort

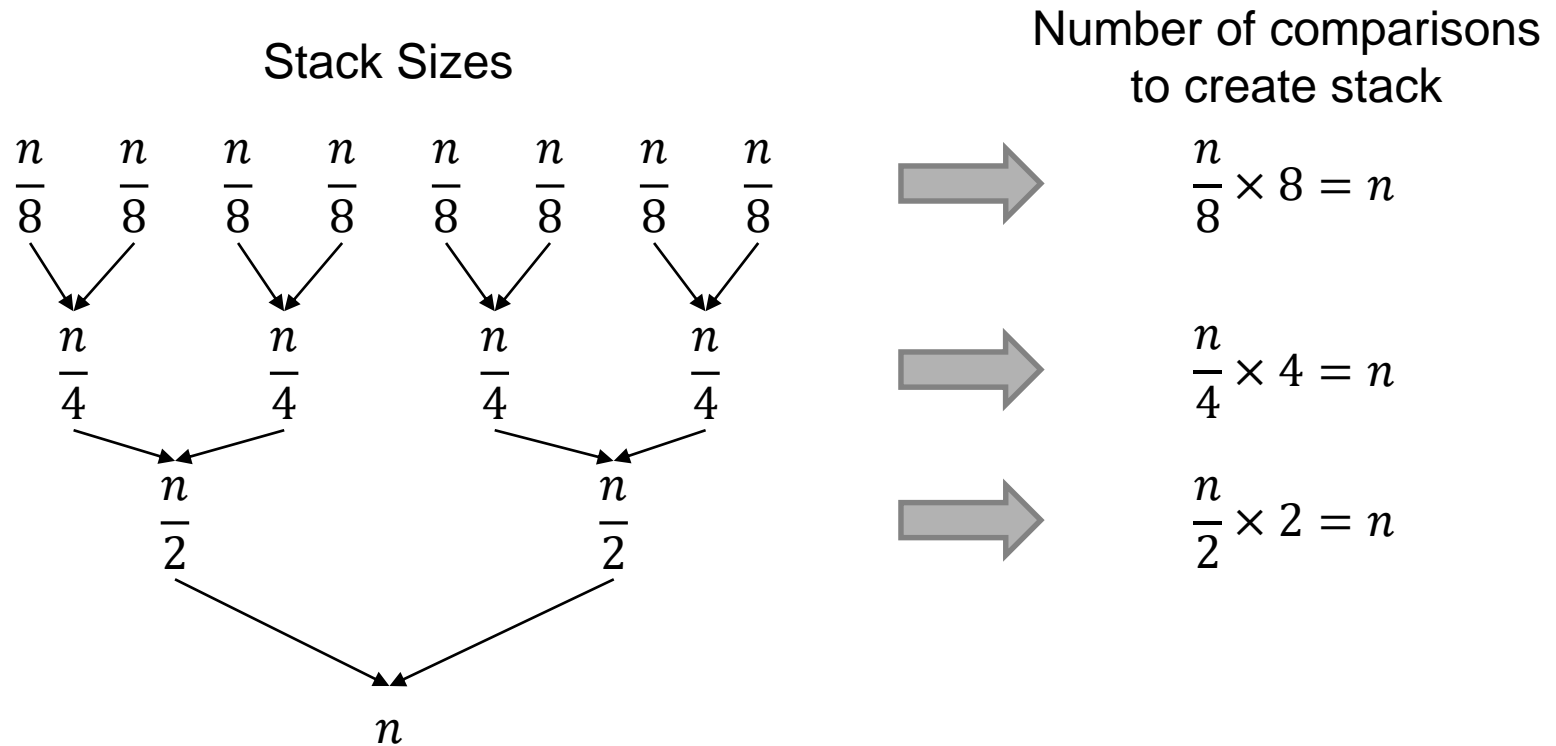


Merge Sort



<http://lifexplorer.me/leetcode-sort-list/>

Merge Sort



- Let's calculate the running time
 - How many comparisons in each step?
 - How many merge steps?

Heap Sort

- “[...] improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.”
- <https://en.wikipedia.org/wiki/Heapsort>

Quicksort

- “[...] it can be about two or three times faster than its main competitors, merge sort and heapsort”
- <https://en.wikipedia.org/wiki/Quicksort>

Sorting Algorithm Running Times

- <http://bigocheatsheet.com/>

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Takeaways

- You should be computing the running times for your algorithms
- You should be aware of the running times of the built-in functions you use
 - Test them to see how fast they are

Python Sorting

- What are the alternative Python methods?
- Native Python `list.sort()`
 - timsort
 - Hybrid merge sort & insertion sort, takes advantage of existing order in list
- Numpy
 - Offers alternative sorting methods
 - <https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html>
- Research is required to find the best/fastest methods

Optimizing Code

- Most of this course is...
 - Getting an algorithm that runs
- Once your algorithm is running...
 - Try to reduce running time
 - Use methods that are fastest
 - Choose among the alternatives
 - It sometime takes research
- This includes simplifying your code

Backup Slides

Python Sorting

- `numpy` offers alternatives

The screenshot shows the SciPy.org documentation page for `numpy.sort`. The page is titled "numpy.sort" and includes a navigation bar with links to "index", "next", and "previous". The function signature is `numpy.sort(a, axis=-1, kind='quicksort', order=None)`. The description states: "Return a sorted copy of an array." The parameters section lists: `a` (array-like), `axis` (int or None, optional), `kind` (string, optional), and `order` (str or list of str, optional). The `kind` parameter is highlighted with a red box. The returns section lists: `sorted_array` (ndarray). The "See also" section lists: `ndarray.sort`, `argsort`, `lexsort`, `searchsorted`, and `partition`. The "Notes" section discusses the various sorting algorithms and their properties.

Parameters:

- `a` : `array_like`
Array to be sorted.
- `axis` : `int` or `None`, optional
Axis along which to sort. If `None`, the array is flattened before sorting. The default is `-1`, which sorts along the last axis.
- `kind` : `{'quicksort', 'mergesort', 'heapsort'}`, optional
Sorting algorithm. Default is `'quicksort'`.
- `order` : `str` or list of `str`, optional
When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

Returns:

- `sorted_array` : `ndarray`
Array of the same type and shape as `a`.

See also:

- `ndarray.sort` : Method to sort an array in-place.
- `argsort` : Indirect sort.
- `lexsort` : Indirect stable sort on multiple keys.
- `searchsorted` : Find elements in a sorted array.
- `partition` : Partial sort.

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	$O(n^2)$	0	no
'mergesort'	2	$O(n \log(n))$	$\sim n/2$	yes
'heapsort'	3	$O(n \log(n))$	0	no

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

Python Sorting

- Reference for other Python sorting utilities
 - <https://docs.python.org/3.6/howto/sorting.html>
 - <https://en.wikipedia.org/wiki/Timsort>
 - <http://stackoverflow.com/questions/10948920/what-algorithm-does-pythons-sorted-use>

Python Sorting Test

Note, using
Python 2.7
back in the
day.

```
7
8 import time
9 import random
10 import numpy as np
11
12 x = range(1000000)
13 random.shuffle(x)
14 print x
15 start_time = time.time()
16 x.sort()
17 print "Time to sort random x.sort():",time.time() - start_time
18
19 xtemp = x[500000]
20 x[500000] = x[500001]
21 x[500001] = xtemp
22 start_time = time.time()
23 x.sort()
24 print "Time to sort x.sort() when it is almost sorted:",time.time() - start_time
25
26 x.sort(reverse=True)
27 start_time = time.time()
28 x.sort()
29 print "Time to sort x.sort() when list is sorted in reverse:",time.time() - start_time
30
31 random.shuffle(x)
32 x_numpy = np.array(x)
33 start_time = time.time()
34 x_numpy.sort()
35 print "Time for numpy quicksort of random data:",time.time() - start_time
36
37 x_numpy = np.array(x)
38 start_time = time.time()
39 x_numpy.sort(kind='mergesort')
40 print "Time for numpy mergesort of random data:",time.time() - start_time
41
42 x_numpy = np.array(x)
43 start_time = time.time()
44 x_numpy.sort(kind='heapsort')
45 print "Time for numpy heapsort of random data:",time.time() - start_time
46
47 random.shuffle(x)
48 start_time = time.time()
49 y = sorted(x)
50 print "Time to sort sorted(x):",time.time() - start_time
51
52 random.shuffle(x)
53 start_time = time.time()
54 x.sort(key = lambda x:x)
55 print "Time to sort x.sort() with lambda key:",time.time() - start_time
56
57 z = [[random.random(),random.random()] for aa in range(1000000)]
58
59 z1 = [(w[1],w) for w in z]
60
61 start_time = time.time()
62 z1.sort()
```

Python Sorting Takeaways

- Lessons
 - Sorting methods take advantage of existing order in a list
 - Use numpy quicksort

```
Time to sort random list: 0.517000198364
Time to sort list that is almost sorted: 0.0140001773834
Time to sort list in reverse order: 0.0160000324249
Time for numpy quicksort of random list: 0.0599999427795
Time for numpy quicksort of almost in order list: 0.00600004196167
Time for numpy mergesort of random list: 0.0769999027252
Time for numpy mergesort of almost in order list: 0.010999917984
Time for numpy heapsort of random list: 0.0989999771118
Time for numpy heapsort of almost in order list: 0.0420000553131
Time to execute sorted(x): 0.742000102997
Time to execute sorted(x) on almost sorted list: 0.0379998683929
Time to sort x.sort() with lambda key: 1.6369998455
Time to sort with decorator sort: 1.882999897
Time to sort tuples using lambda function: 1.42999982834
```