CS1 Lecture 31 Apr. 5, 2017

- HW 7 due Monday morning, 9:00 am.
- Discussion section this week
  - covers basics of making graphs w Pylab as needed for HW7. We won't cover this in lecture (except briefly today)!
- Python with Pylab
  - Probably easiest to use Anaconda (see class website for link)
  - If you want to use Anaconda's IDLE rather than Spyder IDE, simplest way is to start from command line.
    - E.g. in Terminal on Mac:

      /Users/yourloginname/anaconda/bin/idle3

      or, sometimes, /anaconda/bin/idle3
- Exam 2: April 20, 6:30-8:00pm
- Just announced: Turing Award: Tim Berners-Lee

Last time

- Sorting introduction – selection sort, insertion sort

Today

- Faster sorting methods

HW7

Compare sorting methods

- use Pylab to make charts/graphs of their running time behavior
- write about results

making meaningful graphs is often not easy

- You must experiment to find good sizes for data
  - You should test on large enough data to clearly understand differences/similarities
  - You probably can't put all results on one graph
    - At a certain size level, growth differences of some algorithms will be clear but distinctions between others not yet clear. Drop some and continue with larger sizes for ones not yet clearly distinguished
- You should test on data of various kinds – unsorted vs already sorted
- This all takes time and thought

HW7: you can use functions like this:

```
def testInsertionSort(minN = 1000, maxN=20000, step=2000):
    listSizes = list(range(minN, maxN, step))
    runTimes = []
    for listSize in listSizes:
        listToSort = mixup(list(range(listSize)))
        startTime = time.time()
        insertionSort(listToSort)
        endTime = time.time()
        runTimes.append(endTime-startTime)
    pylab.figure(1)
    pylab.clf()
    pylab.plot(listSizes, runTimes, 'ro-')
```

lec31.py

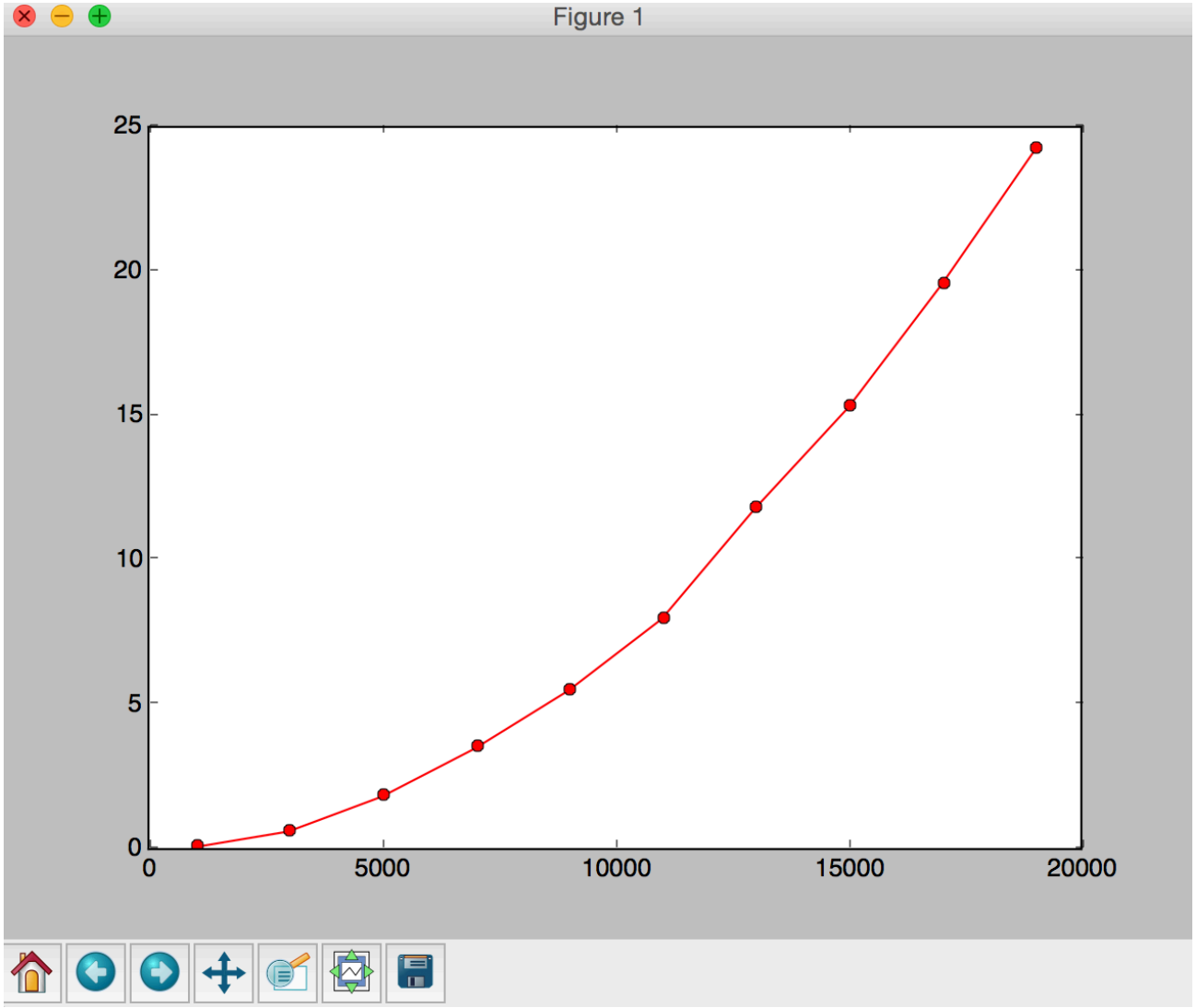


Figure 1
25
20
15
10
5
0
0
5000
10000
15000
20000

Sorting (https://www.youtube.com/watch?v=k4RRi_ntQc8)

It's mostly a "solved" problem – available as excellent built-in functions – so why study? The variety of sorting algorithms demonstrate a variety of important computer science algorithmic design and analysis techniques.

Sorting has been studied for a long time. Many algorithms: selection sort, insertion sort, bubble sort, radix short, Shell short, quicksort, heapsort, counting sort, Timsort, comb sort, bucket sort, bead sort, pancake sort, spaghetti sort … (see, e.g., wikipedia: sorting algorithm)

Why sort? Searching a sorted list is very fast, even for very large lists (*log n is your friend*). So if you are going to do a lot of searching …

So, should you always sort? (Python makes it so easy …)

- We can search an unsorted list in O(n), so answer depends on how fast we can sort. We certainly can't sort fast than linear time (must look at, and maybe move, each item). In fact, in general we cannot sort in O(n). Best "comparison-based" sorting algorithms are O(n log n)
- So, when should you sort? If, for example, you have many searches to do. Suppose we have n/2 searches to do.
  - n/2 linear searches → n/2 * O(n) → $O(n^2)$
  - sort, followed by n/2 binary searches → O(n log n) + n/2 * O(log n) → O(n log n) + O(n log n) → O(n log n) *for large n, this is much faster*

Sorting – selection sort


```
def selectionSort(L):
    for i in range(len(L)):
        # swap min item in unsorted region with ith
        # item
```

Sorting – insertion sort

- Slightly different main step picture than for selection sort

Given:

L[0:i] sorted (but not necessarily in final position)

L[i:] unsorted

How do we "grow" solution?

Move L[i] into correct spot (shifting larger ones in L[0:i] one slot to the right

Running time of selection sort and insertion sort

- Selection sort
  - $O(n^2)$ always – worst, best, average case. It always searches the entire unsorted portion of the list to find the next min. No distinction between best/worst/average cases.

- Insertion sort
  - In best case, while loop never executes, so O(n)
  - In worst case, while loop moves ith item all the way to L[0]. This yields the familiar sum, 0 + 1 + 2 + … + n, once again. Thus, $O(n^2)$.
  - Average case is also $O(n^2)$
  - Among $O(n^2)$ sorts, insertion sort is good one to remember. In practice, it works well on "almost sorted" data, which is common. It is sometimes used as a "finish the job" component of hybrid sorting methods – use an O(n log n) sorting method until the list is "almost sorted, then switch to insertion sort to finish.

How might we sort faster than insertion sort, selection sort, and other simple sorts? Can we do better than O(n^2)?

Try a **divide-and-conquer** approach:

- Divide problem into subproblems
- Solve subproblems recursively
- Combine results

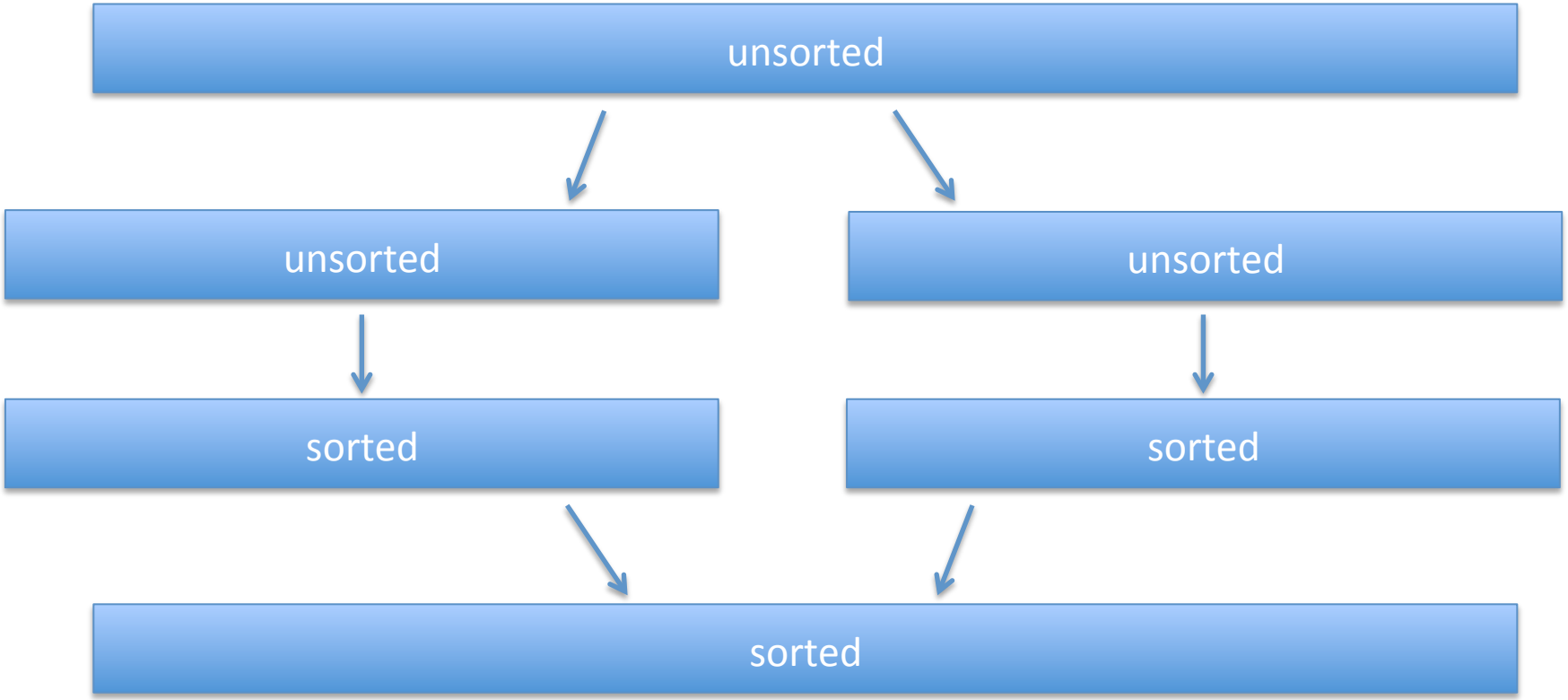
Binary search is a special case of divide and conquer.

- Check middle element and create one subproblem of half the size
- This yielded speed-up from O(n) to O(log n)

Many problems benefit from divide and conquer approach

How can we use divide-and-conquer to sort?

1. divide list into two (almost) equal halves
2. sort each half recursively!
3. combine two sorted halves into fully sorted result



Is it clear how to implement each step?

Sorting by divide and conquer

1. divide list into two (almost) equal halves
   - O(1)
2. recursively sort each half
   - Make two recursive calls
3. combine two sorted halves into fully sorted result
   - Merge algorithm -> O(n)

Merge sort

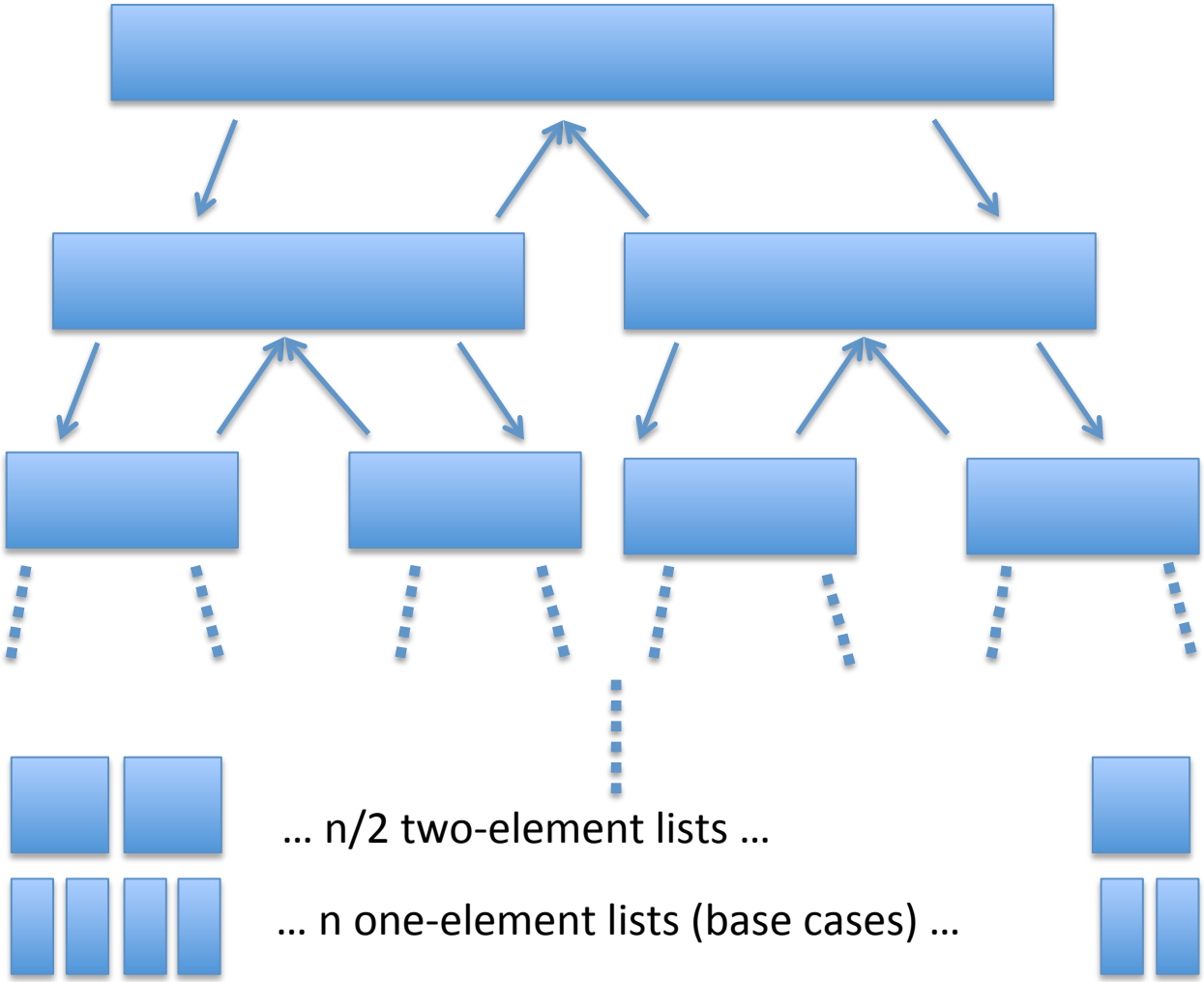
- Implementation in lec31.py
- Running time?
  - In more advanced computing classes, learn about things like recurrence equations. Can express running time via:

    T(1) = c

    T(n) = time for recursive calls + divide time + combine time

    = 2 * T(n/2) + O(1) + O(n)
  - we can also analyze by looking at "recursion tree" and adding up times …
  - Get?

Recursion tree for analyzing merge sort running time



T_{div} T_{merge}

$1 * 1 + n = O(n)$

$2 * 1 + 2 * n/2 = O(n)$

How deep does this go? How many levels? Log n

… n/2 two-element lists …

… n one-element lists (base cases) …

$n/2 * 1 + n/2 * 2 = O(n)$

$n * O(1) = O(n)$

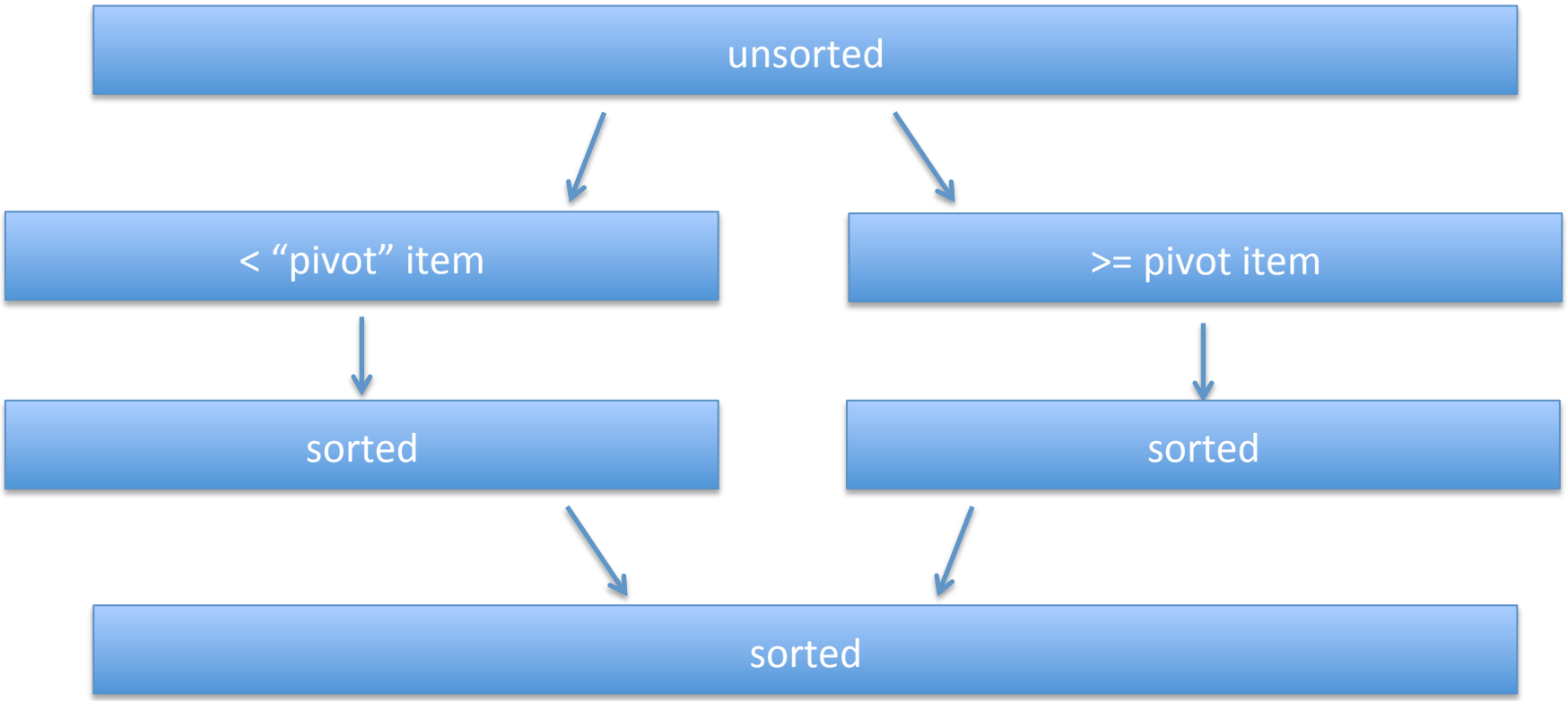
Total: O(n) * # of levels = ? O(n log n)

Mergesort

- Merge sort is very famous sorting algorithm. Classic example of power of divide and conquer. Yields "optimal" O(n log n) sort. Much faster than $O(n^2)$ algorithms.
- Can be implemented non-recursively (but most people find the rec. version much easier to implement)
- One possible concern? Standard implementations require O(n) additional space
- Python's built-in sort is (in most Python implementations) Timsort (named after Tim Peters), a hybrid drawing from mergesort and insertion sort. Has O(n log n) average and worst-case time (like mergesort) and O(n) best case time (like insertion sort)
  - *news last year?!*
    Bug found in Timsort implementations in Python, Java, Android! Has been there for a while …

Another divide-and-conquer style sorting algorithm: quicksort

1. divide list into two parts, one containing all elements < some number, the other containing all elements >= that number
2. Recursively sort parts
3. combine two sorted halves into fully sorted result



Is it clear this time how to implement each step?

Quicksort

- Divide step
  - mergesort: easy – just cut in half O(1)
  - quicksort: takes work – scan through entire list putting elements in > or < (vs pivot) part O(n)
- Combine step
  - mergesort: takes work – step through subproblem solutions, merging them O(n)
  - quicksort: easy, we're done! Just put parts together O(1)
- Subproblems
  - mergesort: the two subproblems always half the size
  - quicksort: subproblem size depends of value of item you choose as pivot. What pivot would yield half-sized subproblems? Can you find that pivot item easily?
    - poor pivot choices can yield poor sorting performance
    - good practical pivot choices: 1) median of first, middle, last items, 2) random item

Quicksort

- Quicksort has worst case running time of $O(n^2)$
- *BUT* average case O(n log n) and if we choose pivot properly we can make worst case very very unlikely. (The detailed mathematical analysis of this is not so easy).
- Unlike mergesort, is "in place" – does not require O(n) extra space).
- When implemented properly is excellent and very commonly used sorting method in the real world.
- Be careful if you implement it yourself. *Easy to get slightly wrong*. E.g. the code at the first (and second) result returned when I googled - quicksort python – is c*orrect* (in that it properly sorts) but a poor implementation because it chooses pivot badly (try it on an already sorted list of, say, 10000 or more elements, or even a list containing many many identical elements!?) – qsbad.py
- Good standard pivot choice – "median of three" – choose as pivot the median value among first, middle, and last elements in the given list. E.g. for [15, 4, 2, 99, 6, 3, 25, 26, 8], choose median of 15, 6, 8, which is 8
  - In this case, good – 4, 2, 3, 6 will be in "less-than" partition, 15, 99, 25, 26 will be in "greater-than" partition.

Many visualizations of sorting algorithms on the web:

- http://www.sorting-algorithms.com, http://sorting.at, https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- https://www.youtube.com/watch?v=kPRA0W1kECg
- https://www.youtube.com/watch?v=ROalU379l3U (dance group demonstrating sorting algorithms …)

Next time

- Graph (the computer science-y ones, not the x-y plots you are using in HW7) algorithms