CS1210 Lecture 31 Nov. 3, 2021

- HW 7 due Monday
- Quiz 4: Nov. 19

Last time

- selection sort
- insertion sort
- demo of plotting/graphing using matplotlib/pylab

Today

- continue sorting - merge sort, Quicksort

selection sort

Given:

L[0:i] sorted and in final position

L[i:] unsorted

How do we “grow” solution?

Find min in unsorted part and move it to position i

Sorting

- Another simple approach – 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

Idea: repeatedly move first item in unsorted part to proper place in sorted part

5 23 -2 15 100 1 8 2

Sorted	Not yet sorted
	5 23 -2 15 100 1 8 2
5	23 -2 15 100 1 8 2
5 23	-2 15 100 1 8 2
-2 5 23	15 100 1 8 2
-2 5 15 23	100 1 8 2
-2 5 15 23 100	1 8 2
-2 1 5 15 23 100	8 2
-2 1 5 8 15 23 100	2
-2 1 2 5 8 15 23 100	

Insertion sort

- running time of insertion sort?
  - best case?
    - sorted already O(n)
  - worst/average case?
    - $O(n^2)$

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, inner while loop never executes, so O(n)
  - In worst case, inner 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.

HW 7 asks you to compare sorting methods and use Pylab to make charts/ graphs of their running time behavior

Making meaningful graphs is often not easy

- experiment to find good sizes for data
  - test on large enough data to clearly understand differences/similarities (for some sorts, need lists hundreds of thousands and/or millions long)
- Experiment on sorted, reverse sorted, nearly sorted, random data

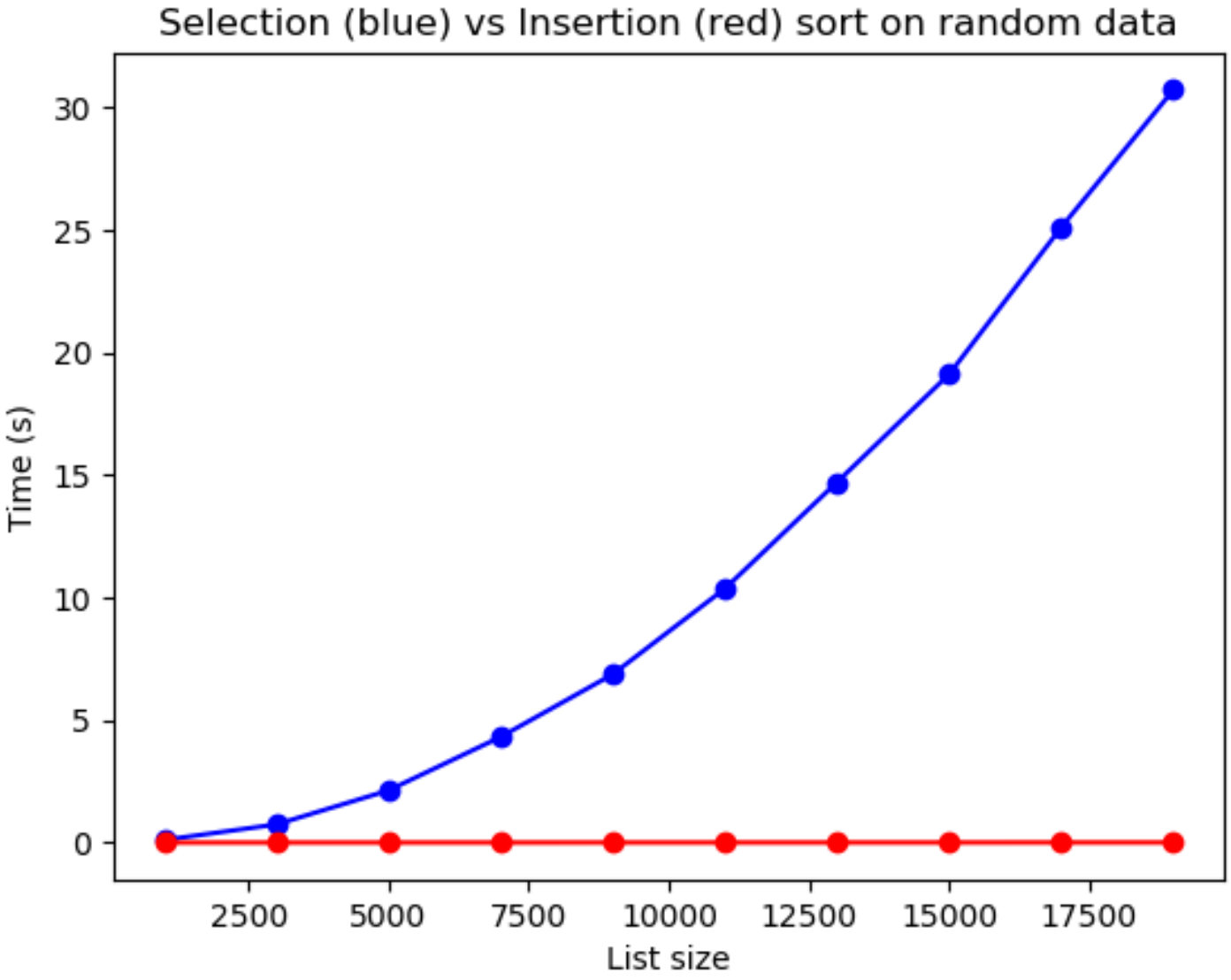
Comparing sorting functions via timing and graphing

```
def compareSorts(minN = 1000, maxN=20000, step=2000):
    listSizes = list(range(minN, maxN, step))
    selectionSortTimes = []
    insertionSortTimes = []
    for listSize in listSizes:
        listToSort = mixup(list(range(listSize)))
        startTime = time.time()
        selectionSort(listToSort)
        endTime = time.time()
        selectionSortTimes.append(endTime-startTime)
        startTime = time.time()
        insertionSort(listToSort)
        endTime = time.time()
        insertionSortTimes.append(endTime-startTime)
    pylab.figure(1)
    pylab.clf()
    pylab.xlabel('List size')
    pylab.ylabel('Time (s)')
    pylab.title("Selection (blue) vs Insertion (red) sort on random data")
    pylab.plot(listSizes, selectionSortTimes, 'bo-')
    pylab.plot(listSizes, insertionSortTimes, 'ro-')
```

Lec31a.py

Lec31b.py

Comments on this result??



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.

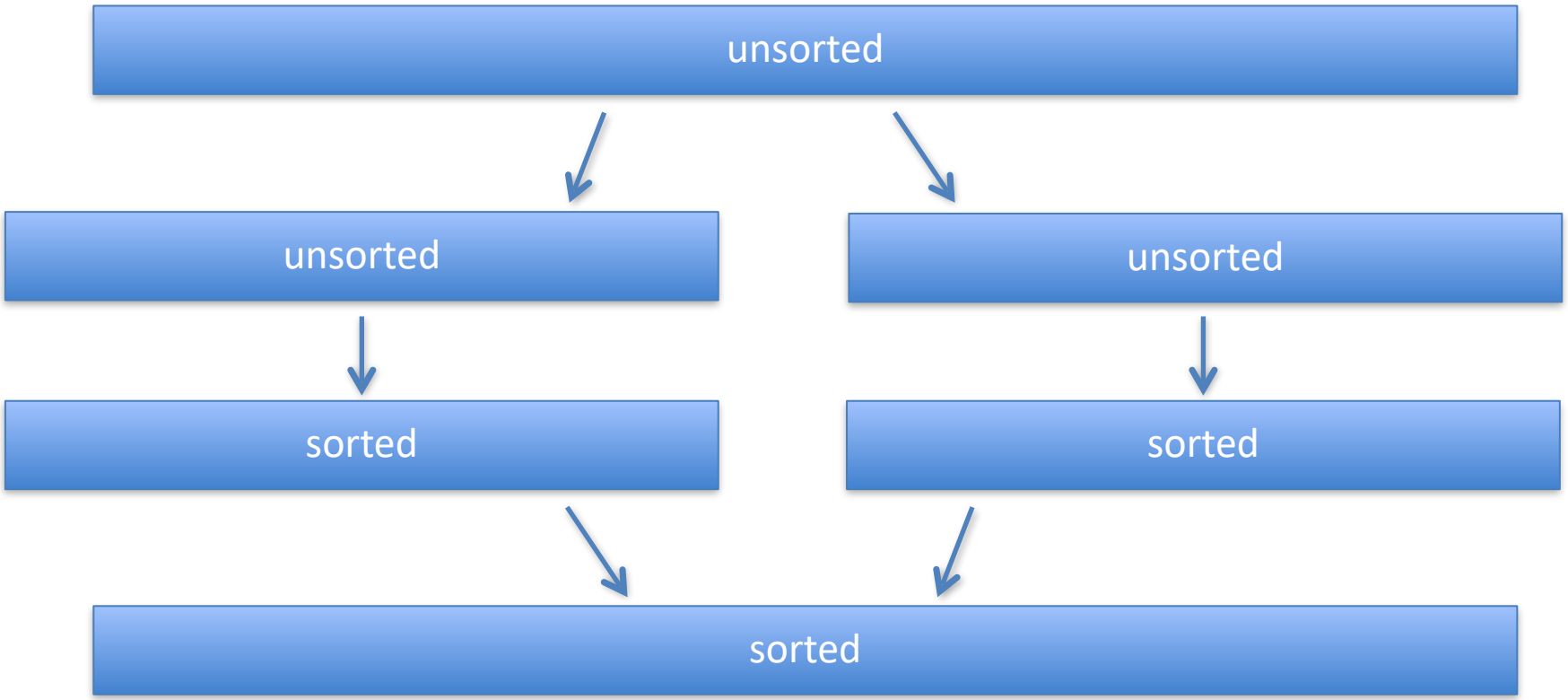
- Check middle element and create one subproblem of half the size
- This yielded speed-up from O(n) to O(log n)

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
3. combine two sorted halves into fully sorted result

recursively!



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

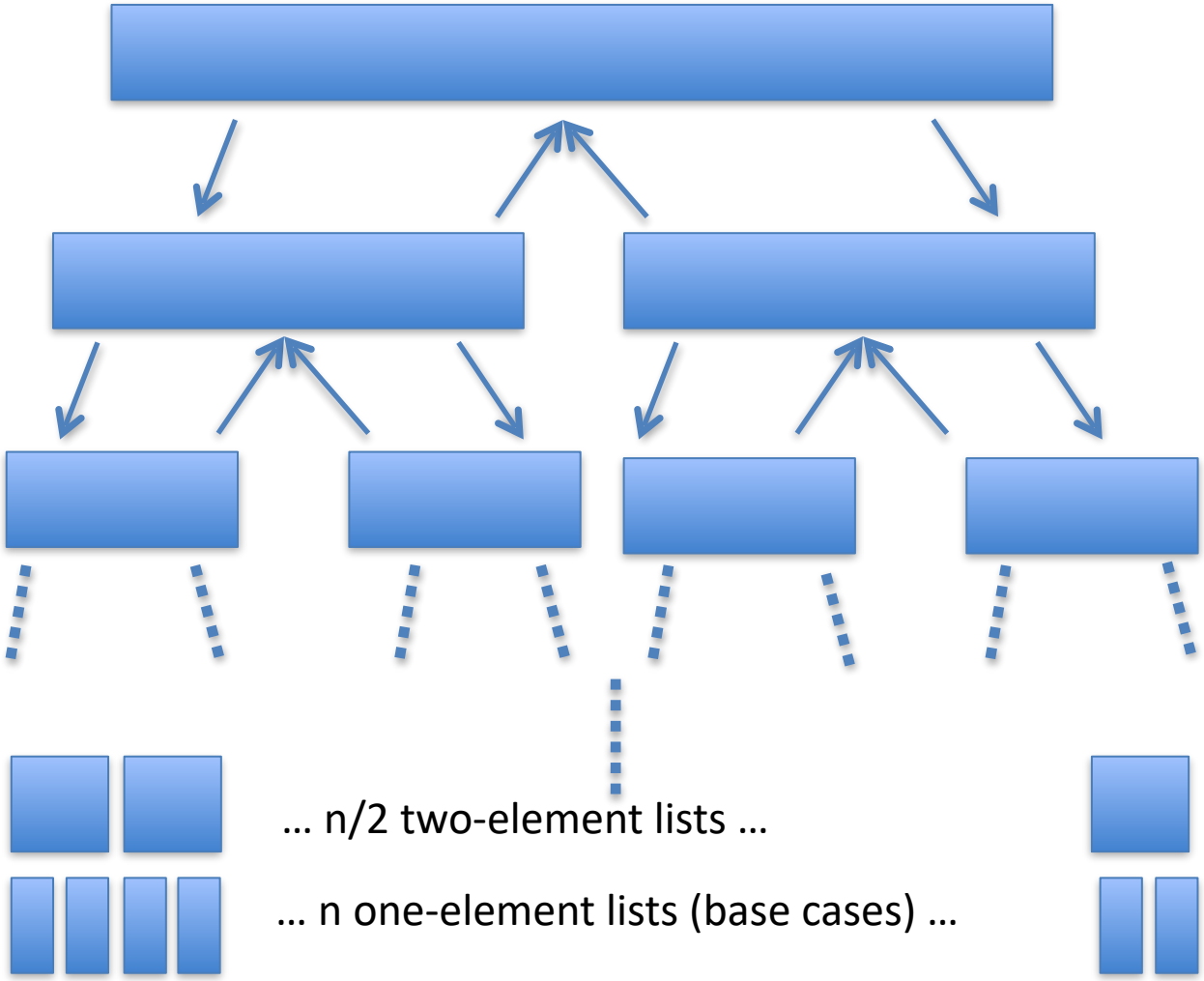
- Implementation in hw7sorts.py
- Running time?
  - In more advanced computing classes, learn about things like recurrence equations. Can express running time via:

    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}

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

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

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

… n/2 two-element lists …

… n one-element lists (base cases) …

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

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

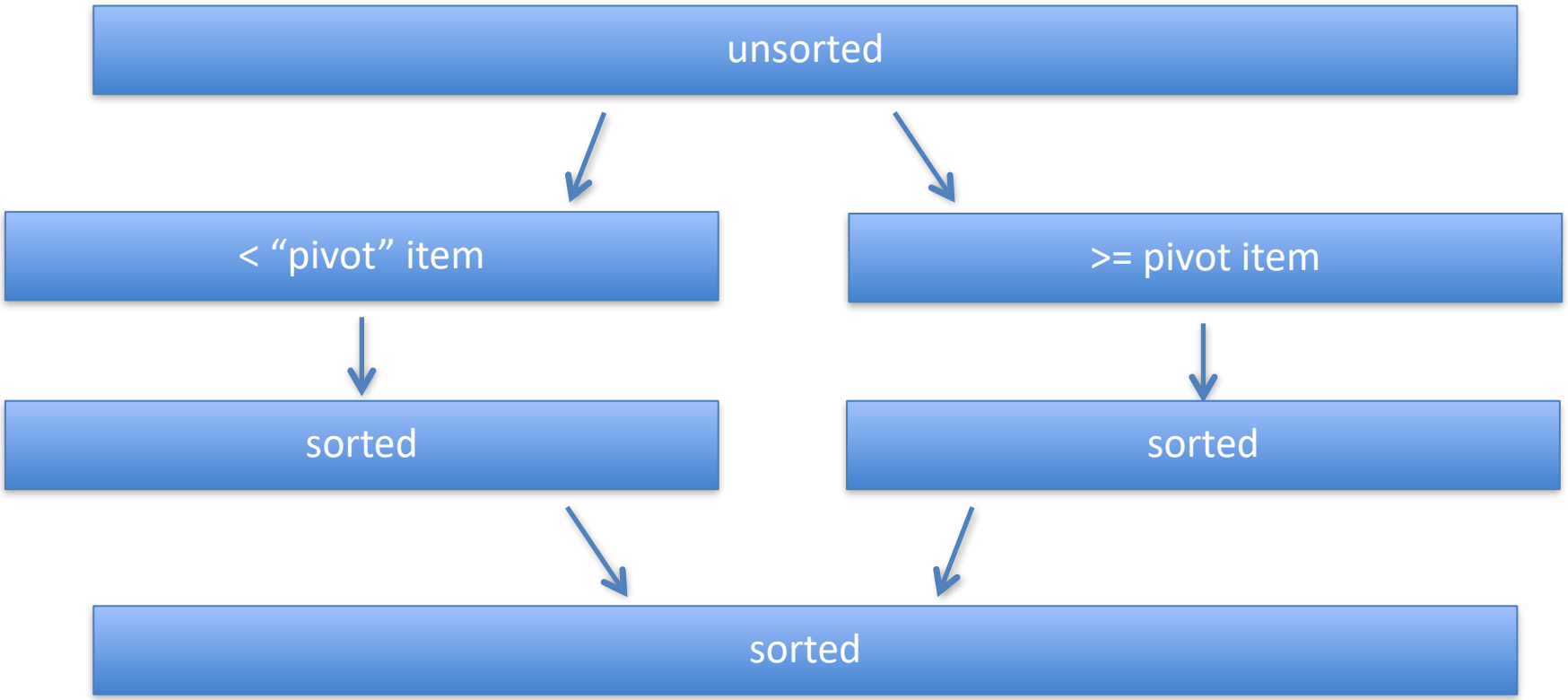
Total: O(n) * # of levels = ? O(n log n)

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 in 2015?!* Bug found in Timsort implementations in Python, Java, Android! Had 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 (called "the pivot"), the other containing all elements >= the pivot 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

Possible quicksort of: [15, 4, 2, 99, 6, 3, 25, 26, 8]

If initial pivot is 8, divide into:

4, 2, 6,3 8, 15, 99, 25, 26

After those are (recursively) sorted

2, 3, 4, 6 8, 15, 25, 26, 99

Combine for result:

2, 3, 4, 6, 8, 15, 25, 26, 99

Possible quicksort of: 15 4 2 99 6 3 25 26 8

If instead 99 chosen as pivot, divide into:
15 4 2 6 3 25 26 8 99

After those are (recursively) sorted
2 3 4 6 8 15 25 26 99

Combine for result:
2 3 4 6 8 15 25 26 99

Still "works" so why do I tell you to worry about pivot choice? Because if you work out the math, having mostly pretty even splits yields O(n log n) running time, while having most very uneven splits yields (n^2) running time. With uneven splits, you can get n levels of recursion rather than log n levels. And since each level takes O(n) work (mainly to do the partition), that yields O(n^2) instead of O(n log n).

SO, pivot choice is VERY important in quicksort. Good pivot choice -> great sorting method. Poor pivot choice – better to just use insertion sort or some simpler O(n^2) sort.

And simple choices of pivot – e.g. first element or last element, yield terrible behavior in simple, reasonably common situations such as the list already being sorted.

Analysis with helpful diagrams at, e.g. https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

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, 8, 15, 99, 25, 26 will be in "greater-or-equal" partition.

A bad quicksort example - qsbad.py

- Demo
  - Load testquicksort into Spyder/Anaconda
  1. Execute testQuicksort()
     - This will produce graph for lists of size 1000, 3000, … 19000, with randomly ordered data
     - Result? nice and fast
  2. Execute testQuicksort(alreadySorted = True)
     - Hmm … what is going on?
  3. To investigate, let's try smaller lists … execute testQuicksort(minN = 100, maxN = 999, step = 100, alreadySorted = True)
     - Result? It *works* this time. Running time behavior? A little hard to see with only these small lists but is O(n^2)
  - OVERALL? qsbad.py chooses pivot poorly. On already sorted data, the left part is all really big and the right part is basically empty. This yields O(n^2) behavior on small lists but eventually crashes on big lists due to limited default recursion depth in Python. A *good* quicksort will *not* have this recursion depth problem (recursion is *not* the problem – it's the pivot choice!)
  - FIX? Easy. Will demo in code but won't write it down in posted slides or posted code.
  4. After fix, execute testQuicksort(maxN = 100000, step = 5000, alreadySorted = True)
     - Result? Good performance – no problems!

- Many visualizations of sorting algorithms on the web:
  - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html, http://www.sorting-algorithms.com, http://sorting.at
  - https://www.youtube.com/watch?v=kPRA0W1kECg
  - dance group demonstrating sorting algorithms:
  - https://www.youtube.com/watch?v=ROalU379l3U
  - https://www.youtube.com/watch?v=ywWBy6J5gz8 (but poor pivot choice! ☺)

- Next time:
  - Greedy algorithms
  - Begin optimization and graph algorithms