CS1210 Lecture 30 Nov. 1, 2021

- Quiz 3 has been graded

Score	0 (most didn't take)	1-3	4-6	7-9	10-12	13-15	16-18	19-20
# of people	18	3	16	21	22	29	25	14

Median: 12

High: 20 (five people)

- **Grades so far:** scores shown/discussed on separate pdf. Total graded points so far: 109. 30 from HW, 21 from DS, 58 from quizzes. Remaining: 59. 30 from HW, 9 from DS, 20 from quiz 4. 32 from (optional) final. Total: 168 without final, 200 with final
- **Quiz 4:** Nov. 19
- DS8 is available, due Wednesday by 8pm. Attendance at actual Tuesday discussion section is optional
  - It is easy but requires you to use the **pylab** module. So you need to use an IDE that includes pylab or figure out how to install pylab in IDLE or Wing or whatever you use. Attend DS tomorrow to get help with that if necessary
- HW7 will be available later today. Very different from other HW assignments. Involves writing some code to compare sorting algorithms and answering questions about the result in written form (i.e. actually writing some sentences!)

Today

- Start sorting: selection and insertion sort

(from last week) Asymptotic notation

Big O notation is used to give an *upper bound* on a function's asymptotic growth or order of growth (growth as input size gets very large)

- if we say f(x) is $O(x^3)$, or f(x) is in $O(x^3)$, we are saying that f grows no faster than x^3 in an asymptotic sense.
- 100 x^3, *.001x^3, 23x^3 + 14 x^2, and x^3 all grow at the same rate in the big picture – all grow like x^3. They are all $O(x^3)$*

Asymptotic notation

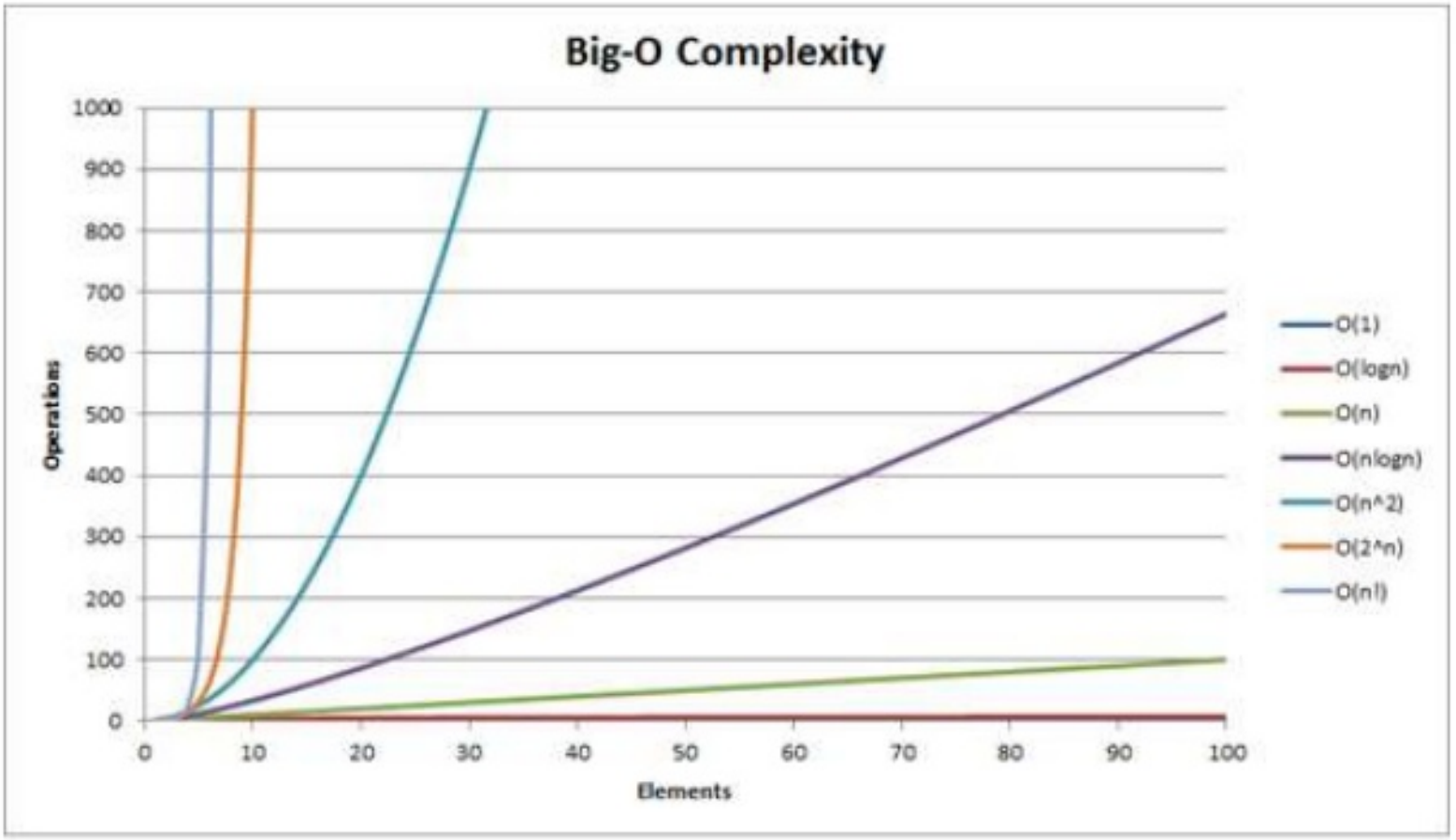
It is also very important to keep in mind that we can separately characterize the best, worst, and average case running times!

For linearSearch:

- Best case? O(1)
- Worst case? O(n)
- Average case? O(n)

Many many students forget this distinction. O is an upper bound, a guarantee on growth rate of something. Best, worst, average case running times are three different somethings about which you can make three big-O statements.

Comparison of Running times



Algorithm Analysis

- Approximate time to run a program with n inputs on 1GHz machine:

	n = 10	n = 50	n = 100	n = 1000	n = 10^6
O(n log n)	35 ns	200 ns	700 ns	10000 ns	20 ms
O(n^2)	100 ns	2500 ns	10000 ns	1 ms	17 min
O(n^5)	0.1 ms	0.3 s	10.8 s	11.6 days	3x10^{13} years
O(2^n)	1000 ns	13 days	4 x 10^{14} years	*Too long!*	*Too long!*
O(n!)	4 ms	*Too long!*	*Too long!*	*Too long!*	*Too long!*

Important complexity classes

Some big-O cases:

- O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), O(2^n), O(2^n), O(n!), and even O(2^{2^n})
- Try to get a feel for which are "good" (or good enough specifications of your particular problem)
- Often, very useful to try to redesign algorithm to convert a factor of n to a log n. O(n^2) → O(n log n)
- Exponential algorithms are *very* slow except for very small inputs. For any but toy problem sizes, you usually need a different algorithm (and sometimes need a whole different approach – aiming for an approximate or heuristic solution rather than an optimal/complete/ perfect one).

Next

- Basic sorting methods
  - Selection sort, insertion sort
- Introduction to plotting w matplotlib/pylab
- more efficient sorting
  - Merge sort, quicksort

This week's discussion section assignment, DS8, and homework, HW7

- Will use Pylab module to plot charts/graphs. Modules/packages like Pylab can be annoying to install. I *strongly recommend* you download the free Anaconda distribution (Python + Spyder IDE plus *many* pre-installed packages) from anaconda.com for use in these assignments.
  - https://www.anaconda.com/products/individual

HW 7 will ask 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, random data

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, sorting is often excellent prep.

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.
- How fast can we sort? Certainly not faster 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

- Python built-in methods, functions
  - myList.sort()
  - sorted(mylist)
  - sorted(mylist, key=lambda item: item[2])
- first, a simple sort
  - how you would sort if given, say, a big list of numbers written on a page? How would you write down the sorted version of the list: 5 23 -2 15 100 1 8 2?

5 23 -2 15 100 1 8 2 → -2 1 2 5 8 15 23 100

Idea: repeatedly find min in unsorted part and move it to sorted

5 23 -2 15 100 1 8 2

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

Sorting – selection sort

Given:

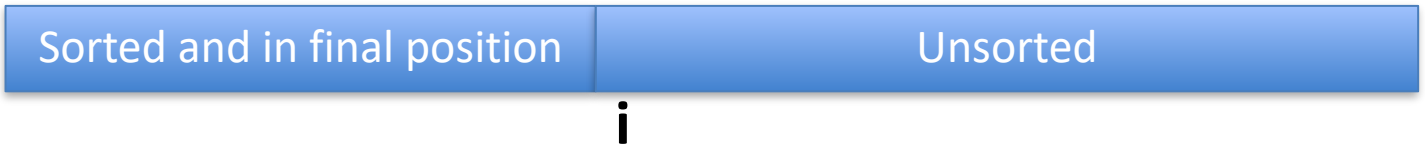
L[0:i] sorted and in final position

L[i:] unsorted

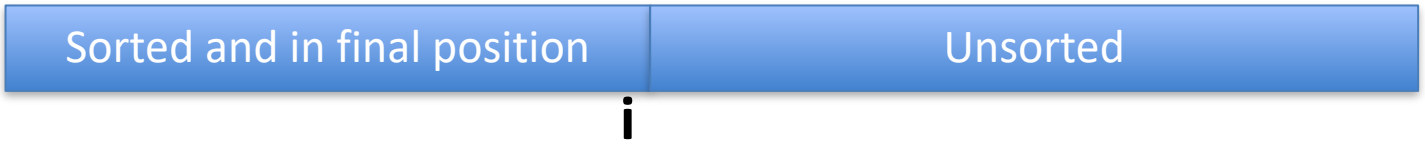
How do we “grow” solution?

Find min in unsorted part and swap it with item currently at position i

Sorting – selection sort




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



Sorting – selection sort


```
def selectionSort(L):
    i = 0
    # assume L[0:i] sorted and in final position
    while i < len(L):
        minIndex = findMinIndex(L, i)
        L[i], L[minIndex] = L[minIndex], L[i]
        # now L[0:i+1] sorted an in final position.
        # Reestablish loop invariant before continuing.
        i = i + 1
        # L[0:i] sorted and in final position
```

```
# return index of min item in L[startIndex:]
# assumes startIndex < len(L)
#
def findMinIndex(L, startIndex):
    minIndex = startIndex
    currIndex = minIndex + 1
    while currIndex < len(L):
        if L[currIndex] < L[minIndex]:
            minIndex = currIndex
        currIndex = currIndex + 1
    return minIndex
```

Sorting – selection sort

- running time – Big O?
- let n be len(L)
- findMinIndex(L,startIndex) - number of basic steps?
  - n-startIndex
- selectionSort(L)
  - calls findMinIndex(L,i) for i = 0..n-1
  - so total steps = (n-0) + (n-1) + (n-2) + … + 1 = ?
  - so, $O(n^2)$

Sorting

- lec30sorts.py code has sorting functions plus
  - timing functions timeSort, timeAllSorts
  - mixup function that takes a list as input and randomly rearranges items (note: contains commented out code that demonstrates *incorrect* random mixup algorithm as well)

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, 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.

```
def testSort(sortFunction, title= '', 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()
        sortFunction(listToSort)
        endTime = time.time()
        runTimes.append(endTime-startTime)
    pylab.figure(1)
    pylab.clf()
    pylab.xlabel('List size')
        pylab.ylabel('Time (s)')
    pylab.title(title)
    pylab.plot(listSizes, runTimes, 'ro-')
```

lec30sorts.py

lec30plot.py

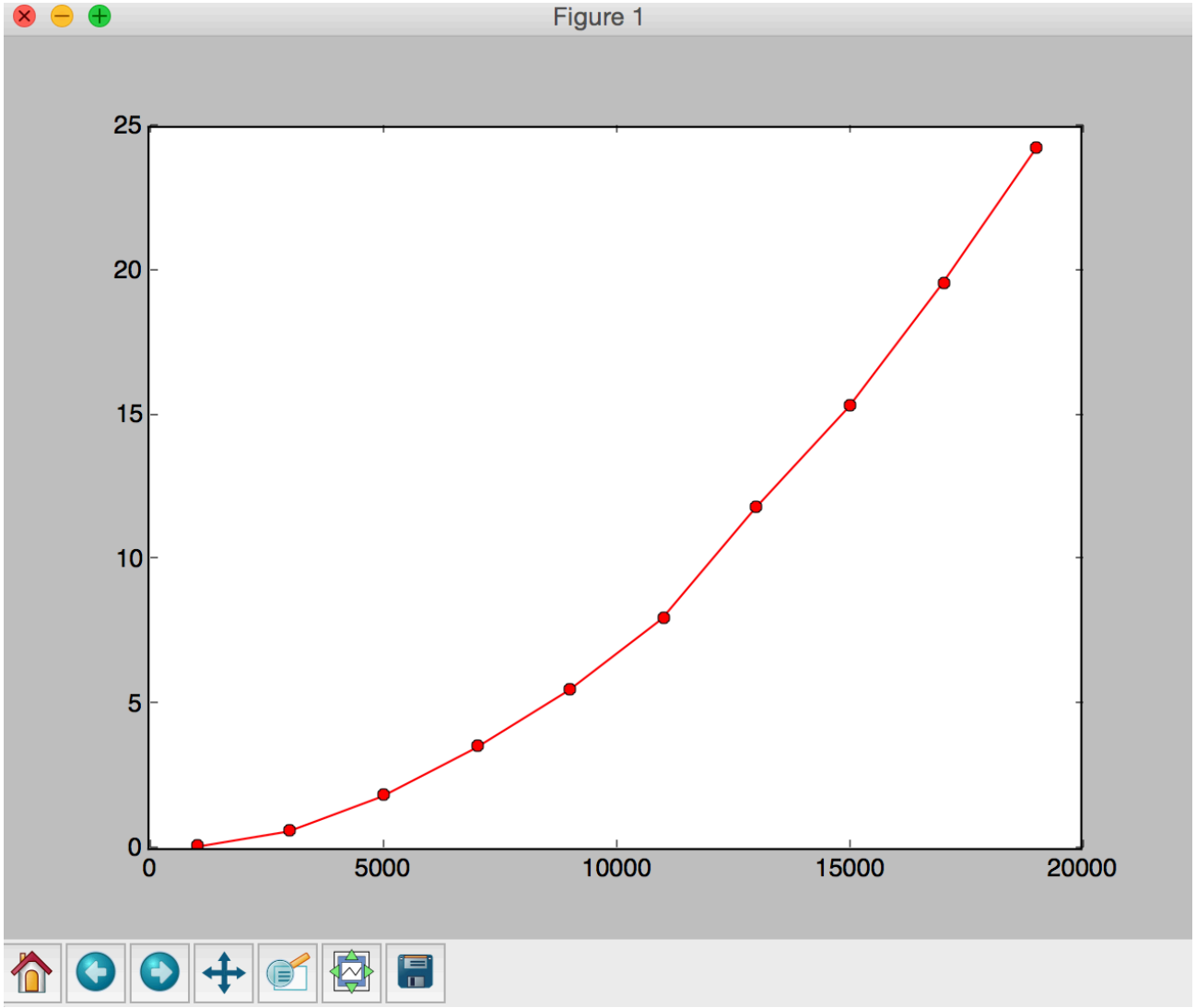


Figure 1
25
20
15
10
5
0
0
5000
10000
15000
20000

Next time

- more efficient sorting:
  - merge sort
  - Quicksort

- 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 …)