CS2110 Lecture 29 Mar. 31, 2021

- **Important schedule change:** quiz 4 changed to April 23
- DS7 due today by 5pm
- HW7 available, due next Wednesday night

Today

- Basic sorting algorithms:
  - Selection sort
  - Insertion sort
- More efficient sorting algorithms

DS 7 and HW 7

- Need to 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.

HW 7 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, nearly 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)

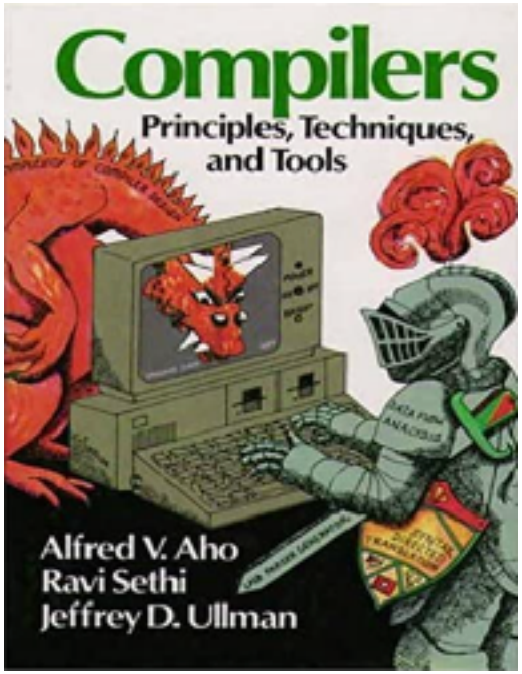
Why sort? Searching a sorted list is very fast, even for very large lists (*log n is your friend*). So if you are going to do a lot of searching, sorting is often excellent prep.

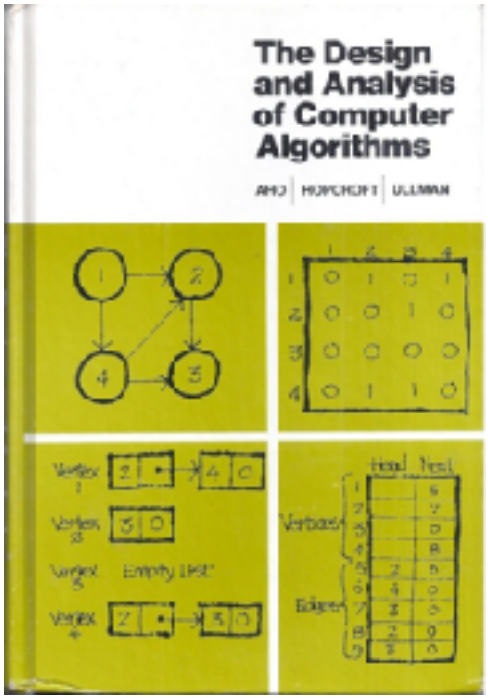
Should you always sort? (Python makes it so easy …)

- 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*

Today's news - this year's Turing Award winners ("the Nobel Prize of computer science")

- https://www.nytimes.com/2021/03/31/technology/turing-award-aho-ullman.html
-





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:

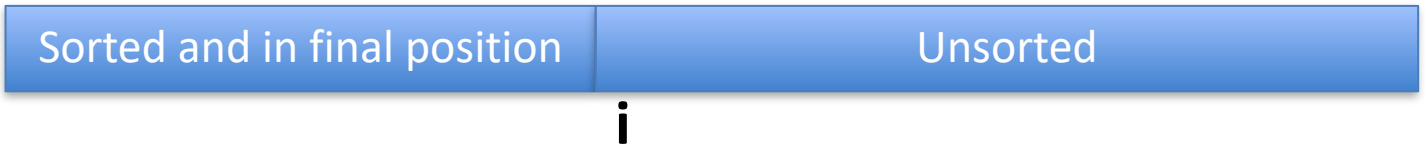
L[0:i] sorted and in final position

L[i:] unsorted

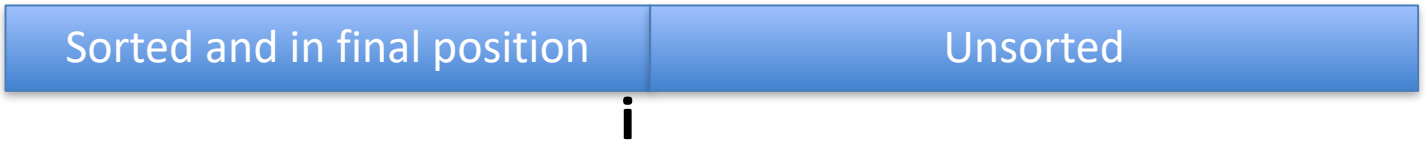
How do we “grow” solution?

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

- lec29sorts.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-')
```

lec29sorts.py

lec29b.py

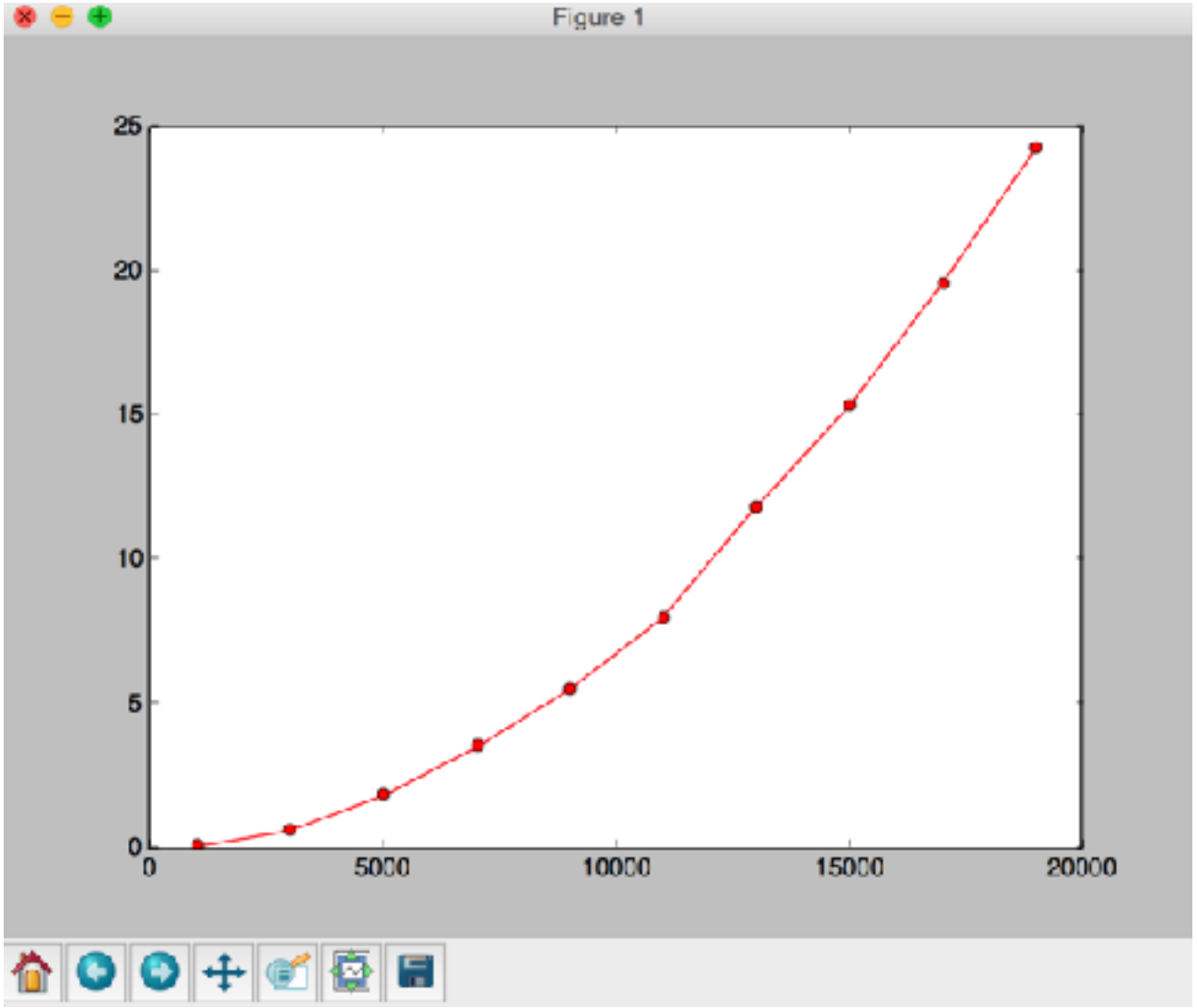


Figure 1
25
20
15
10
5
0
0
5000
10000
15000
20000

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 …)