CS1 Lecture 30 Apr. 3, 2017

- HW 7 available
  - very different than others – you need to produce a written document based on experiments comparing sorting methods
  - If you are not using a Python (like Anaconda) that has pylab installed, get one. You need it for HW7
- Discussion sections tomorrow are important!
  - Will cover basics of making graphs with Pylab as needed for HW7. We won't cover this in lecture!
- Exam 2: Thursday, April 20, 6:30-8:00pm

Last time

- Ch 21 – analysis of algorithms, Big-O notation

Today

- Sorting algorithms

Last time: asymptotic notation

Big-picture thinking led to rules of thumb for describing asymptotic complexity of a program:

- if the running time is the sum of multiple terms, *keep the one with the largest growth rate*, dropping the others
- if the remaining term is a product, *drop any leading constants*

E.g. $132022 + 14\,n^3 + 59\,n \log n + 72\,n^2 + 238\,n + 12\sqrt{n}$

$\rightarrow 14\,n^3 \rightarrow n^3$

There is a special notation for this, commonly called "Big O" notation. We say

- $132022 + 14\,n^3 + 59\,n \log n + 72\,n^2 + 238\,n + 12\sqrt{n}$ is $O(n^3)$

Last time - 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*, .001*x^3*,* $23x^3 + 14x^2$*, and* x^3 *all grow at the same rate in the big picture – all grow like* x^3*. They are all* $O(x^3)$

Important complexity classes

Common big-O cases:

- O(1) denotes constant running time – a fixed number of steps, *independent of input size*. 1, 2, 20000000.
- O(log n): logarithmic running time. E.g. binary search
- O(n): linear time. E.g. linearSearch
- O(n log n): this is the characteristic running time of most good comparison sorting algorithms, including the built-in Python sort.
- O(n^k): polynomial time. k = 2: quadratic, k = 3: cubic, … E.g. some simple sorts (bubble sort, selection sort), or enumerating pairs of items selected from a list
- O(c^n): exponential time. 2^n, 3^n, … E.g. generating all subsets of a set, trying every possible path in a graph

Last time - 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).

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 move it to 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 and 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?
    - $O(n^2)$

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 …)