CS1210 Lecture 28 Oct. 27, 2021

- HW 6 due tomorrow
- Quiz 3 Friday in class
  - Recursion (implement a recursive function like in HW5 and DS6, 6 or 7 points?)
  - Objects and classes (finish implementing a partial implementation of a class, 6 or 7 points?)
  - Big-O (1 basic question, just a few points - 2 or 3)
  - Binary search (1 question, probably completing a partial implementation of binary search, not many points - 4?)

Today

- Continue introduction to analysis of algorithms, searching, and sorting. Appendix B of (non-interactive version of) textbook (or Ch 21 if you have printed version)
  - "Big-O" notation
  - Binary search
  - Details of simple sorting algorithms

Last time discussed “RAM” model used to count steps of program execution.
Considering again the 4 + 3n steps for foo(n)

```
def foo(n):
    i = 0
    result = 0
    while i <= n:
        result = result + i
        i = i + 1
    return answer
```

- I said that we usually ignore the 4. It turns out we are also usually happy to ignore the leading constant on the n. **n** is what's important - ***the number of steps required grows linearly with n***.
- Throwing out those constants doesn't always make sense - at "tuning" time or other times, we might want/need to consider the constants. But in big picture comparisons, it's often helpful and valid to simplify things by ignoring them.

We’ll look at two more examples before formalizing this throwing-away-stuff approach via *Big-O notation*.

From last time - when can we search quickly?

- When the input is sorted. (old examples: dictionary, phone book? Stack of hw/exam papers sorted by name?)
- Algorithm : check middle item, if item is too soon in sorted order, throw out first half. If too late, throw out second half. Repeat.
- This algorithm is called binary search – mentioned briefly earlier in the semester.

lec27.py contains recursive and non-recursive versions. Both should be fairly easy to understand; the recursive version somewhat more natural to write. You need to understand them! Note that in the iterative version you need to be careful when updating startIndex, endIndex. See incorrect versions in bsearchAB.py

Linear search vs. binary search

- We said that in worst case linearSearch of 100000000 items takes 2n+1 or 200000001 steps. Let's just throw out the factor of 2 and extra 1 and call it a hundred million.
- How about binary search??
  - A few basic steps * number of recursive calls. How many recursive calls?
  - In the iterative version (binarySearchIterative in lec26.py), approx. 5 basic steps * number of loop iterations
  - Number of recursive calls/loop iterations? Can put in code to count them to help us get a feel …

    Always less than 28
- It should be clear that here, the 2 multiplier on a 100M doesn't make a difference in telling us which algorithm is superior. And we could do 10 or 20 or even many more basic ops inside the binary search core. The key term for linear search is the factor of n.
- Do you know what function of n characterizes the key part of binary search: how many recursive calls are made, or how many loop iterations occur? I.e. what function relates 27 to 100000000?

  27 ~ log(100000000))

- Programs that run proportional to *log n* steps are generally much faster than programs that require linear number of steps:

$a + b*\log(n) < c + d*n$ for most a, b, c, d that would appear in actual programs.

Another example

Consider the following function. What part of the function dominates running time as n gets large?

```
def f(n):
    ans = 0
    for i in range(1000):
        ans = ans + 1
    print 'number of additions so far', ans
    for i in range(n):
        ans = ans + 1
    print 'number of additions so far', ans
    for i in range(n):
        for j in range(n):
            ans = ans + 1
            ans = ans + 1
    print 'number of additions so far', ans
    return ans
```

lec27.py

For this example, let's ignore basic steps other than additions. What equation characterizes the number of additions?

$$\text{Number of additions} = 1000 + n + 2\,n^2$$

When n is small the constant term, 1000, dominates. What term dominates with larger values of n? (At around what point does the "switch" take place?)

- at n == 100, the first term accounts for about 4.7% of the additions, the second for 0.5%
- at n == 1000, each of the first two accounts for about 0.05%
- at n == 10000, the first two together account for about 0.005%
- at n == 1000000, the first two account for 0.00005%

Clearly, the $2\,n^2$ term dominates when n is large …

Asymptotic notation

- So, for this last example the double loop is the key component for all but small n.
- Does the 2 in 2 n^2 matter much? It depends on your needs. Getting rid of one addition (here easy – just change to ans = ans + 2) would halve the required steps for a particular input n.
  - If n == 1000000 required 5 hours to run, that would be cut to 2.5 hours. Perhaps that'll be sufficient for your needs …
- But you'll also have to consider that each time you double n, you quadruple the number of required additions (whether or not that 2 is there). So, for n == 2000000, the requirement would be 20 hours (2 adds in inner loop) or 10 hours (1 add in inner loop).
  - If quadrupling is a concern, the leading 2 isn't the issue, the power 2 of n^2 is, and you might need to redesign the algorithm to try to achieve a "biggest term" of n log(n) or n instead of n^2

Asymptotic notation – Big-Oh

This kind of thinking leads 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. $132022 + 14\ n^3 + 59\ n \log n + 72\ n^2 + 238\ n + 12\ \sqrt{n}$

$\rightarrow 14\ n^3 \rightarrow \quad n^3$

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

- $132022 + 14\ n^3 + 59\ n \log n + 72\ n^2 + 238\ n + 12\ \sqrt{n}$ is $O(n^3)$
- Or in the prior example, $1000 + n + 2\ n^2$ is $O(n^2)$

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 – some details

Note that Big O is used to give an *upper bound* on a growth rate – a guarantee that the growth rate is no larger than some value

- Technically, while 23 n^3 is $O(n^3)$, 14 n is also $O(n^3)$ because n^3 is certainly an upper bound on the growth of 14 n. 14 n's growth rate is not larger than that of n^3
- Typically, though, we try to make "**tight**" statements. If someone says program foo(n) runs in $O(n^2)$ steps, we generally expect that they mean the grow rate of the number of steps is quadratic. I.e. that n^2 is both a lower and upper bound on the growth rate. Computer science people actually have a special notation for this – $\Theta(n^2)$ – but many people just informally use O most of the time
- It'd still be a true statement if foo(n) always took only one step but it wouldn't be very helpful or informative to tell someone it's $O(n^2)$ in that situation – better to have said it runs in constant time, which is written O(1) in asymptotic notation.

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.

So, for these functions from Monday, what are the Big-O bounds?

foo1: O(n*n)

foo2: O(n*n)

foo3: O(n * log n)

foo4: O(n^3)

```
def foo1(n):
    result = 0
    for i in range(n):
        for j in range(n):
            result = result + i*j
```

```
def foo2(n):
    result = 0
    for i in range(n):
        for j in range(i):
            result = result + i*j
```

```
def foo3(n):
    result = 0
    for i in range(n):
        j = 1
        while j < n:
            temp = j * j + j + 1
            result = result + i*temp
            j = j * 2
```

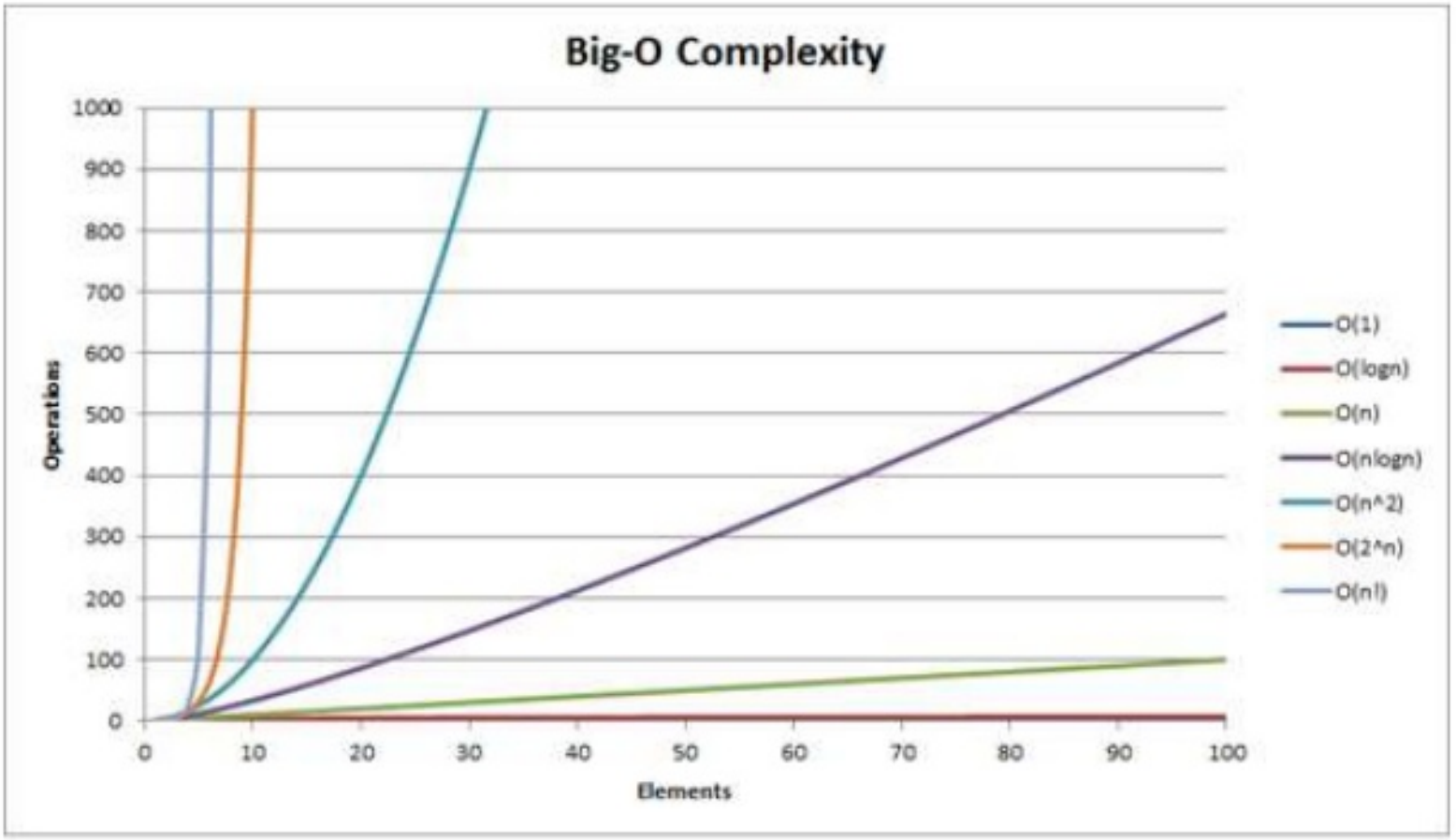
```
def foo4(n):
    result = 0
    for i in range(n):
        for j in range(n*n):
            result = result + i*j
```

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

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 week

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

Next week: 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*

Next week

- Sorting algorithms
  - Basic sorts: selection and insertion sort
  - More efficient sorting methods

- 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 …)