CS1 Lecture 29 Mar. 31, 2017

- Several people are waiting for email responses from me – I will answer soon – sorry!
- HW6 (and earlier)
  - a number of people write emails saying something like "ICON was slow the minute before the deadline – please accept this one minute late …". *You* need to avoid this. Submit a first version well before the deadline, keep updating. There are many situations in the real world where 1 minute late gets you nothing ...
- HW 7 available by Monday morning
  - For HW7 you will need matplotlib/pylab.
  - Anaconda (and some other Python installations, such as Canopy) make using pylab easy.
  - If you are not using a Python that has matplotlib/pylab installed/ available, it's time to get one …
  - Check via:

    >>> import pylab

    If no error, you are all set

    (e.g. pylab.plot([1, 2, 3, 4, 5, 6, 7], [2, 4, 9, 16, 25, 36, 49])
    pylab.show())

Last time

- Ch 21 – intro to analysis of algorithms

Today

- Chapter 21 – more algorithm analysis, Big-O notation

Last time – discussed counting basic steps

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

- 4 + 3n basic steps
- Said that we usually ignore the 4. 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***.
- In big picture comparisons, it's often helpful and valid to simplify things by ignore such constants

Last time – discussed linear search vs. binary search

```
def linearSearch(L, x):
    for item in L:
        if item == x:
            return True
    return False
```

- Can we make a simple equation like we did for foo? Hmm…
- If L has a 100 million elements and we search for 3:
  - if we are lucky - the best case - 3 is first in the list. Function returns immediately after perhaps three steps (start for loop, check if, return)
  - if unlucky - worst case - it's not in L. Steps? 2 * len(L) + 1. If we use "n" to stand for len(l), this is 2n+1
  - on avg, perhaps we find it halfway → 2 * (1/2 * n) + 1 → n+1
- Algorithm designers typically focus on worst case

And, because it basically does the same thing, saw via timing demo last time that Python's 'in' is not a basic step – can be slow!

Last time - binary search - 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.

lec28.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!

Last time - 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 lec28.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.

Asymptotic notation – a formal way of describing "big picture" computation complexity

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
```

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 …

Asymptotic notation

- So, for that 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

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

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

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 $O(n^3)$
- 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

- Technicallly, 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.

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

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. $O(n^2) \rightarrow$ 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).

Calculating complexity

A couple of examples:

- intToStr: ?
  - O(log n)
- matrixMult ?
  - try timeMatrixMult(100) … and 200 and 400
  - O(n^3) for n-by-n matrices

- Slow and fast versions of parts of HW5 Q1

lec29.py, hw5partsSlowFast.py

Next time

- Sorting algorithms

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 …

So, 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. We certainly can't sort fast 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*