

# CS1 Lecture 28

Mar. 29, 2017

- HW 5 scores posted
- HW 6 due tonight
  - Q1. Questions about Box?
  - Q2. NimGame – make sure only legal moves are allowed: 1-3 balls but no more than max remaining
- HW 7 available Friday

## Last time

- Ch 18: class attributes and inheritance

## Today

- Chapter 21 – analysis of algorithms

# Ch 21 Introduction to Analysis of Algorithms (or “Computational Complexity”)

- The first step of programming (before implementation/ typing in the code) should be design. At this stage, you need to worry both about designing a correct solution *and* a solution that will be efficient enough (in a big picture sense) to meet your needs
- Then, after first correctly implementing your program there can be, if necessary, a *tuning* process to increase program speed as much as possible. But improvements at this stage are usually much less substantial than those created by selecting the right solution approach at design time. Sometimes, you will need to redesign and re-implement, using a better algorithm, if tuning doesn't yield enough improvement.
- Note that a critical first consideration is *correctness*. Worrying about details of *efficiency* should generally come second.
- Over the next few lectures we'll take a brief look at formal tools and techniques for thinking about “big picture” program running time and algorithmic efficiency.

WANT TO HAVE FORMAL WAY TO TALK ABOUT RUNNING TIME/EFFICIENCY DIFFERENCES BETWEEN ALGORITHMS ...

HOW SIMILAR/DIFFERENT ARE THE FUNCTIONS BELOW IN RUNING TIME/EFFICIENCY?

```
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):  
        for j in range(n):  
            if (j % 2 == 0):  
                result = result + i*j
```

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

# Thinking about Computational Complexity

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

How do we assess how long f takes to run?

- Time it? That can be very useful. But again, we often want to analyze before implementation and get a big picture sense of how the program will perform
- Instead of trying to measure in actual time units - e.g. milliseconds - which would not hold up well on computers of different speed, etc - we will characterize program efficiency in terms of the **number of basic steps executed by program**
- You don't need to know the details, but such analysis is based on a hypothetical computer called a Random Access Machine (RAM). Under the RAM model: basic step is one that takes a fixed amount of time - binding a value to a variable, making a simple comparison, doing arithmetic ops like +-\* /

# Thinking about Computational Complexity

In the RAM model, a program like:

```
def bar()  
    x = 3  
    y = 4  
    result = x*y  
    return result
```

requires four steps to run (always).

In many programs, however, the number of steps executed is affected by the input – the size of an input list, the value of a given number,  $n$ , etc.

Consider foo again ...

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

How do we deal with this? Count the steps ...

- 4 always – i and result initialization, condition test, return
  - Each iteration of loop: one test, two inside ops ....
- Therefore, 3 \* number of iterations. Number of iterations is n.

So,  $4 + 3n$ . Right? Perhaps ...

But what if the first line was, say:  $i = 23 * n + 14$  or

the first line in the loop was something like:  $result = 3 * result + (i + i + i)$ ?

Do we know exactly how many basic steps?  $4 + 3n$ ?  $4 + 4n$ ?  $5 + 5n$ ?  $4 + 6n$ ???

Not usually – depends on particular processor and compiler.

**BUT DON'T WORRY!** We're interested in a BIG PICTURE sense of the running time of this program, and we're going to get used to throwing away "small" terms like the 3 or 4 or whatever it is. The key term is  $c * n$ , for some fixed number c. For any but very small n, the number of steps that foo requires is approximately  $c * n$ . As n grows, the number of steps executed grow **linearly**.

And when we know something grows linearly, we know things like: if I double the input size the program will take ??? as long? Twice as long

## Another example, a little more complex:

```
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?  $2 * \text{len}(L) + 1$ . If we use “n” to stand for  $\text{len}(L)$ , this is  $2n+1$
  - on avg, perhaps we find it halfway  $\rightarrow 2 * (1/2 * n) + 1 \rightarrow n+1$

*Note: what does this say about Python's “in”? Try it! Is it a basic step?*



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

- 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?  $2 \text{ len}(l) + 1$ . If we use “n” to stand for  $\text{len}(l)$  n, this is  $2n+1$
  - on avg, perhaps we find it halfway  $\rightarrow 2 (1/2 * n) + 1 \rightarrow n+1$

So, this has added some complexity to our analysis approach. Different characterizations for different situations. Best, worst, and average cases.

**Algorithm designers typically focus on worst case** since that gives guaranteed upper bound on running time/steps (“I can promise you it’ll never be worse than *this*”)

# Let's look again at the $4 + 3n$ steps for $\text{foo}(n)$

```
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*.

# 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!**

# 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?

**$27 \sim \log(100000000)$**

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

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

Next: 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?

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

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  $2n^2$  term dominates ...

- Next time: the formal notation – Big-O notation - for characterizing running time in terms of inputs.