CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki

Administration

- Final is Aug 16 7pm-10pm in SF3201
- Assignment 2 update.
  - Silence for last 24 hours that the assignment is due.
  - Office hours W 2-4 next week instead of F 2-4 next week.
- Help Centre is in BA2270 2-4 M-R.

Sorting Overview

- We covered three types of sort: Bubble, Insertion, and Selection.
- Selection sort minimises swaps.
- Insertion sort is optimal for small data.
- Bubble sort is optimal for nearly sorted data.
- Each sort can be viewed as running a helper function inside of a for loop.
  - The helper function contains the 'core' of the sort.

Sorting in practice.

- In practice bubble, selection, and insertion sort are all sort of slow.
- There are better sorting methods out there.
  - The most commonly used ones are merge, heap and quick sort).
  - These all rely on recursion.
- Python uses an adaptive form of merge sort.
- Bubble and insertion sort have specific instances in which they are useful and are used.

Thinking about Loops

- Commonly one thinks of loops as single big elements.
  - One imagines the whole loop doing something.
  - The program is in one state, executions the loop in another.
- Sometimes one needs a more refined way of looking at loops.
  - Commonly to find errors.
  - Also used

Refined Loop Descrition

- A useful tool for analysing loops is a loop invariant.
- A loop invariant is a statement that is true every time to loop begins.
  - So it depends on the loop index.
- They have both informative and imperative functions.

Loop Invariant Example

```
for j = 0 to n-i-1
        if my_lst[j]>my_lst[j+1]
                swap my_lst[j] and my_lst[j+1]
```

- Here we see that the jth element is always the biggest that we've seen. So a loop invariant would be:
  - my_lst[j] is the largest element in my_lst[0:j]
    - This tells us a truth at the beginning of any iteration.
    - It also tells us what we need to do in any iteration.

Pseudocode and Loop Invariants

- Loop invariants are really useful in pseudocode, since they point towards the overall design of the program.
- Also can be useful in finding +/- 1 errors.
  - Explicitly stating things in terms of the index makes it easier to catch mistakes.

Good programs

- What do we want from a good program?
  - Correctness
  - Legibility
  - Speed

How do we measure speed?

- We can measure directly.
  - Get time from the computer, perform our code, get time again and measure the difference.
- We use the time module to get time from the computer.
  - use time.time() to get the time in seconds since Jan 1st, 1970.
- Speed testing can be a part of testing.

Problems with direct speed measurements

- Dependent on system architecture.
- Dependent on other programs that are running.
- Dependent on input!
  - It is hard to tell if you will have bad inputs from a suit of tests.
  - If your code has varying speed performance what can one say about it.

Architectures

- Code can often run much faster on one machine than another.
- Certain compilers and languages are betters suited for certain machines than others.
  - The same code in different languages can run at different speeds.
- Architectures change quickly, how can we tell that we're testing the code and not the architecture?

Moore's Law

- Says that processing power doubles every 18 months.
  - Although that number is up for debate.
  - Still the point is that computer speed increases fast.
- This means that larger inputs are getting tractable all the time.
  - Speed testing doesn't cover this really well unless you constantly update them.
  - Large scale speed tests are costly.

Alternative to Speed Testing

- Should be architecture independent.
  - Just based on code, or even the idea behind the code.
  - One program might have a worse idea but be faster due to the language it is written in.
- Should be scalable.
  - Don't need to update tests.

Break, the first

Computational Complexity

- Developed by Computer Scientists to make general claims about the speed of programs that are scalable.
- The idea:
  - take an algorithm designed for arbitrary input.
  - Assume it takes n input.
  - Give a function that describes how long it takes.
  - Compare the function size.

Inputs

- What if your algorithm is really fast on some inputs but slow on others?
- Consider the worst case.
  - The idea behind this is that you don't know what inputs are going to come in practice.
  - But you want to make a guaranteed claim

How do we measure speed?

- Look to how processors are built.
  - This tells us arithmetic operations all take the same amount of time.
    - Note that (1+3) + (5 * 6) counts as 3 arithmetic expressions.
  - So do variable lookups, and boolean expressions.
- So we set this to be our base unit and count 'steps'.

Counting steps.

- We could go through a program and explicitly count every step.
- This is difficult.
- Moore's Law suggest that there's not a huge difference between something that takes x steps and something that takes x + 4 steps.
  - In 18 months, the whole thing is divided by two anyways.

Moore's Law and Counting.

- This suggests that taking something from x + 4 steps to x steps is not a good use of time.
- Recall that we have input of size n.
  - So we just tend to put all the constant terms into one big constant and leave it at that.

Counting Steps:

- What about loops?
  - Here we count the number of lines in the loop block, times the number of times the loop is executed.
  - Keep in mind that if we have nested loops, then to get the total number of lines of code we execute, we need to multiply the number of times the inner loop runs time the number of times the outer loop runs.
  - For while loops we need to consider the worst-case scenario.

Terms dependent on n

- So we know that each individual step takes a constant amount of time.
- To get something dependent on n, we need to do something for each element of the input.
- And example of this would be a for loop over an input list.
  - If each iteration of the loop takes a constant amount of time, the complexity is n, since we do n operation.

Multiple terms

- What if we calculate our function and it has 15 + 2n + n*n steps?
- We saw that we treat all constants the same.
- For similar reasons, we can ignore the 2n and just focus on the n*n.
  - In particular we don't care about the constant in front of the n.
- So we say that the code grows quadratically.

Some general guidelines:

- Always take the worst case if your code varies what it does.
- Constants aren't important.
  - We'll see why this is formally in 165.
  - But roughly, this has to do with the way functions grow at large numbers.
    - Which we justify using Moore's law.
- A for loop that depends on list or dictionary length almost always adds a power of n.
  - So a lot of basic complexity analysis is just counting for loops.

Complexity Analysis.

- Generally done at the conceptual stage.
- But can be applied to explicit code.
- Generally constants are ignored, and we only care about the biggest term in the function.
- Smaller functions are considered better.

Speed Testing Vs. Complexity

- Speed Testing is very good for optimising solutions.
- Complexity analysis is better for deciding which solution to use.
- Even testing at a bunch of scales isn't guaranteed to find the best solution.
- A nice talk about how sometimes using complexity can lead to improvements in speed.
- http://epresence.kmdi.utoronto.ca/1/watch/889.aspx