CSC 108H: Introduction to Computer Programming

Summer 2011

Marek Janicki

Administration

- Assignment updates.
- Brief exam information.

Judging Programs.

- We can judge programs by correctness, and whether they meet specifications.
  - We use testing and proofs to do this.
- We can judge them by legibility.
  - We have style guides to help us with this.
- Finally we can judge them by how fast they run.
  - ?

Judging speed.

- Obviously, we want our programs to run as fast as possible.
- One way is just to time our programs.
- But there are a few things that make this not as useful as it seems.
  - Differing architectures.
  - Moore's Law.
  - Differing input sizes.

Architectures.

- Code can often run much faster on one machine than another.
- It can even run faster if it's written in certain programming languages vs. others on the same machine.
- So if we time our programs, how do we know we're testing the quality of the algorithm, and not just how well our program matches 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.
- So if we compare times on different machines we may just be testing the machines and not the algorithms.

Input Sizes.

- Moore's law means that the maximum 'reasonable' input size for our algorithm is constantly increasing.
- But what if our program does well at small inputs, but once it hits some threshold it becomes much slower than other algorithms.
  - For example, insertion sort is optimal for small (<7 or so) inputs.
  - How can we be certain that our algorithm scales well?

Recap.

- We want a way of judging programs that is:
  - Architecture independent.
  - Captures how the program scales.
  - Is independent of the speed of the computer we're running our tests on.

Computational Complexity

- What computational complexity is, very roughly, is reading code and making a rough estimate of how long it takes per input size.
- Each line of code that is executed is counted as one step.
  - So an arithmetic statement is considered to count as much as a boolean expression which is as much as a print statement.
  - This is an oversimplification, but it is reasonable due to Moore's law.

Counting Steps:

- What about code that does a varying number of steps?
- Like and if/elif block. Maybe one block of code has a single statement, and the other has 20 statements.
    - In this case we will use Worst-Case analysis, and always assume that the longest block of code is executed.

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.

Counting Steps:

- So if we've counted all the steps we get a rough mathematical formula that tells us how many steps the code takes to run.
- Because of Moore's law, we tend to ignore constants, and just focus on the biggest term that involves the input size.
- If we assume that the input size is n, then we care only about the biggest term with n in it (after we've removed the constants).

Some general guidelines:

- Always take the worst case if your code depends on boolean expressions.
- 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.

Breaks,

Base 10 notation.

- Regular number notation is base 10.
- That is, we have 10 digits (0-9), and each digit in a number tells us how many of a power of 10 we have.
- So we can write:
  - 59 = 5 * 10**1 + 9 * 10**0
  - 237896 = 2 * 10**5 + 3 * 10**4 + 7 * 10**3 +
    8 * 10**2 + 9 * 10**1 + 6 * 10**0

What if we didn't have 10 digits?

- What if we only had 5 digits (0-4).
- Well we could still write 21 as 2 * 10**1 + 1 * 10**0.
- But if we want to write 27 in base 4? We can't do anything in the form 2 * 10**1 + x * 20 ** 0, because that will only get us number from 20 to 24. And if we change the first digit to 3 then the lowest we can go is 20

What if we didn't have 10 digits?

- So clearly we can't use 10 as our powers.
  - But we say that when we had 10 digits, we used 10 as our base.
  - So if we only have 5, then we should use 5 as our base.
- So now when we write 21_5 in a base 5 system that means 2 * 5 ** 1 + 1 * 5 or 11 in our regular base 10 system.

Binary notation.

- Binary is just a base-2 system.
  - So a number system when we only have two characters (0-1).
  - Used in computers due to hardware reasons (since there are only two options, can represent them as high and low voltages).
- So 1010110_2 is 1 * 2**6 + 0 * 2**5 + 1 * 2**4 + 0 * 2**3 + 1 * 2**2 + 1 * 2**1 + 0 * 2**0.

Some useful tricks.

- If we want to get only the lowest k bits of a base j system from a number n, we do:
  - n % j**k
- If we want to get only bits that are higher than the lowest k bits in a base j system from a number n, we do:
  - n / j**k
- We can use these to clear out low or high order bits, or to figure out what a specific bit is.