CS 6012: Data Structures and Algorithms

Prof Ben Jones

Course Content

- In this course we'll implement **and analyze** common data structures and algorithms
- What you should get out of this course:
  - Coding practice by implementing things in Java
  - Familiarity with common DS's and algorithms that you'll use frequently
  - The ability to come up with, evaluate, and compare various solutions to problems

Different from 6010/6011

- We will expect you to “know how to code”
  - Sort of. You're still new, but we'll view coding as a necessary step, not the focus of assignments
- Usually we'll have a small, general set of needs and will evaluate many possible solutions:
  - Example: I have a group of things that I need to add to + find/delete the “biggest” thing
- Most of our discussions will be at a high level, maybe pseudocode. It shouldn't be too hard to turn that into Java
- We'll spend about as much time analyzing solutions as coming up with them

Example: Sorting

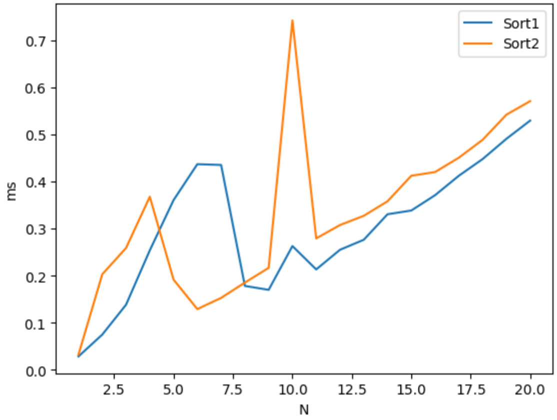
- Given an unsorted list of items, put the items in ascending order
  - Detail that will be important when coding: what kind of objects? What do you mean in order?
- We've already seen a couple of algorithms for this
- How do we know which is best?

Scientist type 1: Experimentalist

- Set up an experiment
- Implement the solutions
- Run them on some “representative inputs”
- Time them
- Compare the results
- This MIGHT be the best thing to do!

Example:

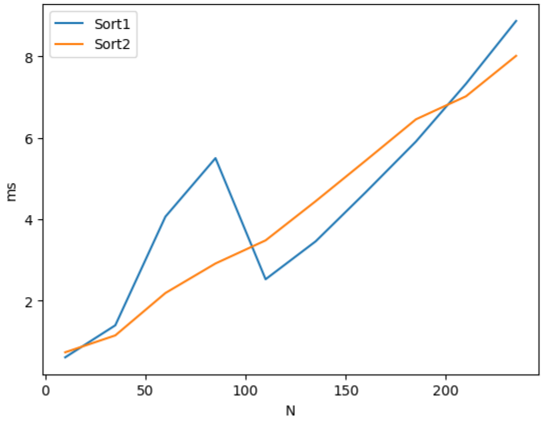
Experiment comparing 2 sorting algorithms:



Measurements

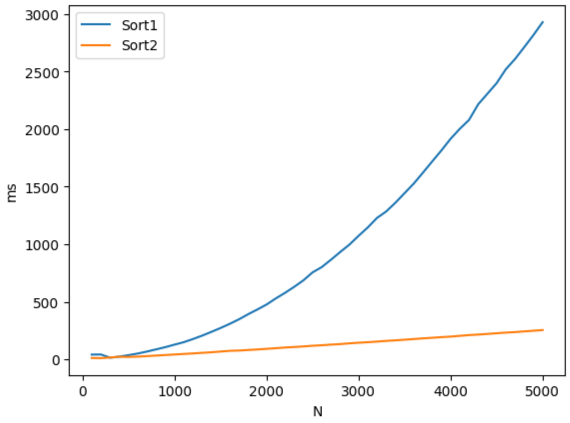
Example?

Same algorithms:



Measurements

Example?



Measurements

Experiments

- Hard to come up with good experiments!
- Behavior may change DRAMATICALLY based on input size!
- There may be a LOT of measurement noise

Scientist type 2: Theoretician

- Predict what will happen before running an experiment
- How do we make predictions about algorithms?
- We saw that real world measurments are noisy/unintuitive… What assumptions and tradeoffs are we making?

Theoretical approach to sort 1:

```
static void sort1(int[] arr){
    var i = 1;
    while(i < arr.length){
        var j = i;
        while( j > 0 && arr[j-1] > arr[j]){
            swap(arr, j, j -1);
            j--;
        }
        i++;
    }
}
```

WHY NOT BOTH?

?

Both

- If our experiments are similar to our “production workloads” then they're the definitive way to compare methods
- They will capture the insane complexity of modern systems
- Theoretical analysis lets us compare algorithms from pseudocode
- They give us a good understanding of how our algorithms scale with input size
- In this class we'll usually perform a theoritical analysis, then implement algorithms and see if experiments confirm those predictions

Writing Code

- We'll spend a lot of time talking about algorithms + data structures in terms of pseudocode
- You'll have to implement lots of them in Java
- The algorithms can be pretty tricky so you'll need good habits in order to avoid headaches
- My biggest suggestion: test early, test often
- We'll spend a lot of time writing test cases for our algorithms. Write and run these early and you'll catch bugs when they're easy to fix
- Try to write the smallest amount of code that you can test, then test it

Writing Tests

- A big part of your assignments will be writing test cases for your programs
- This is an important skill that you'll be using if you work for a healthy company. If you work for a company that doesn't have thorough, automated testing, look for another job
- Often we'll be testing a single method at a time. To write a test, we need to come up with an input/output pair where `output = method(input)`
- You should exercise different parts of your code (different choices in if/else branches, different numbers of times through loops, etc) in each test
- Having 10 tests that exercise the same code path isn't really helpful

Tests to include

- “Normal” but small inputs. For example, if we're testing a sorting method, [5, 3, 10, 4] is a good input
- Edge cases, which might include:
  - size 0 or 1 arrays if applicable
  - inputs where all values are the same
  - inputs where all values are adjacent (1, 2, 3 vs 4, 10, 20)
  - forward or reverse sorted inputs
  - inputs at/around thresholds. If you have `if size > 10: ...` have tests where size is 10 and 11
- Look at your code. Which parts are tricky? Try to include inputs that will force those tricky parts to run

What's next:

- Lab 1: Unit testing
- Assignment 1: Working with 2D arrays
- Tomorrow: Java Generics