10 Introduction

About INFO6205

- Title: *Program Structure and Algorithms*
- Better Title? *Data Structures, Algorithms and Invariants*
- Why?
  - Because algorithms and data structures are like Yin and Yang
  - They are dual concepts
  - They are like a marriage

Algorithm

- Al-Khwarizmi, 9th Century Persian* mathematician
- Arithmos (αριθμός), Greek word for number
- Recipe or blueprint?

* Actually, modern-day Uzbekistan

Wiki-definition

An **algorithm** is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.

… Data Structures

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "

— Linus Torvalds (creator of Linux)

" Algorithms + Data Structures = Programs. " — Niklaus Wirth

Languages?

- Ideally, algorithms and data structures should be taught in a language-agnostic context.

So, which language will we use?

- Java. Why?
    - Languages are *orthogonal* to algorithms and data structures.
    - We could use *any* of the mainstream languages and many more besides. Donald Knuth, who wrote the first major work on computer algorithms (four volumes, may expand to seven), actually invented his own language (MIX) just so that the language issues wouldn't get in the way.
    - But, the textbook is written in Java, and Java is the world's #1 programming language so it makes sense to use it here.
    - *However*, I am a big fan of functional languages so, if I think it is appropriate, I will code in Java8 (occasionally, I might slip in a little Scala). Java8 isn't a functional language, but it has many of the features that you expect in such a language.

Computer Architecture, Languages, etc.

- Procedural language
  - Specifies *how* the computer system is to perform its task.
  - examples: Java, C++
- Declarative language
  - Specifies *what* you want to accomplish and the computer system figures out how to do it.
  - examples: SQL, Prolog

Von Neumann, Turing, etc.

- These architectures are based on the notion of memory locations, registers* and load/store instructions (very low-level stuff);
- They gave rise to the idea of *variables* and *arrays* in programming languages (these are the logical counterparts to physical memory addresses/registers).
    - Actually, you don't always need all those variables: applications written in functional languages like Scala hardly use mutable variables (if at all);
    - However, it turns out that in the development of algorithms and data structures, you can often increase efficiency by judicious use of variables and arrays.

* *Registers are special memory locations supporting load/store and part of the CPU*

Data structures & Algorithms

- So, practically speaking, what are data structures and algorithms?
  - A data structure is an orderly collection of memory locations for the purpose of representing a state (including the end state—the result) of an algorithm.
  - An algorithm is a sequence of steps each of which manipulates a data structure to achieve some new state.
  - Note that this is a mutually recursive definition.

Why are DS&A important?

- Languages come and go; there are flavors of language which are best-suited to a particular type of application. Some languages are “better” than others: easier to write; easier to read; produce more efficient code; compile/interpret faster; higher/lower “level”; more or less verbose; etc. etc.
- But data structures and algorithms are **fundamental** to solving problems—for this reason, they haven’t changed very much in the last forty or fifty years.
- There are two precious commodities in any computer system: cycles (i.e. time) and memory (i.e. space). Good data structures and algorithms aim to minimize the use of these commodities.

What about invariants?

- An invariant is something that never changes (a constant). Sure, but what relevance does that have to data structures and algorithms?

- Elements of data in a data structure have relationships to other elements: these relationships are described by invariants—properties of the relationship that do not change.

Invariant in insertion sort

- In insertion sort, the data structure is divided into two partitions:
  - On the left of the current element, all elements are in order;
  - On the right of the current element, all elements are randomly arranged;
  - When determining where to insert the current element, we can *make use* of this invariant to speed things up.
  - When the algorithm starts, there are no elements in the left partition; when the algorithm finishes, there are no elements in the right partition.

Historical algorithms

- Many ancient algorithms:
    - Sieve of Eratosthenes (200BC) for finding prime numbers;
    - Euclid's algorithm (300BC) for greatest common divisor (see Graphical animation of Euclid's Algorithm):

```
function gcd(a, b)
    if b = 0
       return a;
    else
       return gcd(b, a mod b);
```

What about Euclid’s Data Structure?

- So, are *a* and *b* the data structure?
- Yes, part of it.
- What happens when *b* != 0? We call the method recursively. What happens to the original *a* and *b* when they are apparently replaced by *b* and *a%b*, respectively?
- The original *a* and *b* are placed on the [Java system] stack.
- So, the system stack, which you get for free, is also part of Euclid’s data structure.

More on Euclid's Data Structure…

- We put *a* and *b* on the stack in case they are needed when we return from the recursive call.
- But when we return from the recursive call, we immedi-ately return to our caller. We no longer need *a* and *b*.
- We say that the recursive call is in *tail position.*
- Since we don't need to keep *a* and *b* around, we could just use variables and no stack at all.

```
public int gcd(int a, int b)
    while (b != 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
```

Why are DS&A important? cntd.

- Their impact is broad and far-reaching:
  - Internet. Web search, packet routing, distributed file sharing, …
  - Biology. Human genome project, protein folding, ...
  - Computers. Circuit layout, file system, compilers, databases, …
  - Computer graphics. Movies, video games, virtual reality, ...
  - Security. Cell phones, e-commerce, voting machines, ...
  - Multimedia. MP3, JPEG, DivX, HDTV, face recognition, ...
  - Social networks. Recommendations, news feeds, advertisements, graph databases, …
  - Physics. N-body simulation, particle collision simulation, ...
  - Big Data, Meteorology and Engineering: Finite-element analysis, map-reduce

Why are A&DS important? (Continued)

- I used to have a colleague who was fond of saying: “There’s nothing like hardware to improve software.”
  - This was during the 80s when hardware was undergoing rapid improvements in both performance and, especially, price.
  - Unfortunately, it’s a short-sighted view because, while the cost of cores (CPUs) and memory have drastically reduced over recent decades, the clock speeds of processors have not, at least, not as much.
- And, even if it clock speeds were still improving, improvement will never be much better than linear and, in the next slides, we’ll see that that is no match for real life!

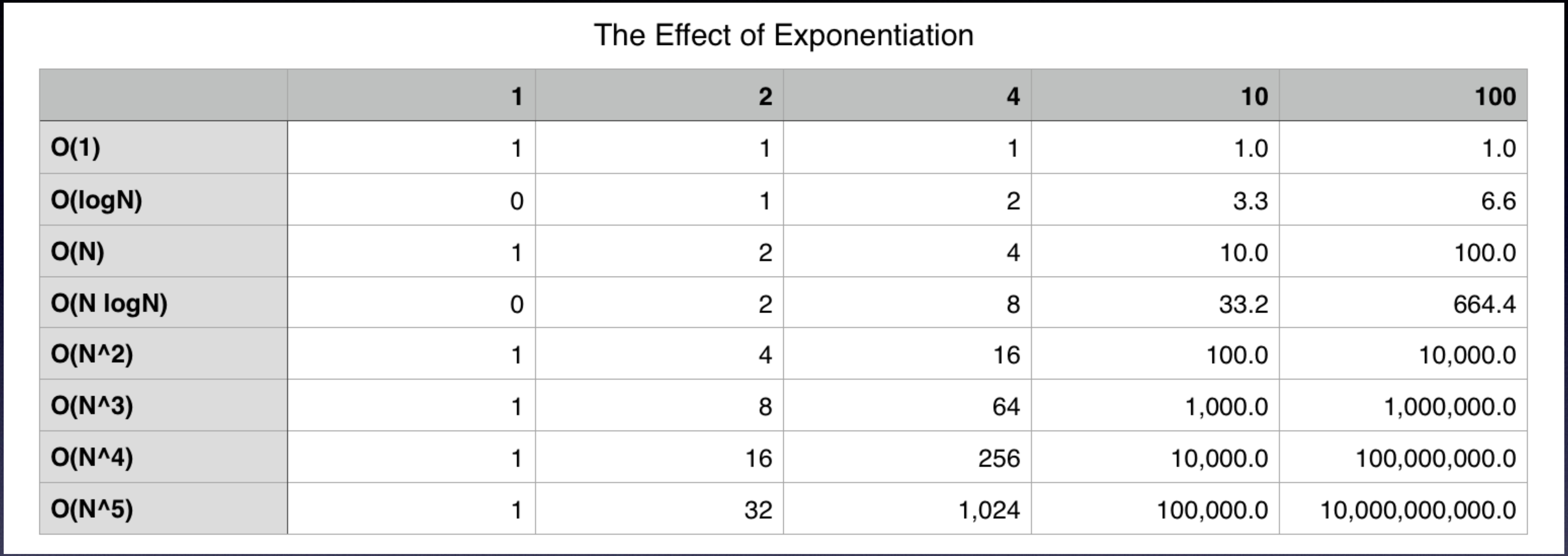
The Exponential Curse

- A silly little (but unsophisticated) database story:
  - Imagine that you create a database of your favorite songs (it doesn't really matter how you do this). You have 1000 songs registered and when you search for one it always takes about 1 second. The space you're using is about 10 Mbytes which is fine: you have plenty of space available.
  - Now, you get a girlfriend/boyfriend and they want to add their 1000 songs to your database. That's fine. But now, when you search for a song, it doesn't take 2 seconds like you'd expect. It takes 4 seconds. Hmmm! You also notice that you're using about 40 Mbytes of space now.
  - Your entire class of 100 students all want to store their songs and use your database. But to search for a song, it now takes, 10,000 seconds (2.75 hours!!!) and you've used up all your memory because it takes 100 GBytes to store everything. Aargh!

$O(N^2)$

- What we just experienced (in this highly fictitious story) was an "N^2" problem, both in time and space. We write this as **O**(N^2), pronounced "Order—N-squared." or, more correctly, "Big-O —N-squared."
- Obviously, that's completely unacceptable! Now let's look at the power of exponentiation.

Exponentiation



The Effect of Exponentiation

	1	2	4	10	100
O(1)	1	1	1	1.0	1.0
O(logN)	0	1	2	3.3	6.6
O(N)	1	2	4	10.0	100.0
O(N logN)	0	2	8	33.2	664.4
O(N^2)	1	4	16	100.0	10,000.0
O(N^3)	1	8	64	1,000.0	1,000,000.0
O(N^4)	1	16	256	10,000.0	100,000,000.0
O(N^5)	1	32	1,024	100,000.0	10,000,000,000.0

- But who would ever create an algorithm that was O(N^5)?
- I did!
  - It was a *very* long time ago;
  - I was part of a team and possibly not the most guilty;
  - BTW, 10,000,000,000 seconds is 320 years!!

Why are DS&A important? cntd.

- Execution:
    - As we just saw, the efficiency of an algorithm can make a huge difference to running time (or space)—indeed it can easily make the difference between running the application and not;
- They are ***interesting***, ***elegant***, and very ***worthy*** of our study;
- And a more prosaic reason:
    - many of the companies for which you will apply to work will ask you to implement algorithms either on the white board or using something like ***HackerRank.***

Non-deterministic algorithms

- Earlier, I left out an important sentence from the Wikipedia article. In fact, many algorithms introduce some randomness into their execution:
  - Either due to timing (for concurrent algorithms);
  - Or due to the use of random values (either from measuring devices or, deliberately, employing PRNGs);
- Examples of the latter include:
  - Genetic Algorithms—a favorite topic of mine;
  - Neural Nets;
  - Monte Carlo methods (e.g. MCST);
  - Particle Swarm Optimization.

What will we be covering?

topic	Data Structures and Algorithms
intro & data abstraction	Reduction, entropy, induction, APIs, ADTs, stack, queue, bag, union-find, O(N)
sorting	quicksort, merge sort, heap sort (priority queues)
searching	symbol tables (maps), binary search trees, hash tables
graphs	undirected and directed graphs, minimum spanning trees, shortest path
advanced topics*	Interviews, compression, radix sorts, finite state machines, non-deterministic algorithms, P & NP

* if there's time!

Basic Programming Model

- See Chapter 1.1 of Textbook
  - Ensure that you understand about primitives in Java (their range of possible values) and their corresponding (boxed) Objects
  - I want you to start thinking about the difference between mutable and immutable variables and data structures:
    - Always initialize variables when you declare them (to do otherwise is a serious code smell);
    - If you don’t plan on allowing them to mutate, mark them as *final*.
  - Arrays are, by definition, mutable and they will be used a lot in this class.

Collections

- It’s not very interesting, or useful, to deal with one thing at a time. In most code (not all, obviously), you will be dealing with *collections* of things. One of the most important use cases for collections is to serve as longer-term data structures. [This is as opposed to shorter-term data structures which are there to support algorithms, such as sorting].

- Collections are built from abstract data types and the specific requirements for creation and access are what determine the particular implementation.

Lists, arrays, symbol tables

- What are the *essential* differences between the following:
    - A variable
    - an array (how long does it take to index an array?)
    - a list (how long does it take to search a list?)
    - a symbol table (aka hash table or map)

Degrees of Freedom

- As a data structure, a variable has one degree of freedom:
    - Its value
- Once it’s initialized, it also has a constraint:
    - Its value
- In Engineering, constraints=degreesOfFreedom implies:
    - Stable system

Lists

- A list element has two fields: *value* and *next*:
    - 2 degrees of freedom;
    - Constraint: the last element has a *null* next pointer:
    - Total *dof* for list of length *N* is *2N-1*.

Lists, arrays, etc.

	Size	Access	Construction
Array	Fixed	Random, by index	Random, by index
List	Variable	Sequential, starting from the head	Sequential, starting from the head
Hash Table	Growable	By key	By key and its hash

Assignment 0

- What was this all about?
  - This was designed to get you started with some Java, especially the use of functions.
  - It is based on an algorithm that doesn't really have much of a data structure.
  - Is it deterministic, though?
- Topic/method:
  - This is based on the Newton-Raphson method of approximation. This kind of algorithm is very different from most of what you'll be learning in this class. But it doesn't hurt to get a feel for Numerical computing.
- Get a feel for the algorithm:
  - Watch the simulation from Wikipedia (https://en.wikipedia.org/wiki/Newton%27s_method#/media/File:NewtonIteration_Ani.gif)

An algorithm without a data structure

```
public class NewtonApproximation {
    public static void main(String[] args) {
        // Newton's Approximation to solve cos(x) = x
        double x = 1.0;
        int tries = 200;
        for (; tries > 0; tries--) {
            final double y = Math.cos(x)-x;
            if (Math.abs(y)<1E-7) {
                System.out.println("The solution to cos(x)=x is: "+x);
                System.exit(0);
            }
            x = x + y/(Math.sin(x)+1);
        }
    }
}
```

- Newton-Raphson method (https://en.wikipedia.org/wiki/Newton%27s_method) to approximate the solution to cos(x) = x
- Originally written in FORTRAN in 1968 by manually punching holes (“chads”) in a partially perforated “Hollerith” card. Turnaround time: 1 week!
- In my Scala/BigData class, I use this as an exemplar of “imperative-style” programming (and then I show how to improve it with functional programming). But as a Java algorithm, it works fine.

```java
package edu.neu.coe.info6205.functions;
import java.util.function.DoubleFunction;
public class Newton {
   public Newton(final String equation, final DoubleFunction<Double> f, final DoubleFunction<Double>
dfbydx) {
        this.equation = equation;
        this.f = f;
        this.dfbydx = dfbydx;
    }
   public Either<String, Double> solve(final double x0, final int maxTries, final double tolerance) {
        double x = x0;
        int tries = maxTries;
        for (; tries > 0; tries--)
            try {
                final double y = f.apply(x);
                if (Math.abs(y) < tolerance) return Either.right(x);
                x = x - y / dfbydx.apply(x);
            } catch (Exception e) {
                return Either.left("Exception thrown solving " + equation + "=0, given x0=" + x0 + ",
maxTries=" + maxTries + ", and tolerance=" + tolerance + " because " + e.getLocalizedMessage());
            }
        return Either.left(equation + "=0 did not converge given x0=" + x0 + ", maxTries=" + maxTries + ",
and tolerance=" + tolerance);
    }
    public static void main(String[] args) {
       Newton newton = new Newton("cos(x) - x", (double x) -> Math.cos(x) - x, (double x) -> -Math.sin(x)
- 1);
       Either<String, Double> result = newton.solve(1.0, 200, 1E-7);
       result.apply(
               System.err::println,
                aDouble -> {
                   System.out.println("Good news! " + newton.equation + " was solved: " + aDouble);
        });
    }
    private final String equation;
    private final DoubleFunction<Double> f;
    private final DoubleFunction<Double> dfbydx;
}
```