Thematic Welcome Statistics 650/750 Week 1 Tuesday

Christopher Genovese and Alex Reinhart

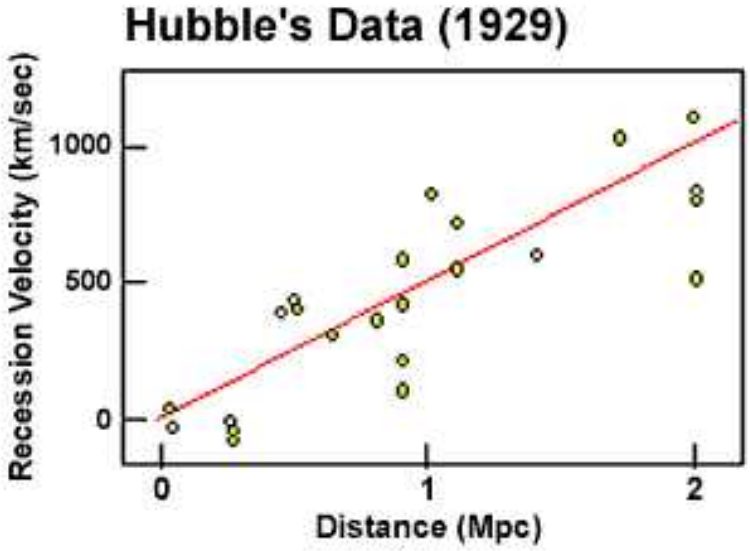
29 Aug 2017

Welcome and Introduction

Motivations

Modern data analysis can be a complex business

- Not too long ago:



- Now: cleaning and processing the data, performing analyses, and visualizing/summarizing the results can all require complicated logic and algorithms.

Creating good software to manage this complexity has become an essential skill for statisticians.

Computing is taught *at the margins* in most statistics curricula

- Typical statistical computing courses focus on the details of methods and algorithms for various concrete problems.
- Students are expected to learn the practice of computing and software engineering implicitly during their research.
- Typical feedback and incentives (e.g., get the paper done, conference deadlines, what are the results?) can obscure the benefits of building good software.

The organic development cycle for statistical software in research

1. Start with questions, ideas, objectives – sometimes incomplete
2. Try many different approaches and methods
3. The code starts out rough and quick, supporting these approaches.
    - Assumptions about the data and algorithm get baked in
    - Meaning and documentation are sparse
    - Structure and design are secondary to "getting it working"
    - There are working examples but few distinctive tests
4. Once the paper is out, it's time to move on...
    - Build on existing code base despite flaws
    - If it's used, extensions build on top of that edifice, possibly for some time.
5. Repeat

This process can limit correctness, clarity, and reusability of the code.

Building efficient, elegant, reusable software increases our productivity

Good software engineering emphasizes:

- Managing complexity
- Communicating clearly
- Finding effective abstractions
- Crafting well chosen solutions to problems
- Obtaining good performance, reuse, and generalizability

Sound familiar?

Programming well is lots of fun

It involves problem solving, design, creativity, making cool tings happen. . . great stuff.

Design and Analysis Activity

Towards A Simple Design Method

Most interesting problems reward careful thought about the nature of the problem for finding a solution. We would like to build a set of ideas and methods for solving complex problems with computer programs.

Toward that end, consider the simple method of answering the following five questions:

1. What data (entities, state, input parameters) do we need to solve the problem?
2. How might we represent those data?
3. What operations do we need to perform on those data?
4. How do we arrange those operations to find a solution?
5. How do our answers to these questions change when we generalize or modify the problem?

An illustrative example: FizzBuzz

We will start with a simple example that illustrates the method.

Problem: Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

What data (entities, state, input parameters) do we need to solve the problem?

The data we are operating on here is the *sequence* of numbers from 1 to 100.

How might we represent those data?

Each number has a simple representation, but because the sequence is ordered and regular, we have a choice in how we represent it.

- Explicitly: an array or list containing all the numbers in sequence
- Implicitly: a counter (or other "iterator") that exposes one number at a time

What operations do we need to perform on those data?

Considering what we have to check and do for each number, there are three basic operations:

1. `divisible_by_three?(number)` – Is `number` divisible by three? True or false.
2. `divisible_by_five?(number)` – Is `number` divisible by five? True or false.
3. `output_token(number)` – String to print for given number.

Note that we can specialize these operations in several ways, and we could add `divisible_by_fifteen?(number)` to this list. The `output_token` operation uses the others.

These operations will be represented by *functions* in our program.

How do we arrange those operations to find a solution?

In this case, the arrangement is a linear sequence:

```
for each integer from 1 to 100:
    print output_token(number) on its own line
```

What we can see from this is that the problem reduces to computing the output token for a single number. That's progress.

There are several approaches to `outputToken`. Here is one that uses a pre-defined table (starting index 1):

```
tokens = ("Fizz", "Buzz", "FizzBuzz")

define output_token(number):
    which_token = 0
    add 1 to which_token if divisible_by_three?(number)
    add 2 to which_token if divisible_by_five?(number)

    if which_token is 0:
        return as_string(number)
    else:
        return tokens[which_token]
```

Of course, just because we think of the operations as functions does not mean that we have to code them that way. Here is a more direct approach (in pseudo-code):

```
For each integer n from 1 to 100:
   if n is divisible by 3:
       print "Fizz"
   if n is divisible by 5:
       print "Buzz"
   if n is not divisible by 3 or 5:
       print n
   print newline
```

How do our answers to these questions change when we generalize or modify the problem?

Possible generalizations:

- Different ranges or sets of numbers
- Different sets of divisors (and possibly more than two)
- Different tokens

Our solutions would like change with these generalizations, but some more than others.

A Richer Example: The Game of Life

Background

The Game of Life is a deterministic, discrete-time process devised by mathematician John Conway that is an example of what is called a *cellular automaton*.

A cellular automaton is an arrangement *cells* in a grid of some shape that evolves in discrete time according to fixed rules. At any time step in the process, each cell can be in one of a finite number of states. The automaton is specified by a set of **rules** that describe how the state of each cell in the *next* time step depends on the state of the cell and its neighbors at the *current* time step.

In the Game of Life, the cells are the squares in an infinite, two-dimensional, regular grid in the Euclidean plane. The cells have two possible states – `Alive` and `Dead` – and each time step is called a *generation*.

Conway's rules specify that:

- Any live cell with fewer than two live neighbors dies. (Think isolation.)
- Any live cell with two or three live neighbors lives on. (Think happy.)
- Any live cell with more than three live neighbors dies. (Think overcrowding.)
- Any dead cell with exactly three live neighbors comes alive. (Think reproduction.)

It is possible to configure the cells to produce a wide variety of interesting patterns including patterns that oscillate and move, patterns that grow without bound, and patterns that generate other patterns at regular intervals. In fact, the Game of Life can even be with configured with patterns that simulate boolean logical operations, clock pulses, memory, and instructions – making it a programmable computer.

Here is a simulation to play with.

Task

Simulate the game of life starting with an arbitrary input configuration.

Design

Let's try to answer the five questions.

What data (entities, state, input parameters) do we need to solve the problem?

- Alive or dead state of all cells
- Locations of all alive cells
- Size of the universe (finite grid only)
- Geometry of the grid

How might we represent those data?

- Sparse array of cell states
- An array of cell "objects" (state, maybe # neighbors, etc)
- List of tuples (x,y) representing live cells
- Hash table mapping locations to live cells

What operations do we need to perform on those data?

- change cell state
- create new generation
- count number of live neighbors for a given location
- randomly sample cells
- is a cell at the edge?
- get neighbors of all live cells
- get neighbors of a given location
- apply rules
- render or visualize the universe

How do we arrange those operations to find a solution?
Fixed array case

1. Create new generation (another fixed array)

2. For each cell (in some order) Look at/count neighbors of that cell Apply the rules Change cell state in the next generation

3. Render next generation and update

Tuples of live cell locations

1. Make list of all neighbors of live cells

2. Count how many times each occurs

3. Apply rules

(1 2) (20 40) (1 4) (7 5)...
(1 3) (1 1) .... (21 40) (21 41) ... (1 3) (1 5) (2 4)...
(1 3) 2, (21 20) 3, (1 1) 1, ...

How do our answers to these questions change when we generalize or modify the problem?

Generalizations to consider:

- Accept any of the basic rules sets determined by the number of neighbors required to produce a live cell from live or dead cells.
- Work on a *hexagonal grid*
- Require an infinite grid

(Thoughts)

Main Themes of the Course

Theme #1: Good programming practice

- Will future you (weeks or months or years from now) have a clue what your code does?
- Will a collaborator?
- Are the names of variables, functions, and classes meaningful?
- Is the code formatted in a consistent and readable manner?
- Is the code well documented? Does the documentation match the code?

- What is the role of comments? What makes a comment helpful? When should the code speak for itself?
- Can you *read* the code? Can you tell what the algorithm is?
- Does the organization of the code help the reader understand the code's purpose?
- How do we achieve the proper balance between error checking, readability, and performance? What determines where errors are "handled"?
- How do we ensure correctness of our programs?

Theme #2: Good tools and Efficient workflows

Good programming practices can make you more productive because it is easier to write correct, reusable, and generalizable code. But it also helps to use good tools.

Essentials:

- A good editor (*Emacs*, Vim, Sublime, ...) or IDE (RStudio, Eclipse, IPython)
- Version control (e.g., git) for managing changes and collaborative development
- Debugger for finding tricky bugs
- Profilers for performance tuning
- Databases for managing complex data sets
- Testing framework to easily run and check your tests
- Documentation format for writing/disseminating documentation on your code

Brief aside on editors

- There are many good editors Emacs, Vim, Sublime Text, Atom, ...
- Any good editor has a learning curve but it makes you more powerful in several ways
- Recommendations and support

Theme #3: Good software design

- Does your code repeat itself frequently?
- If you change one part of the code, what else needs to change?
- Does each function or object only have access to the information it needs?
- How easy is it to reuse parts of your code?
- How easy is it to adjust the code to solve a more general problem?
- How easy is it to reason about the structure of your program?

Consider:

```
if(thisLife.equals("X"))
{
        if(count==2 || count==3)
        {
                nextLife = "X";
        }
}
else
{
        if(count==3)
                nextLife="X";
}
tempData.put(i, nextLife);
```

Theme #4: Good choice of representations, data structures, and algorithms.

- How does the runtime (or memory...) of your program scale as the size of the problem increases?
- Does your representation of the problem allow for an elegant/efficient solution?
- Does your data representation match well with the types of operations you are doing?

- What are the performance characteristics of the algorithm you are using?
- Can you use a well-tested library or package to run the algorithm?

Theme #5: Killer Apps!

Putting the pieces together to solve interesting and challenging statistical problems.

Examples:

- Identifying audio files from small snippets
- Estimating object velocities from images
- Spell checking and text completion
- Searching spatial and high-dimensional data sets
- Training agents to play interactive games
- Classifying images and identifying objects
- ...

Course Design Highlights

What we believe

- A broad and firm foundation in computing will pay off throughout your career
- The way to get better at programming is to **practice** programming
- Good software design and programming practice are skills every statistician needs
- Revision is a critical part of the development process
- Having (at least a passing) understanding of multiple languages will make you a better programmer

Learning Objectives

By the end of this course, you should be able to

- develop correct, well-structured, and readable code;
- design useful tests at all stages of development;
- effectively use development tools such as editors/IDEs, debuggers, profilers, testing frameworks, and a version control system;
- build a moderate scale software system that is well-designed and that facilitates code reuse and generalization;
- select algorithms and data structures for several common families of statistical and other problems;
- write small programs in a language new to you.

What will we do in class?

Classes will feature a combination of lectures, interactive discussion and problem-solving, and single/group programming activities.

You should bring your laptop to every class

Resources

- All course materials and course work will be accessed through `github`.

  You should sign up for a free account at `https://github.com/`.

  Then, visit `https://classroom.github.com/a/MMfm3qKz` to obtain access to the course repositories and your homework assignment repositories.

  Key course materials:

Repository	Contents
`https://github.com/36-750/documents`	Announcements, lecture notes, data, and other
`https://github.com/36-750/problem-bank`	Assignment descriptions and associated files an
`https://github.com/36-750/git-demo`	A supplementary demo for tinkering with git

- Several useful books will be available for you to borrow anytime. They are listed in the documents repository and are available (for the moment) from Professor Genovese. *Do not hesitate to ask for them.*

- The instructors will be available to meet to answer your questions. Scheduling of office hours will be data-driven.
- You will interact with the TAs mostly through discussions on github, but they will also be available to answer questions about their feedback. They may hold fixed hours or be available by appointment depending on need.

Assignments

Details on assignments are spelled out in the syllabus. **Read the Syllabus**

The work in this course consists of programming and related exercises, including revision of your own programs.

Assignments are drawn from a large repository of problems that vary in length, complexity, topic, and scope. This repository is available on `github` via `https://github.com/36-750/problem-bank` and will be expanded as the semester proceeds.

You will submit assignments and revisions, receive feedback from the TAs, and ask questions about that feedback using "pull requests" on this repository. We will describe the procedure in detail before the first assignment

Because revision is an important part of the development process, we allow you to revise (and refine based on feedback) your submitted assignments. Each assignment or challenge project may be revised at most twice.

Note that feedback from the TA is contingent on meeting the basic requirements for submission. So for example, if required tests or scripts are missing or if there are significant deficiencies with coding style and organization, the review will stop at noting that issue and you will have used a submission/revision for little gain.

You can submit an assignment at most twice per week, and revisions to a previous submission count. This gives you 28 opportunities during the semester to submit an assignment. There will also be two scheduled challenges during the semester, each of which will extend over more than a week. When planning your submissions, please remember that you will need to spend time on the challenges.

Grading

Each exercise in a stand-alone or vignette assignment is graded as either "Mastered" or "Not yet mastered." Challenges are graded as "Sophisticated," "Mastered," or "Not yet mastered." The meaning of these classifications and

the criteria required to achieve them is spelled out in the course grading rubric, which is available in the documents repository. Each challenge will have its own specialized rubric.

Key points:

- It's OK to make mistakes.
- Try to find ways to expand your skills and perspectives.
- Ask for help if you need it, from us and your peers.
- Pay attention to the rubric.
- Practice, revise, repeat.
- Challenge yourself.

Grades for the course are determined by the number of "Mastered" and "Sophisticated" marks that you receive, with adjustments for participation and other considerations. See the syllabus for the specifics.

Note you are therefore **not penalized** for "Not yet mastered" grades, and the purpose of revisions is to improve your programs.

Request

Please put `[650]` or `[750]` in your email subject lines!

A Few Words about Programming Languages

What are computer languages for?

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 89 b6  00 00 00 00  39 d3 7e 13  29 c3 39 c3
75 86 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

This collection of hexadecimal numbers represents code to compute the GCD of two integers via Euclid's algorithm. This is given in the machine language of the Intel x86 microprocessor's, which depends on the inner works of the hardware and is difficult to make sense of.

A little better, but not much:

```
   pushl %ebp                      jle  D
   movl  %esp, %ebp                subl %eax, %ebx
   pushl %ebx                   B: cmpl %eax, %ebx
   subl  $4, %esp                  jne  A
   andl  $-16, %esp             C: movl %ebx, (%esp)
   call  getint                    call putint
   movl  %eax, %ebx                movl -4(%ebp), %ebx
   call  getint                    leave
   cmpl  %eax, %ebx                ret
   je    C                      D: subl %ebx, %eax
A: cmpl  %eax, %ebx                jmp  B
```

This is *assembly language* for the same hardware, a more "readable" version of the machine code where the numbers for particular operations and parts of the processor are given names.

We can figure out a few pieces immediately, and with a good look at the algorithm and a few clues, we can work it out. But despite a few cases where it might make sense to program this way, this is too tiring for any program of substantial complexity.

We would like to be able to program in a way that makes the concepts clearer and that offers a wider range of abstractions.

Let's look at the algorithm. The key idea is that if a = q b + r (with non-negative integers) then GCD(a,b) = GCD(b,r). Here's one way to express that idea.

```
Euclid's Algorithm

Input: Two non-negative integers a and b.
Output: The greatest common divisor (GCD) of a and b.

1. If a < b, swap a and b.  (So, we can assume a >= b.)
2. If b is 0, return a.
2. Divide a by b to express  a = q b + r for non-negative
   integers q (quotient) and r (remainder).
3. If r is 0, return b.
4. Otherwise, replace a by b and b by r.
5. Return to step 2.
```

Notice that in this description we have introduced some **abstractions**, from familiar ideas such as comparing and dividing integers to more slippery ideas such as swapping and replacing the values of variables.

Programming languages give us a syntax to express abstractions by which we can represent data and describe algorithms easily, clearly, and efficiently. Here, for instance, is an attempt to express the above algorithm:

```
gcd <- function(a,b) {
    while ( a != b ) {
        if ( a > b ) {
            a <- a - b
        } else {
            b <- b - a
        }
    }
    return(a)
}
```

Does it succeed? Why or why not?

Why is there more than one programming language?

There is a dizzying number of programming languages, with new languages being developed regularly. Some are used by thousands, some by only a handful. Many are general purpose; others fill a very specialized niche. Why are there so many?

A few reasons:

- Evolution of ideas
- Performance trade-offs
- Specialized purposes
- Personal preferences and styles
- Different criteria for "better"
  - Expressive power

- Ease of learning (for a novice)
- Library support
- Community support
- Acceptance in the workplace
- Standardization
- Availability of efficient compilers
- ...

We believe that learning more than one programming language is valuable, even if you do not use the other languages for your main work.

One reason for this is that the structure of a language and the constraints it imposes affect how you think about and approach problems. Learning a new language, especially one very different from one you are familiar with, opens up new ways of thinking. (This is a technological version of the Sapir-Whorf hypothesis.)

In this course, you should plan to do some (e.g., two or three assignments) *basic* work in a language outside your comfort zone. We are happy to advise you on choices and help you get comfortable. The reserve book *Seven Languages in Seven Weeks* also gives a gentle introduction to each of several nice languages.

Some recommendations

C++	A powerful but complex object-oriented language, works with Rcpp	
Javascript	The heart of web-based programming	
Java	Versatile object-oriented language	
Clojure	An elegant functional language with immutability, concurrency, multiplatform	*
Rust	A fast and modern systems language	*
Racket	A well-designed, modern lisp with a well-supported ecosystem	*
Haskell	A pure and mind-blowing functional language	*
Python	A clean language that is commonly used for data science	

Books and support materials for these languages are available

Appendix: A Rough Taxonomy

As with spoken languages, programming languages have evolved across a family tree of related ideas and syntax. But it is worth looking briefly at a few distinctions.

Imperative versus Declarative

Imperative languages focus on **how** the computer should perform a task. Declarative languages focus on **what** that task is.

Familiar languages such as C, R, Java, C++, and Python are imperative, with some distinctions within that class. (Languages like R and Java do provide some support for declarative features.)

Here is another R version of the algorithm:

```
gcd <- function(a,b) {
    while ( b != 0 ) {
        last_b <- b
        b <- a %% b
        a <- last_b
    }
    return(a)
}
```

We have three variables which we set and re-set, telling the computer the exact steps to take. This function accepts inputs and computes outputs. We might for instance compute gcd(44,36).

In contrast, here is a declarative expression of Euclid's algorithm, in a logic language called Prolog.

```
gcd(A,B,G) :- A = B, G = A.
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).
```

The program specifies a sequence of rules and facts and accepts queries from which it deduces the answers. The symbol :- can be read as "if," and the clauses on the right-hand side are separated by "and."

We might give a query `gcd(44,36,?g)`. The system would respond with `?g = 4`.

Imperative Languages: Von Neumann versus Object-Oriented

The **Von Neumann** languages like C and Fortran tend to work "close to the hardware."

- They natively represent only a few primitive types of data: integers, fixed-precision floating-point numbers, strings, arrays, and and records that aggregate these other types.

  In C, for instance, memory addresses are stored and manipulated directly as "pointers."

- The fundamental abstractions are iterative loops, conditional statements, and functions. Programs are organized into a collection of functions that can be called to compute the desired result.

- The basic unit of code is a *statement*, either a simple statement which represents one "step" in the computation

```
i = i + 1;
calculate_foo();
write_to_file(a[i]);
```

  or a compound statement that groups other statements within a loop or conditional

```
while( index > 0 ) {
    position[index] = 2*index + 1;
    printf("%d\n", position[index]);

    if ( index > 10 ) {
        index -= 2;
    } else {
        index--;
    }
}
```

The **Object Oriented** languages like Java, C++, Python, Ruby, and Smalltalk model the problem domain in terms of its *nouns*, which are called **objects**. Each entity in the problem is represented by a class of objects that define the data they store and the behaviors (verbs) that they can exhibit. A program is often crafted by defining appropriate classes of objects and letting them interact.

- Object-oriented languages typically have more elaborate systems of types arranged in a hierarchy. Classes of objects and various collections of them all represent types with various relations among them.
- The class/object is the fundamental abstraction, intended to promote clarity and code reuse. Programs are organized by classes that each encapsulate their own data, behavior, and interface with the other objects in the system.
- The basic unit of code is still a statement, but the flavor of these languages more to abstraction than the Von Neumann languages.

```
import math

class Point(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def toString(self):
        return "(%f,%f)" % (self._x, self._y)

    def radius(self):
        return math.sqrt(self._x * self._x + self._y * self._y)


p = Point(3.0, 4.0)
print p.radius()
```

Declarative Languages: Functional, Dataflow, Logic

Functional languages like Clojure, Haskell, and OCaML represents computation as the threaded evaluation of functions operating on data. In contrast to object-oriented languages, functional programming models the problem domain through its *verbs*.

- Functional languages tend toward the ethos stated by Alan Perlis: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

- The function is the fundamental abstraction. Functions can be created on the fly and can be passed and returned to and from other functions.
- The basic unit of code is the expression, typically a function call of some sort. Even conditonals and loops are expressions. Function calls are often in **prefix order**, e.g., "+ x y" rather than "x + y".
- Functional languages tend to prefer immutable data and often use recursive logic.

```haskell
gcd :: (Integral a) => a -> a -> a

gcd a 0 = a
gcd a b = gcd b (a `rem` b)
```

```clojure
(defn gcd
  "(gcd a b) computes the greatest common divisor of a and b."
  [a b]
  (if (zero? b) a (recur b (mod a b))))
```

Dataflow languages (like spreadsheets) model a program by the flow of data through operations in a directed graph. Changes to data at any point in the graph propagates throughout the system. For example, changing a cell in a spreadsheet updates the results of other cells that depend on it.

Logic languages like Prolog represent a problem by a sequence of facts and logical rules. Computation of the result is obtained by deriving the logical outcome of a query given particular inputs.

Crossing the families

Many languages do not fit easily into a single paradigm. Common lisp, for instance, has sophisticated imperative, functional, and object-oriented features. R looks like a Von Neumann language, but has several different object systems and a number of functional features built on top of that. Javascript is often treated as an object-oriented language but fits other paradigms (including functional) quite well.

Languages Everywhere

There are other types of languages. An important example are languages designed for concurrent processing, such as Erlang.

There are also many languages specialized for particular tasks and domains, including database operation, specifying web pages, graphical and design specifications, typeset documents, and electronic circuits.

It is also common to define **domain specific languages**, mini languages that operated within some narrow context. This includes, for instance, the formal syntax in Excel and the specification of tikz graphs in LaTeX.

Plan going forward

- Phase I Tools (Version Control, Command Line Shell, ...)
- Phase II Practices (Style, Organization, Design, Testing, Profiling, ...)
- Phase III Algorithms and Data Structures
- Phase IV Putting it all together

Upcoming Assignments

Assignment tags: git, intro-vignette

Thursday:

- Read the syllabus
- Sign up for github and setup your class repositories
- We will use this in the next class

Next Tuesday:

The *Git and Shell Challenge* Assignment

Appendix: Some FizzBuzz Solutions

Here are some solutions to the FizzBuzz problem in various languages. It's worth studying to see the commonalities and differences across languages

in how these basic ideas (iteration, calling functions, conditionals) are expressed.

A few questions to consider as you look at the solutions:

- How do we evaluate the various solutions?
- Is the meaning of the code clear? Can you *read* the code?
- What complexity is worthwhile?
- How easy is it to generalize this code? For instance, if we wanted to specify the divisors how could we do it?
- How does the algorithm scale with the size of the problem?

```
for i = 1:100
    if i % 15 == 0
        println("FizzBuzz")
    elseif i % 3 == 0
        println("Fizz")
    elseif i % 5 == 0
        println("Buzz")
    else
        println(i)
    end
end
```

```
#include <stdio.h>

int main (void)
{
    for ( int i = 1; i <= 100; i++ )
    {
        if ( i % 15 == 0)
            printf ("FizzBuzz");
        else if ( i % 3 == 0 )
            printf ("Fizz");
        else if ( i % 5 == 0)
            printf ("Buzz");
```

```
        else
            printf ("%d", i);
        printf("\n");
    }
    return 0;
}
```

```
fizzBuzz2 <- function(n.max) {
    values <- 1:n.max
    ifelse(values %% 15 == 0, "FizzBuzz",
           ifelse(values %% 5 == 0, "Buzz",
                  ifelse(values %% 3 == 0, "Fizz",
                         values)))
}

fizzBuzz3 <- function(n.max) {
    values <- 1:n.max
    divisible.by.3  <- (values %% 3)  == 0
    divisible.by.5  <- (values %% 5)  == 0
    divisible.by.15 <- (values %% 15) == 0
    values[divisible.by.3]  <- "Fizz"
    values[divisible.by.5]  <- "Buzz"
    values[divisible.by.15] <- "FizzBuzz"
    values
}
```

```
def fizzBuzz1(n_max):
    "Simple and direct"
    for n in range(1,n_max+1):
        if (n % 15) == 0:
            print "FizzBuzz",
        elif (n % 3) == 0:
                print "Fizz",
        elif (n % 5) == 0:
            print "Buzz",
        else:
            print n,
```

```
12
13 def fizzBuzz2(n_max):
14     "Some fancy language features"
15     for n in range(1, n_max+1):
16         print "Fizz" * (n % 3 == 0) + "Buzz" * (n % 5 == 0) or n,
17
18 def fizzBuzz3(n_max):
19     assert isinstance(n_max, (int, long))
20     for n in range(1, n_max + 1):
21         divisible_by_5 = (n % 5) == 0
22         if (n % 3) == 0:
23             if divisible_by_5:
24                 print "FizzBuzz",
25             else:
26                 print "Fizz",
27         elif divisible_by_5:
28             print "Buzz",
29         else:
30             print n,
```

```
1 mapM_ (putStrLn . fizzbuzz) [1..100]
2
3 fizzbuzz x
4     | x `mod` 15 == 0 = "FizzBuzz"
5     | x `mod`  3 == 0 = "Fizz"
6     | x `mod`  5 == 0 = "Buzz"
7     | otherwise = show x
```

```
1 (defn fizzbuzz []
2   (let [natural-numbers (drop 1 (range))
3         label           (fn [n]
4                           (cond
5                             (zero? (mod n 15)) "FizzBuzz"
6                             (zero? (mod n 3)) "Fizz"
7                             (zero? (mod n 5)) "Buzz"
8                             :else n))]
9     (map label natural-numbers)))
```

```
10
11 (take 100 (fizzbuzz))
```