Introductory Computer Programming

Deepayan Sarkar

About this course

- Compulsory non-credit course (pass marks: 35%)
- Does not count towards composite score, but you need to pass
- Syllabus
  - Basics in Programming: flow-charts, logic in programming
  - Common syntax
  - Handling input/output files
  - Sorting
  - Iterative algorithms
  - Simulations from statistical distributions
  - Programming for statistical data analyses: regression, estimation, parametric tests

Exercise

- Think of tasks that cannot be easily done without a computer
- Could be both related and unrelated to what you are studying

Some specific examples

- Can be solved using scalar variables only:
  - Is a given natural number $n \in \mathbb{N}$ prime?
  - Given integer $k \geq 0$, compute its factorial $k!$, and $\log k!$
  - Given integers $n, k \geq 0$ such that $k \leq n$, compute $\binom{n}{k}$
- Probably need vector objects to be solved:
  - Find all prime numbers less than a given number N
  - Sort a given collection of numbers
  - Produce a random permutation of a given set of numbers
  - Given set S and query object x, determine whether $x \in S$ (set membership)

Some examples of simulation

- Simple random walk (+1 or -1 with probability p and $1 - p$):
  - How long does it take to return to zero for the first time?
  - When was the last return to zero before time $2n$?

- Toss a coin (with probability of head p) until you get k consecutive heads.
  - Based on observed value, can you test for $p = \frac{1}{2}$?
- Given a game of snakes and ladders, how many throws of the dice does it take to reach the end?
- Shuffle a deck of cards.
  - How can we probabilistically model a shuffle?
  - How many times do we need to shuffle to make the deck approximately random?
  - How can we "test" for randomness?

Some general problems

- Given a function f, solve for $f(x) = 0$, e.g.,
  - solve non-linear equations like $e^x + \sin x = 0$
  - solve linear equations (e.g., as part of fitting linear models)
- Optimization: given a function f, find x where $f(x)$ is minimized
  - Sometimes this can be done by solving $f'(x) = 0$
- Solution used usually depends on context

Algorithms

- We will spend a lot of time discussing algorithms
- An algorithm is essentially a set of instructions to solve a problem
- Algorithms usually require some inputs
- Instructions are executed sequentially, finally resulting in an output
- You can think of an algorithm as a recipe (inputs: ingredients, output: food!)

Example: is a given number n prime?

- Basic idea: see if n is divisible by any number between 2 and $n - 1$
- Obviously, enough to check is n is divisible by any number between 2 and $\sqrt{n}$
- Intuitively, the second approach is more "efficient"
- We will usually write algorithms in the form of *pseudo-code* as follows:

```
is_prime(n)

i := 2
while (i ≤ sqrt(n)) {
    if (n mod i == 0) {
        return FALSE
    }
  i := i + 1
}
return TRUE
```

- The meaning of this algorithm / pseudo-code should be more or less obvious
- Assumes availability of certain basic operators / functions (mod, sqrt)

- We often employ some *conventions* and use some *structures* in pseudo-code
- For example,

```
is_prime(n)

i := 2                    // variable assignment
while (i ≤ sqrt(n)) {     // loop while condition holds
    if (n mod i == 0) {   // branch if condition holds
        return FALSE      // exits with output value
    }                     // end of blocks within loops, branches, etc.
    i := i + 1            // update variable value
}
return TRUE
```

- These conventions are not standard; alternative forms could be:

```
is_prime(n)

i = 2 // different assignment operator
while i ≤ sqrt(n) // end of loop indicated by indentation
    if n mod i == 0
        return FALSE
    i = i + 1
return TRUE
```

```
is_prime(n)

i <- 2 // yet another assignment operator
while i ≤ sqrt(n) // end of loop indicated by end keyword
    if n mod i == 0
        return FALSE
    end
    i <- i + 1
end
return TRUE
```

Theoretical questions about algorithms

- **Is an algorithm correct?** To be correct, an algorithm must
  - stop after a finite number of steps, and
  - produce the *correct output* for *all possible inputs* (i.e., all *instances* of the problem).
- **How efficient is the algorithm?**
  - What resources does the algorithm need to run, typically in terms of time and storage?
  - How does it compare with other algorithms for the same problem?
- To answer such questions, we need a model for computation

Ingredients of a computational model

- There are actually many different approaches to programming
- We will mostly consider structured programming
- Characterized by use of various control flow constructs (if, then, while, for, etc.) and block structures

- More specifically, we will focus of procedural programming
- Characterized by use of modular procedures (usually called functions)
- We are mainly interested in procedures that perform some computations
- Most algorithms we will discuss directly correspond to procedures or functions when actually implemented
- We will not discuss other kinds of programs (e.g., operating system, web browser, editor, etc.).

Functions and control flow structures

- The main components of our programs are going to be functions.
- Usually a programming language will have many built-in functions
- Additional libraries or packages will provide more standard functions
- Functions usually
  - have one or more input arguments,
  - perform some computations, possibly calling other functions, and
  - return one or more output values.
- The main contribution of a function is the second step
- The standard model for performing computations is **sequential execution**
- In other words, a function executes a set of instructions in a specified sequence
- Some control flow structures may be used to create branches or loops in the flow of execution
- Briefly, the main ingredients used are:
  - Declaration of variables (implicit in some languages). *The details of how variables store values, and who can access them (scope) are important, and will be discussed later.*
  - Evaluation of expressions. *Can involve variables provided they have been defined in an earlier step.*
  - Assignment to variables (to store intermediate results for later use).
  - Logical tests (equal?, less than?, greater than?, is more input available?).
  - Logical operations (AND, OR, NOT, XOR).
  - Branching - take different paths based on result of a logical operation (if-then-else).
  - Loops - repeat sequence of steps, usually a fixed number of times, or while a condition holds (for / while).

Common operators (may have language-specific variants)

- *Mathematical operators*:
  - `+` (addition)
  - `*` (multiplication)
  - `/` (division — possibly integer division)
  - `^` (power)
  - `%` (the modulo operation)
- *Logical operators*:
  - `&` (AND)
  - `|` (OR)
  - `!` (NOT)
- *Comparisons*:

    - `==` (equality)
    - `!=` ($\neq$)
    - `<`, `>` (strictly less than or greater than)
    - `<=` `>=` ($\leq$, $\geq$)
- *Mathematical functions*: `round, floor, ceil, abs, sqrt, exp, log, sin, cos, ...`

Practical implementation: programming languages

- The algorithms we discuss can be implemented in many programming languages
- Some standard languages suitable for structured programming are
    - C (compiled)
    - C++ (compiled)
    - R (interpreted)
    - Python (interpreted)
    - Julia (interpreted)
- There are also many others with various relative strengths and weaknesses
- In this course, we will mainly focus on
    - **R** because it already has an extensive collection of statistical software that we can use
    - **C** / **C++** because it is easy to call C / C++ code from R (useful when R code is inefficient)

Example: The `is_prime` algorithm in various languages

- Recall the `is_prime` algorithm to determine if a number is prime
- With slight modification to use only integer arithmetic

`is_prime(n)`

i := 2
while (i * i $\leq$ n) {
  if (n mod i == 0) {
    return FALSE
  }
 i := i + 1
}
return TRUE

- Implemented in C, the algorithm would look like this:

```c
int is_prime_c(int n)
{
    int i = 2;
    while (i * i <= n) {
        if (n % i == 0) {
            return 0;
        }
        i = i + 1;
    }
    return 1;
}
```

- C is a compiled language, so actually running this code involves some additional work
- Note that all variable *types* need to be explicitly declared

- This includes the types of function arguments (inputs) and return value (output)
- The same algorithm would look like this in R:

```
is_prime_r <- function(n)
{
    i <- 2
    while (i * i <= n) {
        if (n %% i == 0) {
            return (FALSE)
        }
        i <- i + 1;
    }
    return (TRUE);
}
```

- The basic structure is very similar, but with some differences:
    - The assignment operator is different (but `=` also works in R)
    - The function declaration looks like a variable assignment
    - The modulo operator is `%%` instead of `%`
    - Uses `TRUE` and `FALSE` instead of `1` and `0` for logical values
    - Statements do not end with a semicolon (although they could)
    - Variable types are not declared
    - The return value must be put in parentheses
- We can call this function after starting R and copy-pasting the function definition

```
is_prime_r(4)
```

```
[1] FALSE
```

```
is_prime_r(10)
```

```
[1] FALSE
```

```
is_prime_r(100)
```

```
[1] FALSE
```

```
is_prime_r(101)
```

```
[1] TRUE
```

- The implementation looks a little different in Python:

```
def is_prime_py(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return 0;
        i = i + 1
    return 1
```

- The main difference is that indentation defines code blocks
- Changing indentation will change meaning of code, which does not happen in C or R
- However, code in all languages *should be indented properly for readability*
- Again, we can start python, define the function, and run the following code

```
print(is_prime_py(4))
```

```
0
```

```
print(is_prime_py(10))
```

```
0
```

```
print(is_prime_py(100))
```

```
0
```

```
print(is_prime_py(101))
```

```
1
```

How can we run C / C++ code?

```c
#include <stdio.h>
#include <stdlib.h>

int is_prime_c(int n)
{
    int i = 2;
    while (i * i <= n) {
    if (n % i == 0) {
        return 0;
    }
    i = i + 1;
    }
    return 1;
}


int main(int argc, char *argv[])
{
    int i, n;
    if (argc > 1) {    /* one or more arguments supplied  */
    for (i = 1; i < argc; i++) {
        n = atoi(argv[i]);  /* converts string to integer */
        printf("%d -> %d\n", n, is_prime_c(n));
    }
    }
    else printf("Usage: %s <n1> <n2> ...\n", argv[0]);
    return 0;
}
```

- The code needs to be "compiled" before it is run
- It also needs a `main()` function to be defined
- `main()` is run first when the program is executed
- Here is a complete file that can be compiled
- How to compile & run depends on the operating system

```
gcc -o is_prime cdemo/is_prime_wrapper.c
./is_prime
```

```
Usage: ./is_prime <n1> <n2> ...
```

```
./is_prime 4 10 100 101
```

```
4 -> 0
10 -> 0
100 -> 0
101 -> 1
```

Compiled code vs interpreted code

- R, Python, etc., are "interpreted" languages that read and evaluate code interactively
- Compiled code is usually (but not always) much faster than interpreters
- Most interpreters are themselves written in a compiled language
- However, compiled languages have several disadvantages:
  - They are not interactive!
  - Trying out ideas (edit-compile-run) takes longer
  - Most importantly: limited initial set of tools
  - For example, you will need to write your own functions to import data, make plots, etc.
- Ultimately, choice depends on the purpose of the program
- We will mainly use R (to take advantage of its many useful features)
- We will not write C programs designed to be run directly
- However, we *will* sometimes call C / C++ code **from R** to take advantage of its speed
- The easiest way to do this is using a *package* called Rcpp
- Python code can similarly be called using the reticulate package
- And Julia code can be called using the JuliaCall package
- I will give an example of Rcpp to illustrate its usefulness
- We will look at it in more detail after learning more about R and C

An example of using Rcpp

- The first step is to compile a C function so that it can be called from R

```
library(package = "Rcpp")
sourceCpp(code =
"

#include <Rcpp.h>

// [[Rcpp::export]]
int is_prime_c(int n)
{
    int i = 2;
    while (i * i <= n) {
        if (n % i == 0) {
            return 0;
        }
        i = i + 1;
    }
    return 1;
}
```

```
")
```

- Alternatively, compile code in a file

```
library(package = "Rcpp")
sourceCpp("cdemo/is_prime_rcpp.cpp")
```

- The C function can then be called just like an R function

```
is_prime_c(4)
```

```
[1] 0
```

```
is_prime_c(10)
```

```
[1] 0
```

```
is_prime_c(100)
```

```
[1] 0
```

```
is_prime_c(101)
```

```
[1] 1
```

- We can call both versions on a sequence of integers as follows
- The time required is recorded using `system.time()`

```
system.time(r_primes <- sapply(1:1000000, is_prime_r))
```

```
   user  system elapsed
 11.950   0.008  11.958
```

```
system.time(c_primes <- sapply(1:1000000, is_prime_c))
```

```
   user  system elapsed
  2.454   0.016   2.471
```

- The C version is clearly faster
- Would have been even faster if the loop was also in C
- We can try this later after we discuss vectors / arrays

What is the advantage of doing this in R?

- We can use R utilities to check that the results are the same

```
sum(r_primes == TRUE)   # counts number of TRUE in a logical vector
```

```
[1] 78499
```

```
sum(c_primes == TRUE)
```

```
[1] 78499
```

```
tail(which(r_primes == TRUE))    # extracts last few elements
```

```
[1] 999931 999953 999959 999961 999979 999983
```

```
tail(which(c_primes == 1))
```

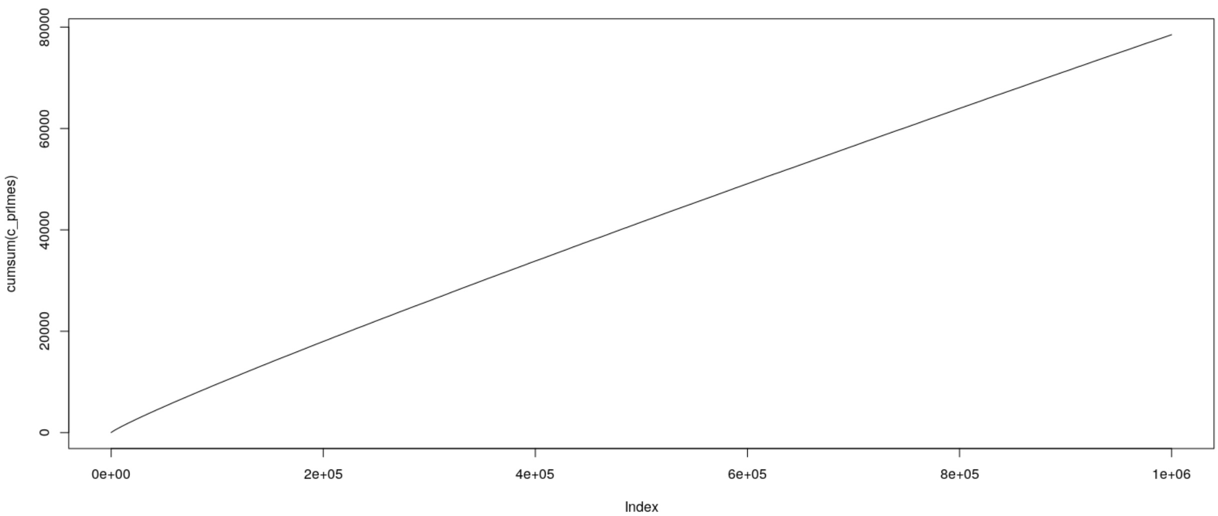
```
[1] 999931 999953 999959 999961 999979 999983
```

```
identical(r_primes == TRUE, c_primes == 1) # tests whether two arguments are identical
```

```
[1] TRUE
```

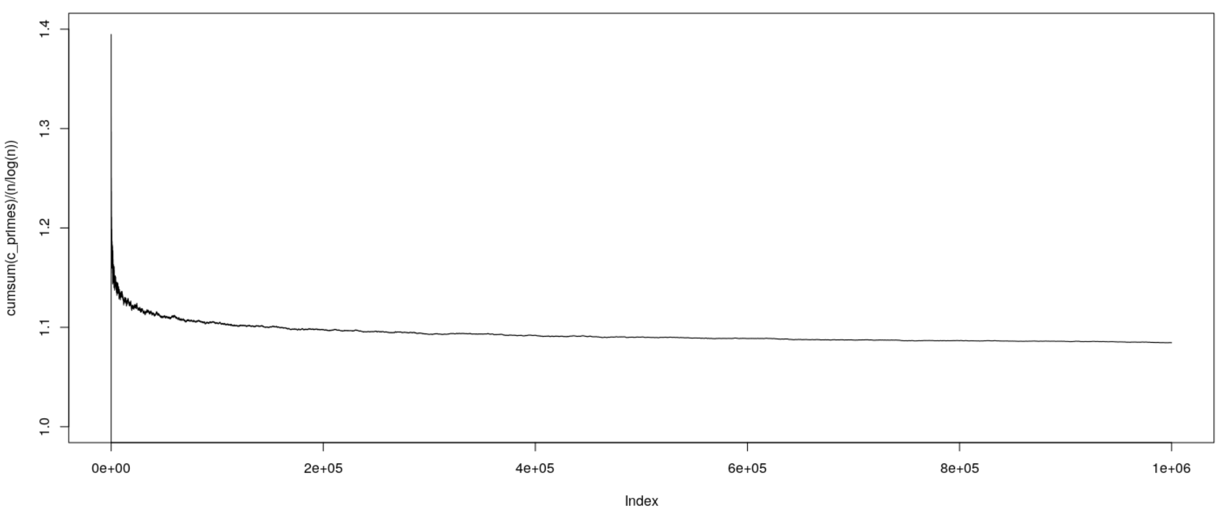
- We can use R to visualize the prime counting function $\pi(n)$

```
plot(cumsum(c_primes), type = "l")
```



- Is $\pi(n) \approx n/\log n$? (Prime Number Theorem)

```
n <- 1:1000000
plot(cumsum(c_primes) / (n / log(n)), type = "l", ylim = c(1, 1.4))
```



What next

- Over the next few classes, we will learn R more formally
- We will then come back to study algorithms in more detail