Week 5
R programming

The five points of programming: successful computer programming depends on mastering five attributes of a computer language. Good computer languages provide (at least) these capabilities:

1) The ability to execute statements and to get user input. You are already familiar with executing statements and with getting input (think of what happens when you read a data table in R).

2) A method for storing and retrieving data. You have already worked with vectors, matrices, lists (like dataframes), and other data storage objects. You have learned that different types of data are stored in objects specific to those datatypes. For example, data can be stored in a numeric, character, or logical vector.

3) Iteration: the ability to repeatedly execute one or more statements until some condition is met. You have learned two forms of iteration: the **for** loop and the **while** loop. The **for** loop assumes that, from the start, you know how many times you would like to execute a set of statements, whereas the **while** loop can execute statements indefinitely. If you are sick, a doctor might treat you with a **for** loop ("for each of 30 days, take pills"), or a **while** loop ("take pills each day while you are sick").

4) Conditional execution/branching: conditional execution is the ability to execute a set of statements depending on some condition. Fittingly, conditional execution in R uses the keywords **if**, **else if**, and **else**. Thus,

```
if it is 4:50 pm on wednesday,
    GO to R class;
else if it is 4:50 pm mon, tues, thurs, or fri
    GO to bus stop to catch bus;
else
    DO R homework, walk dog, party hard...
```

5) Use of functions: functions are extremely useful and pop up everywhere in R programming. Functions can take *arguments*, and they may have a *return value*. For example, the function **sum** takes a vector of numbers as an *argument*, and *returns* the sum.

So far, we have learned items 1-3, and this week we'll touch on item 4. Once you've mastered these five programming points, you have all of the tools you need to address almost any computational challenge that may arise.

This week we will cover:

1) review

2) random number generation

3) introduction to conditional execution (*maybe...*)

---

while loops gone wrong: a cautionary note

while loops, like U.S. presidencies, can go very, very wrong. Remember that the basic form the loop is:

```
while (SOME CONDITION is TRUE){
    execute statements
}
```

Because the loop continues to run as long as SOME CONDITION is TRUE, it is imperative that any while loops you write have an exit plan: it must always be possible for the loop condition to become FALSE. Failing to do this is a recipe for a computer crash (because the loop will continue spinning, eating up memory *ad infinitum*...).

1) Which of the following expressions might result in an infinite loop? Why?

a)
```
while(TRUE)
      print('TRUE!')
```

b)
```
while(FALSE)
      print('NOT TRUE!')
```

c)
```
while(1)
      print("having fun in the loop!")
```
(this will not be obvious. Is 1 TRUE or FALSE? How about 2? 0? 100?)

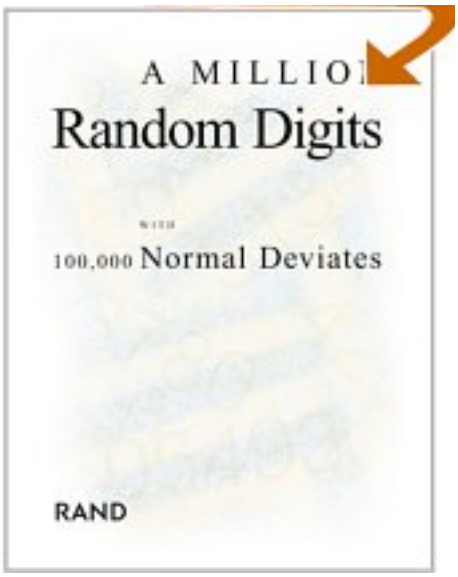
d)
```
x<-5;
while(x==5)
      cat('x =', x, sep=' ');
```

e)
```
x<-0;
while (x < 100)
     x<-x+rgeom(1, .5);
```

f)
```
x<-0;
while (x != 100)
     x<-x+rgeom(1, .333);
```

A digression on the **random number generator** (RNG)

The ability to generate random numbers quickly and easily is fundamental to biologically relevant computer programming. Almost all modeling applications and many statistical analyses require random number generators at some stage.



I really love this book published by the RAND corporation circa 1955. Apparently, when RAND wasn't developing game theoretic models of a global nuclear holocaust, they were busy writing books consisting of nothing but random numbers. Imagine how tedious even the simplest modeling projects would have been in the pre-modern computing era.

I show this to give you an idea of the power of random number generators in R (or any

language). The RAND book – which actually was used for early stochastic simulation – devotes 400 pages to the tabulation of 'one million random digits'. For comparison, let's see how long it takes to generate one million random numbers in R. Try:

```
>system.time(rnorm(1000000))
```

The function `system.time()` takes a *process* as an argument and returns the time that it took to complete the process. In the above example, the *process* is the generation of 1,000,000 random numbers from a standard normal distribution, using the function `rnorm`. You can (and will) use `system.time` in your own work as you come to care about the actual amount of time required to complete a process. However, for the sorts of applications you will be conducting on a regular basis, the amount of time required to complete a process will be very low, e.g:

```
>system.time(sum(1:10000)) #what does the sum here do?
```

R provides handy random number generators that can be used to simulate random draws from almost any probability distribution. If you want to simulate random numbers from a normal distribution, you can use the function `rnorm`, which takes three arguments: the *number* of random numbers you wish to generate, the *mean* of the distribution from which you will draw the numbers, and the *standard deviation* of that distribution.

Create vectors of random numbers with the following specifications and visually inspect histograms of each:

(a) 1000 random numbers from a normal distribution with mean of 5 and standard deviation of 2

(b) 100 random numbers from a normal distribution with mean of 0 and standard deviation of 10

(c) 100 random numbers from a normal distribution with a mean of 0 and a standard deviation of 2

(d) In the vector of length 100 you created for part (b), how many numbers are greater than 2? How about the vector of numbers you created for part (c)?

There is no need to restrict ourselves to the normal distribution: for every simulation problem we might have, there is probably a random number generator ("RNG") that suits our needs. Look at the help pages on the following functions:

```
rbinom
rgamma
rpois
rgeom
rexp
runif
```

This set of RNGs contains many of the functions you will wish to use. Each of them accepts as an argument the number of random numbers to generate as well as the *parameters* of the distribution. In the case of `rnorm(100, mean=2, sd=3)`, the mean and standard deviation are the parameters that determine the distribution from which random numbers are drawn.

Random numbers are not random: with few (questionable) exceptions, random numbers are not *really* random. Rather, the RNG is merely an algorithm that starts out with some initial number and generates a sequence of numbers that behave as if they were random (but they aren't really random). In the context of programming, the RNG always begins with a random **seed**, which is the starting number for the RNG algorithm. If you start the same RNG with the same **seed**, you will recover exactly the same sequence of numbers. Let's look at 5 runs of the uniform random number generator without giving it a seed (choosing 5 random numbers each time):

```
for (i in 1:5){
    print(runif(5));
}
```

Let's compare that to 5 runs with a specified seed (using `set.seed`):

```
for (i in 1:5){
    set.seed(1);
```

```
    print(runif(5));
}
```

Notice anything? In the first case, you didn't specify a seed, so it gave 5 sets of non-identical numbers. But in the second case, you specified the same seed (1) for each run of the uniform RNG, so it produced the same set of numbers.

But wait a minute, you say. Didn't I just tell you that RNGs always require a seed? Yes, I did...and yes, they do. If you don't provide a seed, the RNG is seeded by the *computer time* (well, most RNGs are seeded like this...I'm not positive that R does this. Your computer has an internal clock, and when you type `rnorm(500)` without first providing a seed, R goes deep into your computer processor and pulls out the current time. Some string of digits from this *time* becomes the seed.

If random numbers aren't really random, and if random numbers eventually repeat themselves (they do; all known RNGs are periodic), do we have to worry about correlations between successively drawn random numbers? Probably not: the length of periodicity of RNGs is enormous. One famous RNG, known as the Mersenne Twist, has a period of `2^19937 – 1`. This is a very, very large number. For the purposes of comparison, the number of particles in the universe is somewhere in the vicinity of `2^262`. Nonetheless, the fact that random numbers aren't really random leads many computer scientists to refer to them as *pseudorandom number generators* (or PRNGs).

This stuff is both technical and fascinating, but more on the topic is largely irrelevant: all you need to know is that (i) R provides useful random number generators, and (ii) that these RNGs have been chosen for both the *speed of execution* and their large *length of periodicity*.

Random number exercises:

2) Compare the total elapsed system time in seconds required to produce ten million (yes, 10,000,000) random numbers from the following distributions:

(a) Normal, mean=0, standard deviation = 1
(b) Poisson with lambda = 2

(c) Uniform with min=0, max = 10
(d) Geometric, with prob = 0.5
(e) Binomial with n=1, prob = .5.

If you don't know what these parameters mean (particularly if you haven't taken a good statistics course or probability theory), don't worry too much – just know that they control the shape and location of a distribution of random numbers. For example the uniform distribution has *min* and *max* parameters that determine the upper and lower limits of the distribution (if this isn't clear, you should do some experimenting until you have a solid intuitive understanding of what is going on). Based on your results above:

It is fastest to simulate from the _____ distribution.

It is slowest to simulate from the _____distribution.

3) Does the time required to simulate from a normal distribution depend on the mean of the distribution from which you are simulating?

Start with a seed of 1, and use a **for** loop to generate 5,000,000 random numbers from a normal distribution with mean of 10^0, 10^1, 10^2, 10^3...through 10^6. Print the timings returned by `system.time` to your console with each iteration. Provide your code

4) **`rbinom(n, size = 1, prob= .5)`** simulates coin tossing with a fair coin (the `prob` argument is like '*probability of heads*' in this context). Try a few runs with n = 5, 10, and 20. What are the 1s and 0s? Repeat the exercise with `prob = 0.8` and see whether it clarifies things.

Pseudocode: all programmers write pseudocode. Pseudocode is a structured way of laying out the logic of a program that you wish to write without worrying about all the details of syntax, variable initialization, etc. It is more than just brainstorming: proper pseudocode contains a logically flawless description of the algorithm that you wish to code.

Back to the geometric growth problem: take a look again at #3 from last weeks homework. I asked you to do a discrete-generation simulation of

population growth where the distribution of progeny per individual follows a geometric distribution with parameter p = 0.333 (thus, each individual has 2 progeny on average), and to continue the simulation until the population size exceeds some maximum.

Go back to homework 4 (#3) and re-read it if this isn't clear.

How would we go about writing pseudocode for this problem? My pseudocode would look something like this:

```
WHILE the number of progeny is less than the maximum allowable pop
size:

      DO have each individual give birth (with geometric dist. of
            progeny, where mean # progeny = 2)

      DO Sum all the progeny totals – this is the new population size

      DO append the new population size to the vector of population
            sizes
```

That's really all there would be to that example. And the real code would then look something like this:

```
MAXPOPSIZE <- 5000;
curr_pop_size <- 20;
popvector <- curr_pop_size;
set.seed(1);

while (curr_pop_size < MAXPOPSIZE){
     babies <- rgeom(curr_pop_size, .33333);
     curr_pop_size <- sum(babies);
     popvector <- c(popvector, curr_pop_size);
}
```

Note that when writing pseudocode, you omit things like variable declarations, seed-setting, and so on. These are just details that, though necessary, are not part of the core algorithm. Obviously, you could change the seed or MAXPOPSIZE as much as you wanted, and it might change the outcome of the simulation, but it wouldn't change the way the algorithm works.

<u>Now for a more challenging problem:</u>

5) You are interested in the distribution of times it takes to get a population of MAXPOPSIZE. In homework 4, I had you do 10 manual

simulations and tabulate the generation times for each. Now I want you to automate it, such that you can do *many* trials. In other words, if you did 1000 trials of the above and counted the number of generations (NGENS) required to get to MAXPOPSIZE for each trial, what would it look like if you plotted a histogram of NGENS? This type of simulation project will arise repeatedly as you tackle more biologically relevant problems.

(a) Write pseudocode (or modify the above pseudocode) to deal with this problem (you want to be able to run 1000 simulations, getting a new value of NGENS for each simulation).

(b) ***Only*** when you are absolutely convinced that your pseudocode is logically sound, you can implement your code. Do it piecewise. Start with what you know works. Add complexity one small step at a time. ***Never, ever*** try running large numbers of simulations until you are positive that you can get it to work for one (and then several) runs, or you may never figure out why it isn't working. Make sure you have ample comments in your code.

Do this with MAXPOPSIZE = 10000 (ten thousand), an initial population size of 20, an average of 2 progeny per individual, and set.seed equal to 1 at the very beginning of the simulation.

(c) Plot a histogram of NGENS when you have successfully simulated 1000 episodes of population growth.

(d) Repeat 1000 simulations, but modify it such that the average number of progeny per individual is 3, rather than 2. To have an average of 3 progeny per individual, you have to have a parameter p in `rgeom` of 0.25. Do it again with an average of 4 progeny per individual ($p = 0.20$).

(e) Write everything down in a single file that when sourced, does the following:

- Sets up the simulation (initializations, etc).

- Runs the simulation with average of 2 progeny per individual (1000 replicates)

- Prints, to your display, the following sentence:

```
For progeny = 2, mean number gens = x
```

where x is the mean number of generations to reach MAXPOPSIZE.

- Then: repeats simulation with mean number of progeny = 3 and, at the end, prints out the corresponding message.

- AND repeats yet again with a mean number of progeny = 4, prints out the message, etc.

- And finally, plot histograms of the number of generations for progeny=2, 3, and 4 in the same plot window with the same x axis dimensions so they are comparable (think xlim). Hint: remember 'par' and 'mfrow' and/or 'mfcol'.

The general structure of your code should look something like this:

```
-DO simulation 1
-print something
-DO simulation 2
-print something
-DO simulation 3
-print something
-PLOT results
```

Note that you'll have to have a variable to store the results after each simulation.

This is a real simulation project: think hard about this before you begin coding anything; above all, only do things incrementally. If you try to program this in only a few steps, without doing it as incrementally as possible, it almost certainly will not work – and you will have a very, very difficult time figuring out what is wrong (also, the more you change in a given step, the greater the likelihood of having multiple simultaneous problems!).

If you can do this, you are ready for some real, mind-expanding simulation exercises.

6) Central Limit Theorem. If you draw large numbers of random numbers from some particular probability distribution and sum them, the sums will tend to be normally distributed. This is true regardless of the shape of the underlying distribution, as long as this distribution has a finite variance (which applies to most/all of the distributions we will be working with). This result is known as the Central Limit Theorem (CLT) and is fundamentally important in probability theory and statistics.

In the next few exercises, you will simulate random numbers to verify the CLT.

(a) Generate 1000 pseudorandom numbers from a geometric distribution with parameter $p = 0.25$. Now generate 1000 sums of 10 geometrically-distributed random numbers with p = 0.25. In other words, you should end up with a vector of length 1000, where each element is defined by

```
x[i] <- g1 + g2 + g3 + g4 + ... g10
```

where `g1` through `g10` are just geometric random numbers. Note that you'll have to draw 1000 * 10 geometric random numbers to do this exercise. You do **not** need to use two **for** loops for this (*hint: look at what I did in #7 in homework 4).

Now generate 1000 sums of 1000 geometrically-distributed random numbers, with parameter $p = .25$.

Plot histograms of the 3 sets of random numbers in the same plot window. Can you see evidence of the CLT?

(b) Repeat the exercise above, but sample from a uniform distribution with a min of -1 and a max of 5 (again comparing the distribution of uniformly distributed numbers against sums of 10 and 1000 uniform random numbers.

7) Consider the binomial random number generator (`rbinom`). The binomial distribution has two parameters: ***size*** and ***prob***. The binomial distribution can be thought of as the 'coin flipping distribution', because it is easily used to simulate things like coin-flipping experiments.

Consider an experiment where you have two possible outcomes: *success* and *failure*. Or *heads* and *tails*. Or, with respect to evolutionary fitness, *to be* or *not to be*. `rbinom` generates a random number corresponding to the number of successes that occur in ***size*** trials, where the probability of success on each trial is ***prob***.

Consider the following:

`rbinom(1, size=1, p=0.5)` simulates a coin toss experiment with probability of heads $p = 0.5$. Look at `rbinom(100, size=1, p=0.5)`: this is a sequence of 100 coin-tossing experiments.

Now: if you set **size =5**, rbinom returns the number of successes (heads) in 5 coin flips. So `rbinom(3, size =5, prob = 0.5)` simulates 3 random numbers, where each random number is the number of heads that occurred in five coin flips with equal probabilities of heads and tails.

Write out the meaning of the following expressions, in terms of coin toss experiments:

(a) `rbinom(1000, size=1, prob=0.7)`

(b) `rbinom(1, size=14, prob=0.5)` . Also – what are the largest and smallest numbers that could be returned by this? No using your computer here!

(c) `sum(rbinom(1000, size=1, prob=0.5))` . What is this number? What is the largest value you could obtain?

(d) **challenge**: Re-write the expression from part (c) such that it returns a number with exactly the same distribution as `sum(rbinom(1000, size=1, prob=0.5))` , but do not use the *sum* function. Hint: this can

and should be done with at least five fewer characters (e.g., `A, B, C, +, })` than the expression in part (c).

8. I am a rabid, compulsive gambler. I have a coin that I'd like to use in some of the ever-popular coin-toss gambling that occurs in front of the north entrance to VLSB.

I have had a string of bad luck, and I am concerned that my coin is unfair and unjust. In the past 200 flips of my coin, I have observed a total of 139 heads.

I want to know whether my coin is fair (e.g., equal probabilities of heads and tails). Devise an experiment involving simulations from the binomial distribution that will allow you to assess whether my coin is fair.

This is an example of a **Monte Carlo** method in statistics. Here we have a *null hypothesis*: the coin is fair, with equal probabilities of heads and tails. We have a *test statistic*: the number of heads in 200 trials. Now we need to simulate the ***null distribution*** of the test statistic. In other words, if the null hypothesis is true, what should the distribution of the test statistic look like? The only way to get at this is to do large numbers of simulation experiments under the null hypothesis and examine the outcomes.

What is the probability of obtaining a test statistic value as large or larger than that which we observe (139 of 200) ? This is not as hard as it seems: this probability can be estimated simply as the frequency of simulated values as large or larger than our observed test statistic.

Think about this. Your answer should consist of the following parts:

(a) a brief statement of the logic behind the test. Here are some questions to help guide your thinking:

What do we mean by "experiment"?

How many total "experiments" will you do?

What are you going to count/tabulate for each experiment?

(b) Pseudocode required to perform the test

(c) the actual code that performs the test (with the null distribution determined by 5000 simulations of the test statistic). Upon completion, your code should print to your console something like this:

```
Probability of the data under the null hypothesis: Z
```

where Z is the relevant probability.