BioEE 758
Week 6 handout/homework

Topics for week 6

- Pseudocode

- Troubleshooting

- Conditional execution: **if**, **else if**, **else**

A general programming scheme

Good programming practice involves following a logical sequence of steps.

1. State the problem: what are you trying to do?

2. Define inputs, outputs, and relevant parameters. What goes into the program? What parameters do you need to specify? What will come out of the program? A vector? A matrix? A dataframe? Random text?

3. Design the algorithm. You've already defined what will go into and come out of your program. This step involves writing the logically perfect pseudocode 'model' of the program you will create.

4. Identify modular components of algorithm. Suppose you are writing code to do a large number of simulations, such that your pseudocode looks something like this:

```
For each simulation
    initialize variables
    Do simulation
    store results
```

In this case, the variable initialization + Do simulation + store results is a modular block of code: you can run code this bit and execute it without having to do the full set of n-thousand replicate runs.

5. Program modular components. Program!

6. Test program....

5. Turn more modular components into R statements

6. Test program....

.

.

.

etc etc

Keep adding components and testing code until you have implemented the full algorithm. Test everything together. One of the big advantages of **functions** (which we will study in detail next week) is that they make it very easy to make your code modular. If, for example, you were programming a robot to prepare and serve a multi-course meal, you might end up with something like this:

```
while (guests_are_hungry == TRUE)
    soup <- makeSoup();
    salad <- makeSalad();
    meat <- grillHorseflank();
    drinks <- tapWater();
    serve(soup, salad, meat, drinks);
```

Here *makeSoup*, *makeSalad*, *grillHorseflank*, *tapWater*, and *serve* are functions, and each can be tested individually and independently of all other components of the code. This code is thus **modular**. If you were writing code like this, you would write each of the core functions individually, testing them to ensure that each worked before moving on to anything else.

Last week some of us began working on a program to simulate a random walk, or Brownian motion. Here is the problem: every full moon in the progressive hamlet of Zzyzx (Mojave district, California), the town drunk – Brownie – goes on a bender (or 'gets on the piss', as the say in Oz). When Brownie is sufficiently intoxicated, she stumbles to the main drag and staggers aimlessly up and down the street (Zzyzx is something of a one-horse town).

Each time unit *t*, Brownie takes a small step. These step sizes can be modeled as random draws from a normal distribution with a mean of 0 and std. dev. of 1. Thus, suppose Brownie takes steps of the following sizes for her first six staggers:

```
-1, 0.3, 1.2, -0.2, -0.3, 0.1
```

If her initial position is at x = 0, then at time t = 1, she will be at position
```
x = 0-1 = -1
```
And at time t=2, she'll be at
```
x = -1 + 0.3 = -0.7
```
At t=3, she'll be at
```
x = -0.7 + 1.2 = 0.5
```
and at t=4,
```
x = 0.5 – 0.2 = 0.3
```

and so on. See where this is going? After each time unit *t,* Brownie's position is simply her position at time *t – 1* plus some step size (where the step size in this case is a single random draw from a normal distribution with mean of 1 and std. dev. of 0).

As director of Zzyzx Search and Rescue, you are tired of wasting precious resources looking for Brownie every full moon. Brownie is also proprietor of the popular local business *Earth Mother Auto Towing and Aromatherapy,* and you clearly cannot afford to lose this cherished citizen to the dingoes that lurk in the yuccas beyond the edge of town. One concerned citizen suggested that you develop a stochastic model to determine how far Brownie might walk in a some amount of time, given some assumptions about her average stagger. Given such a model, you could easily narrow down your search radius.

Going through this stepwise:

1. State the problem: we wish to simulate the location of Brownie after t time units.

2. Define inputs, outputs, and relevant parameters.

Input: We'll assume that Brownie starts her wander at position x = 0.
Output: Brownie's position at time *t*.

Parameters: Brownie's step per unit time is normally distributed (mean = 0 sd = 1).

3. Design the algorithm. Here's what my pseudocode would look like:

```
Set initial position of Brownie to x = 0
For each time step
     Choose step size for Brownie from normal distribution
     Get new position by adding step size to current position
```

Note that this is just the pseudocode to simulate her position after t time units. What if you wanted a record of where she was at each point in time? You might use a vector where each index is her position at some point in time. So, if `position` is the vector, `position[10]` would be Brownie's position after 10 time steps.

```
Set initial position of Brownie to x = 0
For each time step
     Choose step size for Brownie from normal distribution
     Get new position by adding step size to current position
     Append new location to vector that records locations
```

4. Identify modular components of algorithm. There isn't really anything modular here. However, if you wanted to repeat the Brownie simulation 1000 times, then the pseudocode above would be a modular component nested within a larger simulation.

5 and 6 -Here is an example of code that simulates Brownie's walk:

```
TIME_STEPS <- 20;
INITIAL_POSITION <- 0;

location <- INITIAL_POSITION;
for (i in 1:TIME_STEPS)
{
    step <- rnorm(1, mean=0, sd=1);
    location <- location + step;
    cat('step', i, 'step', step, 'loc', location, '\n', sep=' ');
}
```

The cat() statement is just in there for error checking purposes. It lets me verify several things: (i) the loop is running over all the iterations that I want it to run; (ii) it chooses a step size that makes sense; and (iii) the trajectory of locations makes sense. What do I mean by "makes sense"? Clearly, *if the step size was the same for every iteration,* or *if the location did not change,* then we might suspect that our code was problematic.

1) I gave you two examples of pseudocode above. The simulation code in steps 5 & 6 corresponds to which example?

A digression on error-checking, initializations, and declarations.

On error checking: questions to ask yourself every time you program

1) How much output should I be getting? If I am doing 10 simulations and storing the number of generations in a growing vector, I can check to make sure that my result vector actually contains __________ (fill in the blank!).

2) this is a stochastic simulation, so I expect to observe some variation in results for different runs. Do I observe this?

3) Is my program entering the necessary loops in each simulation?

4) Is my program beginning each simulation with the necessary parameter initializations? Be very, very careful here!

5) Is my program actually iterating through all the (10 or 50 or 5000) simulations that I think it is doing? Or maybe it is just doing i = 1 or i = 5000 and skipping the rest?

6) Do I have any garbage values in my output that indicate problems elsewhere in my code? As an example, in #8 in the homework, if you observe any simulations of 200 coin flips with a fair coin where you observe 0 heads or 200 heads, it is possible that there is a problem. A common explanation for these sorts of patterns is that your code does something like this:

```
x <- rep(0, 100)
for (i in 1:1000)
        x[ i ] <- DO_SIMULATION;
```

Now...because you initialized x as a vector of 0s, if DO_SIMULATION fails, you might find that x is still storing worthless 0s that are not results of a simulation - it still contains the initial values. But if you fail to catch this, you will think that your simulation gave you 0 for some runs.

When I'm building code, I always run and re-run my code with lots of print and cat statements to check outputs, inputs, and values of variables at different stages. I delete these or comment them out when I am convinced that my code works, but it is a good habit to get into. We'll do much with this on Wednesday. I require you to have these error-checking statements (at least commented out, so that I can see that you have been using them) for many homework problems this week.

Declarations

Declarations pertain to variables that are used in your code which remain *constant*. Thus, you set the values initially, but those values are never modified throughout the code. Usually, you declare constants at the beginning of your code, before you provide any other statements to be executed.

For example, in the Brownian motion code, I used two constants: TIME_STEPS and INITIAL_POSITION. It is a good idea to keep constants (and *only* constants) in caps when they appear in your code, because you can easily locate them by visual inspection if you need to. Perhaps the most common constant you will declare is NUM_SIMS (or NREPS) or whatever...this being the number of simulations you wish to do. Once you have declared it, you simply do things like

```
for (i in 1:NUM_SIMS)
```

when you wish to repeat the simulation NUM_SIMS times.

Initializations:

Initializing variables is very important (and tricky in some cases).

2) Write pseudocode for a Brownian motion simulator that will generate the distribution of Brownie's positions after t = 50 time steps. By *generate a* *distribution,* I mean that your code should repeat the simulation 5000 times and record her position at the end of each simulation.

In your pseudocode, you must include variable initializations. I want to see where in your code you are initializing variables. I have no way of enforcing this, but it is **unacceptable** to do this by trial and error. You should be able to do this with pencil and paper and type it up when you are done.

GIGO

Charles Babbage was a mid-19th century mathematician and engineer who conceived the idea of the programmable computer. He developed several complicated mechanical devices that could be programmed to compute logarithms and polynomials, and – while these would be unrecognizable as anything resembling a modern computer – they fulfilled the basic criteria of the computing machine, permitting basic iteration, formatted output, and so on.

Babbage wrote in his autobiography:

“**On two occasions I have been asked, – "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question**.”

This brings us to a central tenet of computer programming: the output from your programs will only be as good as the input provided. This principle is known as Garbage In, Garbage Out (or **GIGO**). Your program may appear to run perfectly well, but your output may be utter garbage. I’ll say it again: *the fact that your program appears to run without evoking an error message gives you no indication whatsoever that the program has successfully performed any particular task.* Fail to properly initialize variables? Expect GIGO. Etc.

Conditional Execution

Finally.

if and related expressions provide a means to allow your code to diverge. Intuitively, **if /else if** statements are straightforward:

```
boyname <- c('casper', 'francis', 'bart');
girlnames <- c('brandi', 'misti', 'barbie');

while(baby does not have name){
     if (baby is male){
          choose random name from boyname;
     }else if (baby is female){
          choose random name from girlnames
     }else{
          doTalkShowCircuit();
          write book;
          make money;
     }
}
```

There is no requirement that you use **if / else if / else** together. You use these expressions three ways:

i) **If** statements only.

```
if (x == 'a')
     doSomething1;
if (x == 'b')
     doSomething2;
if (x=='c')
      doSomething3;
```

ii) **If** and **else** statements: this assumes that there are only two possibilities, and they must be mutually exclusive.

```
if(sick){
     cat('hey! sicko!\n');
}else{
     cat('not sick\n');
}
```

iii) **If**, **else if**, and **else** statements:

```
if (x == 'a'){
     doSomething1;
}else if (x == 'b'){
     doSomething2;
}else if(x == 'c'){
.
.
.
.
}else if (x == 'z'){
     doSomething26;
}
else{
     cat('x != a letter\n')
}
```

if statements can be nested:

```
if (statement1) {
     if (statement2){
          if (statement3){
               doSomething()
          }
     }

}
```

Nested **if** statements are contained entirely within another **if** statement.

Mutually exclusive options should use the if...else if....else construct. For example, suppose we want our program to execute a different statement for each of the conditions x < 0; x == 0; and x > 0. We might write:

```
if (x < 0){
      statement1();
}else if (x == 0){
      statement2();
}else{
      statement3();
}
```

3) confusing conditionals:

You have a list x, with components *color*, *bird*, *event*, and *food*. You don't know anything about the components – not even possible values – but you subject it to the following worthless test:

```
if (x$color == 'blue'){
    if (x$bird == 'hammerkop'){
        if (x$event == 'marathon') {
            if (x$food == 'hamhock'){
                cat('yum');
            }
        }
    }else if (x$bird == 'pigeon'){
        cat('not quite a loon...')
        if (x$event == 'marathon')
            while(1)
                print('I love birds.')
    }else{
        cat('ostrich???')
    }

}else{
    cat('green');
}
```

Answer the following:

a) What are possible or impossible values for *color*, *bird*, *event*, and *food* if- upon sourcing the code – you observe only "yum" printed to your screen?

b) what if you observe only "ostrich"?

c) Is it possible to observe only "I love birds" as output?

d) Of the 5 expressions that result in output, which are mutually exclusive? E.g., can you run the code and observe both 'green' and 'ostrich???' printed to your screen?

4) Wegmans has installed **Robocop**, a friendly, automated ID-checking robot to deal with the constant stream of delinquent and underage youth who repeatedly attempt to purchase alcoholic beverages. Here is part of the pseudocode for **Robocop**:

```
while Wegmans is open
    if someone is trying to buy alcohol
        Card them
        If they are younger than 21
            Tell them to scram
        If they are older than 21
            Thank them for their compliance
            Allow them to purchase
    else
        look for people trying to buy alcohol
```

When does **Robocop** fail?

5) Write a loop that runs from i = 1 to i = 25; if i is divisible by 2, print “even” to the display. Otherwise, print “odd”.

6) Create vector *x* of random numbers, where `x <- rnorm(10)`, generated with `set.seed(1)`. Loop over the vector and consider each element in turn: if x[i] is less than or equal to -0.5, cat something like this to screen:

“-0.7 is less than -0.5!”

and else if the number is between -0.5 and 0.5, display this:

“0.3 is in the middle!”

otherwise, print

“1.7. Whopper!”

substituting, of course, the actual x[i] for the numbers above.

7) Write code that prints to screen the numbers from 1 to 100 on separate lines. However, for multiples of three, print "*Fizz*" instead of the number, and print "*Buzz*" if the number is divisible by five. If the number is divisible by both 3 and 5, print "*FizzBuzz*" *on the same line*. You are limited to **three** conditional expressions (whatever combination of **if**, **if else**, and **else** you may desire).

8) Here we will write a function that takes two numbers x and y as arguments and returns the result of a mathematical operation. The function will evaluate arguments (x, y) as follows:

```
result  = x + y ( when x >= 0 and y >= 0)
result   = x + y^2 (when x >=0 and y < 0)
result   = x^2 + y (when x < 0 and y >=0)
result = x^2 + y^2 (when x < 0 and y < 0)
```

The general structure of the function is

```
myFxn <- function(x, y)
{
    if (condition)
        res <- do something;
    if (condition)
        res<- do something;
    if (condition)
        res <- do something;
    if (condition)
        res <- do something;
    return(res);
}
```

Fill in the relevant bits for *do something* and *condition*. Evaluate the function for the following x, y pairs and manually check your results: (0, 1), (1, 1), (-1, -10), (-5, 3), (3, -5). Although we haven't really covered this, all you have to do to apply `mfFxn` to numbers like x = 1 and y = 100 is `myFxn(x=1, y=100)`.

9) Go back to our exercise on geometric population growth. We will now assume that the population experiences "good" and "bad" growth years. During good years, each individual in the population averages 3 progeny (geometric parameter $p = 0.25$). During bad years, each individual averages just a single progeny (parameter $p = 0.5$). It just so happens that the environment in which our hypothetical population lives experiences alternating good and bad years: when the number of generations is odd, the population experiences a good year. When the number of generations is even, the population experiences a bad year.

a) develop code that simulates population growth in this fashion, using conditional execution within a while loop to specify how progeny are to be chosen depending on the current generation. You may need to use the modulo operator (%%; also known as 'remainder' operator). Assume a starting population size of n = 10 individuals and a MAXPOPSIZE of n = 10000.

b) get the distribution of 5000 such population growth events (this part should be easy by now!) and plot in a histogram.

---

10) Thus far, on a scale of 1:10, my enjoyment of this homework assignment has been:

(a) median(rgeom(5000, p = 0.8))

(b) somewhat better than a root canal *sans* novacaine

(c) my delight was such that i can only express it as
`pleasure <- 1/0;`

(d) Well...something like this:

```
my_priorities<-function()
{
    cat('oops...\n\n');
    myHomework;
}
my_priorities();
```

The next two problems are <u>challenging</u>. In principle, they are quite similar to problems you've already seen, but I have not provided any template code. You need to build these from scratch, following all of the steps I've outlined above. All may involve some combination of **while** loops, **for** loops, and/or **if-if else-else** statements. For 11A and 11B, I want you to submit – in order – the following:

a) statement of the problem you are trying to solve/model

b) what are your outputs and inputs? What variables/values do you feed into the code? What do you expect to get out of the program (e.g., "a vector of length equal to the number of simulations; each element of the vector is the number of hotdogs consumed per simulation").

c) Detailed pseudocode with proper indentation etc. You must email me with your pseudocode plus *a* and *b* above when you get to this point (I will respond quickly with comments...). I am more concerned whether you do this correctly than I am about the actual code.

d) Your code, with detailed error checking bits where relevant (you can comment all of these out, but I want to see that error-checking statements are present).

11) A model for genetic drift! This one is biologically relevant (finally!). You are studying the population genetics of a particular locus you suspect to be involved in *judicial activism*. Before you test whether the locus might be the target of purifying selection (perhaps *purging* is more appropriate here), you want to model the expected dynamics of alleles through time at the locus assuming <u>no selection or other deterministic processes</u>. You are essentially building a *null model* to describe the dynamics of alleles when genetic drift is the only evolutionary process in operation.

You have a population of 20 diploid, sexual organisms and a locus with two alleles: 1 and 2 (or "a" and "b"). Thus, at any particular locus, there are a total of 40 alleles in the population. Each generation, the population undergoes random mating, you allow only 20 randomly chosen progeny to survive (thus the number of individuals and alleles remains constant over time). We will ignore all the details of mating etc: for our purposes, we assume that we choose the population of alleles for the nth generation by sampling with replacement from the pool of alleles in the n-1th generation.

So: if we had 6 alleles in the population at generation 0 (3 a and 3 b alleles), we would form the 1st generation by randomly drawing 6 alleles from this population of 6 alleles. We might thus end up – after 1 generation – with 1 a and 5 b alleles. Now: how would we go about drawing alleles for the 2nd generation?

You will find the function "sample" very useful here. Suppose a population consisted of 1 b and 5 a alleles. Then you could represent the population as a vector:

```
pop <- c('a', 'a', 'a', 'a', 'a', 'b');

or pop <- c(rep('a', 5), 'b');
```

Then

```
sample(pop, 6, replace=TRUE)
```

returns a random sample of 6 alleles from pop, sampling with replacement. Try it! Of course, you can specify pop however you want: if you have a population starting with 30 "dog" alleles and 45 "skunk" alleles, you could start with

```
pop- c(rep('skunk', 45), rep('dog', 30));
```

and you would simulate the population after 1 generation of random mating and drift as

```
sample(pop, length(pop), replace=TRUE)
```

***Note: *sample* rocks. Learn it well! What will happen to allele frequencies over time if you use `replace = FALSE`?

a) develop code that simulates genetic drift for 50 generations given equal initial allele frequencies and a constant pop size of 20 individuals (40 alleles). You should record the frequency of one of the alleles each generation (note that since the frequencies must sum to one, you can easily calculate the other frequency if you have one...).

b) develop code that generates the distribution of waiting times – in number of generations – to fixation of one or the other allele. How many generations on average does it take for the population to become monomorphic for one allele or the other? How does this relate (approximately) to the initial population size? In this case, we don't care about the frequency every generation – just the number of generations until one allele disappears completely.

For #12 below, your assignment this week is only to do items a, b, and c as specified above: state problem, identify inputs/outputs, and write pseudocode. However, completion of the actual coding is optional for this week (but I want to see your pseudocode before you do any work on this!).

12) Bouncing Brownie: in the Brownian motion example described above, we allowed the town drunk to take a random walk, with no constraints on how far she could stray from town. Even though Brownie is taking very small steps each unit of time, it is certainly possible that she could end up tens or hundreds or even thousands of miles from home given a sufficiently lengthy drunken ramble.

We are now going to model a somewhat more complicated version of Brownie's random walk. There are two parts to this problem:

a) We now seek to model Brownie in two dimensions. Thus, each time unit, Brownie shifts her position along both the east-west (= *x*) axis and the north-south (= *y*) axis. Each time unit, she takes a step along the x axis, where the step size is drawn from a normal distribution with mean = 0 and std dev = 1. For consistency, let us do like statisticians would do and represent Brownie's step as ***step ~ N(0, 1)***. This is the conventional way of saying *step is normally distributed with mean=0, sd = 1*. Likewise, Brownie takes a step on the y-axis, where the step is also *N(0,1)*. Provide code that creates vectors of Brownie's x and y coordinates. Let her begin her random walk at (0, 0) and allow the staggering to continue for 100 and 1000 time steps.

When you plot your coordinates, you can simply do

```
plot(y.coord ~ x.coord, type = 'l')
```

and all successive steps will be connected by a line. This should look really cool, and if you do it again, the pattern should be different.

b) Now: Brownie is stuck in a room that is merely length 10 and width 10. In practical terms, this means that you should assume that Brownie can only bounce around between y = (0, 10) and x = (0, 10). Your challenge is to model Brownie's movements within the room if each step is distributed as *N(0, 3).*

You need to find a way to ensure that Brownie never leaves her "room". One solution is to use a *reflecting boundary*. Suppose Brownie takes a step that brings her outside the edge of her room. Since the "room" is only defined for x = (0, 10) and y=(0,10), we could imagine her current position as x = 1, y = 9.7. Suppose she then takes steps along the x and y axis of -0.5 and 1.4, respectively.

Thus, her new position would be x = 1 – 0.5 = 0.5, and her new y position would be y = 9.7 + 1.4 = 11.1. But wait a minute! The room ends at y = 10! To implement a reflecting boundary solution, you'll need to use a series of conditional expressions. For example, in the above example, you might want to put an if statement somewhere that would essentially do the following:

```
If the new y value exceeds the upper room boundary (10)
     - bounce back into room by a distance equal to the
     amount that you are "out" of the room
```

In the above example, we have overstepped the boundary by 1.1 (since our y position is 11.1). So, we would bounce back 1.1 units into the room, and our new y position would be simply *y = room.max – distance.of.overstep,* or *y = 10 – 1.1 = 8.9.*

You can assume that Brownie enters her room through a door at position x = 0, y = 1. Plot the bounce of Brownie for 100 and 1000 time steps. Your output here is not the plot *per se*, but the numbers required to generate the plot. Consider drawing the room out.

c) CHALLENGE. In her drunken stupor, Brownie accidentally locks herself into the room. Fortunately, there is a portal to Middle Earth between positions x = 10, y= 8.5 and x = 10, y= 10. For Brownie to enter to portal, all she has to do is stagger into the wall between positions (10, 8.5) and (10, 10). If Brownie enters the room at x = 0, y=1, how long does it take (how many time steps) for Brownie to exit the room? Do this with set.seed = 1. If you really feel ambitious, count the number of times Brownie hits the wall before successfully exiting the room.

hint: you'll have to think hard about whether (or when) your exit conditions might conflict with the *reflecting boundary* conditions you've specified above.

Again, drawing this out will probably be helpful. Feel free to produce a plot of your results at the end.