STAT375: Statistical Aspects of Competition **Programming in R**

Lecturer: Dr. McShane **Week 3** **Scribe(s): Matthew Kaneb and Damien Ruparel**

This week's focus was programming in R. Building on the time we spent in Week 2 familiarizing ourselves with the game of Qwixx, this week we worked on coding a simulation of the game. First, we took a look at how one might code a simulation of Blackjack, giving us a basic framework for coding board and card games in R.

Outside of class we were tasked with reading Hands on Programming in R (2014), a guide to navigating R's environment system, write your own functions, and use all of R's programming tools. Key takeaways from the reading are outlined below.

Key Takeaways from Hands-on Programming in R (2014)

Objects - Objects provide a structured approach to programming. By defining a dataset as a custom object, a developer can easily create multiple similar objects and modify existing objects within a program. In R, objects can store data using vectors, lists, matrices, arrays, and data frames. Data frames are the most used among data scientists, since they store data in a tabular fashion. We can modify our stored data through using the `<-` operator, which is useful in cleaning smaller data sets. However, with larger datasets we can use logical tests and boolean operators to modify the values.

- A perfect example of using objects is when creating a die. `die <- c(1, 2, 3, 4, 5, 6)`. In this case, `die` is our object, and it is storing an atomic vector which is grouping some sort of data values together with `c`.
- Another common example of using R Objects is to either store arrays or matrices. Matrices store values in a 2-dimensional array, and we can use the `matrix` function to store organize our atomic vector into a matrix; for example: `m <- matrix(die, nrow = 2)`, where the `nrow` argument tells R how many rows to include in the matrix.

Functions - Functions are self-contained modules of code that accomplish a specific task. Functions usually take in data, process it, and return a result. Once a function is written, it can be used over and over and over again. Functions are an easy way to abstract away repeatable intermediate processes in your code. Using functions makes code more clear for a 3rd party as well as for the person programming.

- An example of a function is `sample(x, size)` which takes two arguments `x`, the elements to choose from, `size`, the size of the sample, and returns a random sample of size `size` made up of elements from `x`.

Environments & Scope - R keeps multiple environments; the largest being the global environment that keeps objects that are called in the command line. R will create smaller environments which contain subsets of functions. Objects in smaller environments are not accessible from the global environment. Once a function has completed its commands and returned its results, the intermediate objects used within that function goes away.

Loops and Vectorization - Loops are used to complete repetitive tasks a certain number of times, in the case of a `for` loop, or until a certain condition is met, in the case of a `while` loop. Loops are great however they can be computationally expensive when a number of complex processes are repeated over and over. R is designed for vectorization of these processes, meaning that the complex processes can be performed simultaneously for all of the items in a dataset as opposed to looping over each of them individually.

- Calculating Expected Value with a Loop: Multiply each probability with each element and then add each of them up. This requires N separate iterations, where N is how many outcomes there are.
- Calculating Expected Value with Vectorization: Take the dot product of the probability vector and the elements vector. This requires 1 calculation and is much faster than the iterative approach.

Larger Projects - Larger projects can be tricky to manage. Remembering what everything does and making sure every edge case is accounted for can be hard. Using the following tactics can help:

- Use `#` to add comments to your code. Writing what each chunk of code is intended to perform is incredibly helpful for larger projects. This will help you quickly remember where you left off the last time you were working on the project and will allow you and other members of your team to be on the same page.
- Break projects into many intermediate steps. Large projects can be daunting but if you map out what are the intermediate steps that make up this larger problem and write functions for each one of them, the project seems much more manageable.
- Test and debug often. Writing large amounts of code in between testing will make it especially tricky to identify where your bugs are coming from. Testing more often mitigates this problem.

Coding a Qwixx Simulation

Scribe duos spent the week coding their own simulations of the game with the following guidelines.

- Don't use global scope – game states should be passed from function to function.
- All functions should be broken into small, repeatable pieces.
- Include a `playRound` function that takes in both game boards, an indicator for which player is the active player, and a strategy for each person.
- The strategy should not be directly tied to `playRound` function, and should take ANY strategy that produces legal moves (i.e., two dice/ one die/zero dice + penalty placed by the active player, one die/zero dice placed by the passive player).
- Include a function that checks whether a move is legal given a board state, locked row indicators, whether someone is active roller or passive roller, and if someone is active roller, whether the secret dice move can be played AFTER the public dice move.
- All of your code should be in a single R script, and should be call-able by a single function which takes on player 1 strategy and player 2 strategy and outputs, at minimum, a vector with player 1 and player 2 scores as well as the final game board for both players).
- Include the following Default Strategy:
  - If no possible moves, default to "penalty"
  - Place an X on the leftmost legal box. If there are ties, draw a color at random.
  - Never place two Xs in the same turn.
  - Only place Xs while active roller.

Upon completing their games, duos met to discuss how they solved certain problems and what they did differently. Seeing how different groups structured their simulations and laid out their code was insightful in thinking about how you'd attack a similar project in the future.

At the end of the week, one program was selected to be the version all of the duos would eventually write their own strategies in. Later in the course, duos will compete head-to-head against the rest of the class.

Duo 6's Qwixx Program

Running Simulations

- `simulateGame(players, strats, debug)` : This function is used to simulate Qwixx Games
  - `players` : However many players you'd like to simulate playing a game of Qwixx, which should be set to `2`
  - `strats` : What strategy each player will be playing with
  - `debug` : If `TRUE`, puts the simulation in DEBUG mode.
- `play(p, g, q, strats, db, active, debug)` : This function manages which strategy each player plays with.
  - `p` : player number
  - `g` : dictionary of player info
  - `q` : dice from this roll
  - `strats` : What strategy each player will be playing with
  - `db` : Storage space for player strategies
  - `active` : `TRUE` if player `p` is the active roller
  - `debug` : `TRUE` if in DEBUG mode

```
[1] "==========================PLAYER 2 TURN=========================="
[1] "Qwixx Roll was 1 3 4 6 5 1"
[1] "==========================Possible Moves========================="
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,]    0    0    0    0    0    0    0    0    0     0     0     0     0
[2,]    0    0    0    0    0    0    0    0    1     0     0     0     0
[3,]    0    0    0    0    0    0    0    0    0     0     0     0     0
[4,]    0    0    0    0    0    0    0    0    0     0     0     1     0
[1] "================================================================="
[1] "Player 1 Moves: NULL"
[1] "Player 2 Moves: c(2, 9)"
[1] "==========================PLAYER 1 BOARD========================="
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,]    0    1    1    0    0    0    1    0    1     1     0     0     0
[2,]    0    0    0    0    0    0    0    1    1     1     0     0     0
[3,]    0    0    0    0    1    1    1    1    0     1     0     0     0
[4,]    0    0    1    0    1    1    1    1    1     0     0     0     0
[1] "==========================PLAYER 2 BOARD========================="
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,]    0    1    0    0    0    0    1    1    1     0     0     0     0
[2,]    0    0    0    0    0    0    1    1    1     0     0     0     0
[3,]    0    1    0    1    1    1    1    1    0     1     1     0     0
[4,]    0    1    0    0    0    0    1    1    0     1     1     0     0
```

Figure 1: Example turn feedback when `debug = TRUE`

Writing New Strategies

- See `DefaultStrategy()` for an example of a strategy
- Functions that might be helpful:
  - `findRightXs()` : Find furthest right `X` in each row given a board
  - `possibleMoves()` : Find possible moves given a board and dice roll
  - `allowed()` : Determine if an `X` is allowed to be placed in a given location
- **Other Notes for New Strategies**
  - Boards are 4×13. Ignore the first column. This convention is easier than constantly subtracting one from all coordinates
  - Throughout the code we tried our best to be consistent and only use `[Row, Column]` as opposed to `[Row, Number on the Board]`.
  - The `simulateGame()` function takes care of most of the rules of the game so in your strategies all you have to do is return the coordinates of the `X`s you'd like to mark for this turn.
  - For any helper functions you want to use in your strategy, add the following prefix to avoid having functions of the same name : `teamX_helperFunction()` where `X` is your team number.
  - Name your strategy `teamXStrategy()` where `X` is your team number.

Review Questions

1. When working at the command line, which environment is the active environment?
    a. Local Environment
    b. Global Environment
    c. Natural Environment
    d. None of the Above
2. What kind of R object would you store a sequence of elements of potentially different types?
    a. `matrix`
    b. `vector`
    c. `list`
    d. `array`
3. **Tic-Tac-Toe Logic**: Given a 3×3 matrix where `0` means empty, `1` means X, and `2` means O, write a function that determines if player one (Xs) or player two (Os) has won the game, assuming the game is over. Use these test cases to verify your function.

```
test1 = matrix(c(1,1,1,1,2,2,2,1,2), nrow = 3, ncol = 3)
test2 = matrix(c(2,1,1,2,2,1,0,2,1), nrow = 3, ncol = 3)
test3 = matrix(c(2,1,0,1,2,0,0,1,2), nrow = 3, ncol = 3)
```

4. **Components of Monopoly**: Break the game Monopoly down into its intermediate steps in a similar way to what we did with Blackjack and Qwixx.
    - List the kinds of data we'd want to keep track of for each player.
    - List some of the intermediate processes for which we'd want to write functions.

Review Question Answers

1. *b*
2. *c*
3. **Tic-Tac-Toe Logic Answer**: We included two example approaches below.

A Vectorized Approach

```
checkWinnerVec <- function(board){

  # Create vector of the columns, rows, and diagonals
  diag1 = c(board[1,1],board[2,2],board[3,3])
  diag2 = c(board[1,3],board[2,2],board[3,1])
  lines = list(board[1,],board[2,],board[3,],board[,1],board[,2],board[,3],diag1,diag2)

  # Check each line to see if either player got 3 in a row
  result = lapply(lines, function(x) if (x[1] == x[2] & x[1] == x[3] & x[2] == x[3]) x[1] else 0)

  return(Reduce("+",result))
}
```

An Iterative Approach

```
checkWinnerIt <- function(board){

  # Check rows and columns
  for(x in 1:3){
    if(board[x,1] == board[x,2] & board[x,1] == board[x,3] & board[x,2] == board[x,3]){
      return(board[x,1])
    } else if (board[1,x] == board[2,x] & board[1,x] == board[3,x] & board[2,x] == board[3,x]){
      return(board[1,x])
    }
  }

  if(board[1,1] == board[2,2] & board[1,1] == board[3,3] & board[2,2] == board[3,3])
    return(board[1,1])

  if(board[1,3] == board[2,2] & board[1,3] == board[3,1] & board[2,2] == board[3,1])
    return(board[1,1])
}
```

A Speed Comparison

```
reps = 10000; results = 1:reps
start <- proc.time()
for (i in results) {checkWinnerVec(test1); checkWinnerVec(test2); checkWinnerIt(test3)}
proc.time() - start
```

```
##    user  system elapsed
##    0.81    0.00    0.87
```

```
start <- proc.time()
for (i in results) {checkWinnerIt(test1); checkWinnerIt(test2); checkWinnerIt(test3)}
proc.time() - start
```

```
##    user  system elapsed
##    0.13    0.00    0.13
```

> *While* `checkWinnerVec` *implements an* `apply` *family function (* `lapply` *),* `checkWinnerIt` *is actually faster, since it doesn't create any new objects and exits as soon as it finds a winning row/column/diagonal. Additionally, since there are only three iterations in the* `for` *loop in* `checkWinnerIt` *, little overhead is created/used. — Dr. McShane*

4. **Components of Monopoly Answer**

- List the kinds of data we'd want to keep track of for each player.
  - Money
  - Properties/Houses
  - Location on the board
- List some of the intermediate processes for which we'd want to write functions.
  - Rolling Dice / Moving on the board
  - Buying Properties
  - Buying Houses
  - Paying Taxes
  - Passing GO

References

Grolemund, G. (2014), *Hands-on Programming with* R.