

Math 152 Lab Assignment I

25th January 2007

This lab will introduce the computer software R. R has many capabilities but we'll start with some basic ones: arithmetic, storing and naming results, summarizing, and generating so-called uniform random numbers.

1 Dice

Suppose we roll a pair of dice twenty times, and get the following:

red	2	5	3	5	3	3	2	3	4	3	2	6	3	5	6	1	2	6	4	1
blue	4	1	5	4	4	1	6	1	3	4	2	3	4	6	3	3	5	1	6	5

We would like to know first the frequencies of 1, 2, 3, 4, 5, 6 for each die. We first need to have the data available in R to use. We can use the “c” function to make a list of the results for each die, and name these lists “red” and “blue” as follows:

```
> red <- c(2,5,3,5,3,3,2,3,4,3,2,6,3,5,6,1,2,6,4,1)
> blue <- c(4,1,5,4,4,1,6,1,3,4,2,3,4,6,3,3,5,1,6,5)
```

In R a list where each entry is a number is called a vector. Some useful functions to check that you didn't make a typo are

```
> length(red)
[1] 20
> length(blue)
[1] 20
> min(red)
[1] 1
```

```

> max(red)
[1] 6
> unique(red)
[1] 2 5 3 4 6 1
> sort(red)
[1] 1 1 2 2 2 2 3 3 3 3 3 3 4 4 5 5 5 6 6 6
> table(red)
red
1 2 3 4 5 6
2 4 6 2 3 3

```

We can use “table” to count the different outcomes for the pair, as well, like

```

> table(red,blue)
      blue
red 1 2 3 4 5 6
  1 0 0 1 0 1 0
  2 0 1 0 1 1 1
  3 2 0 0 3 1 0
  4 0 0 1 0 0 1
  5 1 0 0 1 0 1
  6 1 0 2 0 0 0

```

This means that in our 20 rolls, we got 3 on the red die and 4 on the blue die 3 times. On the other hand we got a 4 on the red die and 3 on the blue die only once.

We can give this table a name, say dice, by

```

> dice <- table(red,blue)

```

Then if we want to know the individual entries, we can get them like

```

> dice[3,4]
[1] 3

```

This tells us what’s in the 3rd row and 4th column, that is, how many times in our 20 rolls we got a 3 on the red die and a 4 on the blue die.

2 Sets

We can work with sets in R. R represents a set as a vector (list). So to make the set $\Omega = \{1, 2, 3, 4, 5, 6\}$ we could use the “c” function and write

```
omega <- c(1,2,3,4,5,6)
```

We can define subsets $E = \{1, 2\}$ and $F = \{1, 3\}$ by

```
E <- c(1,2)
F <- c(1,3)
```

The basic set operations union, intersection, and difference are

```
> union(E,F)
[1] 1 2 3
> intersect(E,F)
[1] 1
> setdiff(E,F)
[1] 2
> setdiff(F,E)
[1] 3
> setdiff(omega, E)
[1] 3 4 5 6
```

The last set is $\Omega \setminus E$ which is also known as the *complement* of E relative to Ω .

3 Data input

It’s difficult and error-prone to type data into R (or any computer system; special techniques are used for this when it’s especially important). There are two ways to get around this: use data someone else has already made available electronically, or generate it within R.

3.1 Generating dice data

R has a built-in random-number generator. It generates random numbers between 0 and 1, each with equal likelihood. You use it like this:

```
x <- runif(100)
```

to get a vector of 100 random numbers between 0 and 1 named x.

If we want to have the computer roll dice, we can divide the numbers between 0 and 1 into six equal-sized intervals, then generate uniform random numbers and see which interval they fall into. Here's a quick way to do that in R:

```
> x <- ceiling(6*runif(100))
```

What we're doing here is multiplying by 6 and then rounding up to the next integer. A bit more straightforward is to use the "cut" function, like:

```
> x <- cut(runif(100),breaks=6, labels=FALSE)
```

Type "help(cut)" in R to see information about "cut". Instead of "breaks=6" we could have told "cut" what the breaks should be. This will be useful later.

4 Monty Hall

Use R to simulate the Monty Hall problem. For one game, you need

1. to place the prize randomly behind one of the three doors. So you need a random number in $\{1, 2, 3\}$. Call this "prize".
2. to have the contestant choose a door. You might just want to have the contestant always choose door 1, or you could have the contestant choose randomly, in which case you need another random number in $\{1, 2, 3\}$. In either case call this "choice"
3. Then Monty needs to select a door other than "prize" or "choice". You may want Monty to make a random choice when necessary, in which case you may need a random number in $\{1, 2\}$. It does no harm to generate it even if you wind up not needing it. Or you could have Monty choose the lowest-numbered door which is not "prize" or "choice". In either case you have to figure out which doors are available for Monty to choose. There are several ways to do this; the most natural is probably to create the set of available doors and then take the smallest or a random choice:

```

doorsleft <- setdiff(c(1,2,3),union(prize,choice))
door <- min(doorsleft)
# to choose a random door do
# n <- length(doorsleft)
# r <- ceiling(n*runif(1))
# door <- doorsleft[r]

```

Now you need to work out the logic for each strategy (stay or switch) like this:

```

# stay
if (prize == choice) win <- TRUE else win <- FALSE
# switch
secondchoice <- setdiff(c(1,2,3),union(choice,door))
win <- (prize == secondchoice) # shorter form

```

Now let's put this all together into a function which will play one round for us. Start like this

```

> play <- function(strategy) {
+ prize <- ceiling(3*runif(1))
+ choice <- 1
+ doorsleft <- setdiff(c(1,2,3),union(prize,choice))
+ door <- min(doorsleft)
+ secondchoice <- setdiff(c(1,2,3),union(door,choice))
+ if (strategy == "stay") {
+   win <- (prize == choice) }
+ if (strategy == "switch") {
+   win <- (prize == secondchoice) }
+ return(win)
+ }

```

This is the simplest version: the contestant always chooses door 1 and Monty always chooses the lowest-numbered eligible door when a choice is necessary.

To run this, type

```

> play("stay")
[1] TRUE

```

Playing repeatedly To simulate many plays, we can run “play” over and over. To do so we can use a loop. We can also count the number of wins. These two lines are all we need:

```
wins <- 0
for (i in 1:1000) wins <- wins + play(“stay”)
```

This runs “play(“stay”)” 1000 times, and each time adds the result (TRUE is treated as 1 and FALSE as 0 here) to the current number of wins to get the updated number of wins. When I ran this I got 337 wins, which is pretty close to 1/3 or 1000. Run these two lines again with “switch” in place of “stay” and see what you get.

Modifications We can modify this function to allow us to specify the contestant’s choice rather than have it built in. Just remove the line

```
choice <- 1
```

and modify the first line to

```
play <- function(strategy,choice=1)
```

Then if you use the function as before it will work as before (because “choice = 1” is the default) but you can also try

```
> play(“stay”,choice=2)
```

to play having the contestant choose door 2, or

```
> play(“stay”,choice=ceiling(3*runif(1)))
```

to have the contestant choose randomly.

5 So what’s the assignment?

- Develop your own Monty Hall Problem simulator starting with the function we put together above.
- Modify it to allow various options for play (how Monty chooses, number of plays, anything else you think necessary to be realistic).

- Give the results for “stay” and “switch” for 10, 100, 1000, 10,000 and 100,000 plays.
- When you’re satisfied your function works correctly, print it out and annotate it carefully. By that I mean write an explanation of what it’s doing, line by line, and why that’s a correct simulation of the Monty Hall Problem.
- Hand that in to me Tuesday.