R for 2.4-2.6

A.S. Wagaman, CC BY-NC

R for 2.4 - Conditioning and the LoTP

This section contains a script to accompany Example 2.13, and code to execute a Knuth shuffle to accompany Example 2.14.

Example 2.13 describes the "secretary problem" phrased as just picking the largest number out of a set of n numbers. You are presented with n pieces of paper (with n different numbers written on them), and you get to look at them one a time. Once you look at a number, you can either keep it (can only do once) or reject it and move on to the next number. When you receive a new number, all you know is how it compares to the previous numbers in terms of size. Your goal is to find the largest of the n numbers, with only these options. This is explained in the text.

The suggested strategy (and the strategy that is analyzed) is to pick some fixed "r" $<$ n, and for the first r numbers, reject them all but record the maximum. Then, out of the remaining picks, choose the first number that is larger than that maximum. The value of r can be adjusted, but the text finds the optimal r is n/e (highest probability of correctly choosing the overall maximum) for this strategy. So, how often will you get the overall maximum with this strategy? Let's find out.

SCRIPT - TopNumber.R - pg. 55

```
## TopNumber.R
##
## Numbers are 1, ... , n
n <- 100  # Top number
r <- round(n/exp(1)) # r = n/e = 37 for n=100
ntrials <- 10000
simlist <- vector(length = ntrials)
for (j in 1:ntrials){ #start of outer for loop
  envlist <- sample(1:n, n)
  best <- which(envlist == n) # position of largest number
  prob <- 0
  firstmax <- max(envlist[1:r])  # maximum of the first r numbers
  for (i in (r+1):n){ #start of inner for loop  # start looking after the r-th
    if (envlist[i] > firstmax) { #start ifelse statement
      if (envlist[i] == n) prob <- 1
        break }
    else {prob<-0} #end ifelse statement
  } #end of inner for loop
 simlist[j] <- prob
 } #end of outer for loop
 mean(simlist)
```

```
## [1] 0.3705
```

As you review the script, note several new coding concepts - there are two *for* loops here, with one nested within the other. You may also note the *break* option in the ifelse statement. Basically, there are options such as *break* and *next* to help govern how loops behave. *break* breaks out of the current loop and gives control to the first statement outside the inner-most loop, whereas *next* stops the current iteration and advances

the loop (see help for more detail). See if you can identify what each line does here based on the strategy description in the text.

Example 2.14 discusses uniformly random permutations. The code provided implements a Knuth shuffle to obtain such a permutation.

```
# Simulating Random Permutations - pg. 56
# Implements a Knuth shuffle
n <- 12 #set number of items to permute
perm <- 1:n
for (i in 1:(n-1)) {
  x <- sample (i:n, 1)
  old <- perm[i]
  perm[i]<- perm[x]
  perm[x]<-old
}
perm
```

```
##  [1]  5  6 12  8  2  7  3 11 10  4  9  1
```

Each time you run this, you will obtain a different permutation. This code could easily be rewritten as a function for ease of executing it multiple times, as shown below.

```
# Supplemental
Knuthperm <- function(n){ #start function declaration
  perm <- 1:n
  for (i in 1:(n-1)) { # start of for loop
    x <- sample(i:n, 1)
    old <- perm[i]
    perm[i]<- perm[x]
    perm[x]<-old
  } # end of for loop
perm #output
}
```

Running the entire chunk above will give you a function for generating a random permutation, where you just choose the input, n. We can set a seed for reproducibility when running this several times.

```
set.seed(42)
Knuthperm(12)
```

```
##  [1] 11 12  5  1 10  9  4  8  7  2  6  3
```

```
Knuthperm(12)
```

```
##  [1]  9 12  5  8  2  3  7 10 11  6  1  4
```

R for 2.5 - Bayes Formula

This section contains a script to demonstrate a simulation for the scenario of three coins described on pages 59-60 using Bayesian statistics.

SCRIPT - Bayes.R - pg. 60

```
### Bayes.R
###
```

```
### A coin is picked at random. It is either fair, 2-headed, or 2-tailed.
### The coin is flipped and comes up heads.
### What's the probability the coin is 2-headed?
###

set.seed(612) #added for reproducibility
n <- 50000
ctr <- 0
data <- c(0, 0, 0)  # Stores number of times coin is fair, 2-h, 2-t, resp.

while (ctr < n){
    coin <- sample(c(1, 2, 3), 1) ### Pick a coin at random
    p <- c(.5, 1, 0)[coin]   ### p = Prob(Heads) for the coin picked
    cointoss <- sample(0:1, size = 1, prob = c(1-p,p))
    ## Flip coin with 1-p - P(Tails), p = P     (Heads)
    ## cointoss = 1 for heads, 0 for tails
    if (cointoss == 1) ### Check if heads
                       ### If not, skip through and flip again
                       ### If yes, keep track of which coin was tossed
    {
    data[coin] <- data[coin]+1
    ctr <- ctr + 1
    }
}

### Simulated result
Coin <- c("Fair", "2-H", "2-T")
data.frame(Coin, data/n)
```

```
##   Coin  data.n
## 1 Fair 0.33478
## 2  2-H 0.66522
## 3  2-T 0.00000
```

This script uses another while loop to deal with the condition that we know a head must be observed.

The new aspect of this script is how the output is displayed. The vector "data" (which is probably not a name you want to use when writing your own functions) contains three values, the number of times each coin appeared when a heads was the result of the flip. However, it's easy to forget which order the values are in. To fix this, a label vector called Coin is created, and the label and vector of counts (data) are put together into a data.frame object in order to be displayed.

Textbook Citation

Dobrow, R. P. (2013). Probability: with applications and R. John Wiley & Sons.