HW: Week 2

36-350 – Statistical Computing

Week 2 – Spring 2021

Name: Cherie Hua

Andrew ID: cxhua

You must submit **your own** homework as a PDF file on Gradescope.

Question 1

(10 points)

Write a `for()` loop, that loops up to 1000 times, that samples a single datum from a standard normal. In order, in the loop, sample the single datum using `rnorm()`, utilize `next` if that datum is less than 1, then print the index of the datum (via `cat()` with a following line break, i.e., "\n"), and last utilize `break` if the datum is greater than 2. (The last digit shown should be very much less than 1000. The expected value of the number of times we would sample from a standard normal before observing a value greater than 2 is given by `1/p = 1/(1-pnorm(2))`, which is about 44.)

```r
#samples from a normal distribution and checks the value
for (i in 1:1000) {
  r = rnorm(1)
  if (r < 1) {
    next
  } else if (1 < r && r < 2) {
    cat(i, "\n")
  } else if (r > 2) {
    cat(i, "\n")
    break
  }
}
```

```
## 17
## 24
## 26
## 27
## 40
```

Question 2

(10 points)

Write a `while()` loop that simulates the flipping of eight coins and loops until all eight coins are heads or all eight coins are tails, at which point you should break out of the loop. Define a counter that increments by one every time through the loop, and print the value of that counter after you break out of it. For full credit, utilize an `if` construct with a logical `or`. (Hint: `rbinom(1,8,0.5)` returns the number of successes when flipping a fair coin eight times.)

```
#flips 8 coins and checks how many iterations it takes for all
#8 coins to be tails or all 8 coins to be heads
coins = 0
i = 0
while (coins < 8) {
  i = i + 1
  successes = rbinom(1, 8, 0.5)
  if (successes == 8 || successes == 0) {
    print(i)
    break
  }
}
```

```
## [1] 174
```

Question 3

(20 points)

A rite of passage for every parent is informing his or her three-year-old child that “Despite how happy you were to beat me, winning at Chutes and Ladders doesn’t actually require any skill.” Here, you will write a `while()` loop that simulates a single game of Chutes and Ladders. First, see the image of the game board at http://vignette4.wikia.nocookie.net/uncyclopedia/images/d/d8/Chutes%26Ladders1.gif/revision/latest?cb=20090809223310. Here are the rules, in case you’ve forgotten or never played: you start at 0 (off the board) and spin a spinner that randomly selects an integer between 1 and 6, inclusive. (You can think of this as rolling a fair six-sided die. Use `sample()`.) Move your piece that number of spaces. If you end up at the bottom of a ladder, you move to the top of the ladder (so if you are at 0 and spin a 4, you end up at 14); if you end up at the top of a chute, you move to the bottom of the chute. (Beware space 87!) The game ends when you reach 100 exactly (so, for instance, if you are at 97 and spin a 4, you do not move… but if you spin a 1, you move to 98 and slide down the chute to 78). (Note that *you are the only one playing* the game: it’s the world’s most boring version of Solitaire.) (Hint: after each spin and move, use `which()` to determine which element of, e.g., the vector `ladder.bottom` equals your new position. Call the output of `which()` “`w`”. If the length of `w` is greater than zero, then you are at the bottom of a ladder, so you’d reset your position to the corresponding top of the ladder: `ladder.top[w]`. You’d repeat this process, checking if you are at the top of a chute. Hint II: remember… if your new position after a spin is greater than 100, you don’t move!)

```
#a game of chutes and ladders (single-player)
ladder.bottom = c(4, 10, 13, 15, 20, 24)
ladder.top = c(40, 80, 70, 69, 31, 46)
chute.bottom = c(11, 8, 22, 31, 28, 7)
chute.top = c(99, 32, 45, 73, 58, 51)

location = 0 #the player's position on the board
i = 0 #keep track of iterations
while (location < 100) {
  spin = sample(1:6, 1)
```

```r
  location = location + spin
  w = which(ladder.bottom == location)
  if (length(w) > 0) location = ladder.top[w]
  w = which(chute.top == location)
  if (length(w) > 0) location = chute.bottom[w]
  i = i + 1
}

print(i)
```

```
## [1] 12
```

Question 4

(10 points)

Create a function `char2int()` that returns the position of the input character value in the English alphabet. For instance, "A" returns 1 and "z" returns 26. Make use of `R`'s built-in constants (see, e.g., `?Constants`), and perform checks on the input: return `NULL` if the input is not of type `character`, if the input string contains more than one character, or if the input string contains non-alphabetical characters. (Also, limit the length of the input vector of strings to one.) Comments are optional. Test your function with inputs 1, "abc", "2", c("a","b"), and "m". Only the last input should yield a non-NULL output. **NOTE**: do not use the `match()` function anywhere in your function! It would do the job of `char2int()` trivially, but the idea here is to build your own function that mimics what `match()` is doing.

```r
char2int = function(x)
{
  if (is.character(x) == FALSE || nchar(x) > 1 || length(x) > 1 || length(grep(x, 0:9)) > 0)   {
    return(NULL)
  }
   return(which(letters == x))
}


char2int(1)
```

```
## NULL
```

```r
char2int("abc")
```

```
## NULL
```

```r
char2int("2")
```

```
## NULL
```

```r
char2int(c("a", "b"))
```

```
## NULL
```

```r
char2int("m")
```

```
## [1] 13
```

Question 5

(10 points)

Improve your recursive factorial function from Lab 2 to include a check that the input value is either zero or a positive integer. Have the function return `NULL` otherwise. Compute 5!, 0!, -5!, and "a"!. If in the last case you get a thrown exception rather than an output value of `NULL`, think about the sequence of checks you need to do here. (Possibly useful information: $0! = 1$.)

```r
my_factorial = function(x)
  #factorial function with extra conditions to catch edge cases
{
  if (is.numeric(x) == FALSE || round(x) != x || x < 0) return(NULL)
  if (x==1 || x == 0) return(1)
  return(x*my_factorial(x-1))
}

my_factorial(5)
```

```
## [1] 120
```

```r
my_factorial(0)
```

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

```r
my_factorial(-5)
```

```
## NULL
```

```r
my_factorial("a")
```

```
## NULL
```

Question 6

(20 points)

The Babylonians devised the following algorithm to compute $\sqrt{x}$, using only basic arithmetic operations. First one guesses the root r; for instance, $r = x/2$. Then, either $r^2 > x$ or $r^2 < x$. (Unless we are dealing with perfect squares, $r^2 = x$ is just not gonna happen.) We replace r with the average of r and x/r and repeat. Write a function `root()` that takes as input x and a tolerance t that dictates when you've converged to a solution, i.e., when $|r^2 - x| \leq t$ you stop. Make 10^{-6} the default value for t. Your `root()` function should output a list with two elements: `sqrt.x`, the value of r at convergence; and `n.iter`, the number of iterations required to achieve convergence. Display results for $x = 11$, 101, and 1001, and compare those results to those achieved by using the `sqrt()` function (by computing the absolute difference between `sqrt.x` and the output of `sqrt()`). How many iterations were needed to achieve convergence in each case? (Answer this by displaying the number of iterations; you don't need to add text to your answer.) Note: you need not add comments here if you choose not to. Just remember: in real life, you should always add comments so that others can understand your code.

```r
root = function(x, t = 10^{-6})
{
  n.iter = 0
  r=x/2
  while (abs(r^2 - x) > t) {
    n.iter = n.iter + 1
    r = (r + x/r)/2
  }
  return(list(sqrt.x = r, n.iter = n.iter))
}
```

```r
out = root(11)
abs(sqrt(11) - out[[1]])
```

```
## [1] 1.327219e-09
```

```r
out[[2]]
```

```
## [1] 4
```

```r
out = root(101)
abs(sqrt(101) - out[[1]])
```

```
## [1] 1.232578e-10
```

```r
out[[2]]
```

```
## [1] 6
```

```r
out = root(1001)
abs(sqrt(1001) - out[[1]])
```

```
## [1] 5.329071e-13
```

```r
out[[2]]
```

```
## [1] 8
```