# STATS506_PS2_code

## Problem 1

### Task (a)

Before we start, we need to define a random number generator to control the randomization in all 4 versions of `play_dice` function.

```
#' Function to create a random number generator function
#'
#' @param seed a numeric value
#' @return a function that receives n as input and generates a n-dimensional random vector
myRng <- function(seed) {
  set.seed(seed)
  return(function(n) {
    sample(1:6, n, replace = TRUE)
  })
}
```

Then, we try to implement 4 versions of `play_dice`function.

```
#' Version 1: Implement this game using a loop over the die rolls.
#'
#' @param num_rolls an integer indicating the number of rolls
#' @param rng a function as a random number generator
#' @return total net revenue of the dice-rolls
play_dice_v1 <- function(num_rolls, rng) {

  # Generate the rolls using given RNG
  rolls <- rng(num_rolls)

  # Iterate over `num_rolls` to calculate total revenue
  revenue <- 0
```

```r
  for (i in 1:num_rolls) {
    if (rolls[i] %in% c(2, 4, 6)) {
      revenue <- revenue + rolls[i]
    }
  }

  # Calculate net income using net_income = revenue - cost
  cost <- 2*num_rolls
  return(revenue-cost)
}
```

```r
#' Version 2: Implement this game using built-in R vectorized functions.
#'
#' @param num_rolls an integer indicating the number of rolls
#' @param rng a function as a random number generator
#' @return total net revenue of the dice-rolls
play_dice_v2 <- function(num_rolls, rng) {

  # Generate the rolls using given RNG
  rolls <- rng(num_rolls)

  # Calculate revenue using R vectorized function
  revenue <- sum(2*(rolls == 2)+ 4*(rolls == 4) + 6*(rolls == 6))

  # Calculate net income using net_income = revenue - cost
  cost <- 2*num_rolls
  return(revenue-cost)
}
```

```r
#' Version 3: Implement this by collapsing the die rolls into a single table().
#'
#' @param num_rolls an integer indicating the number of rolls
#' @param rng a function as a random number generator
#' @return total net revenue of the dice-rolls
play_dice_v3 <- function(num_rolls, rng) {

  # Generate the rolls using given RNG
  rolls <- rng(num_rolls)

  # Collapse the die rolls into a single table
  roll_counts <- table(rolls)
```

```r
  # Iterate over items in the table
  # Notice that the length of the table can be 6 at most. So this is not computationally c
  revenue <- 0
  for (i in names(roll_counts)){
    if (as.numeric(i) == 2){
      revenue <- revenue + 2*roll_counts[as.character(i)]
      next
    }
    if (as.numeric(i) == 4){
      revenue <- revenue + 4*roll_counts[as.character(i)]
      next
    }
    if (as.numeric(i) == 6){
      revenue <- revenue + 6*roll_counts[as.character(i)]
    }
  }
  revenue <- as.numeric(revenue)

  # Calculate net income using net_income = revenue - cost
  cost <- 2*num_rolls
  return(revenue-cost)
}
```

```r
#' Version 4: Implement this game by using one of the "apply" functions.
#'
#' @param num_rolls an integer indicating the number of rolls
#' @param rng a function as a random number generator
#' @return total net revenue of the dice-rolls
play_dice_v4 <- function(num_rolls, rng) {

  # Generate the rolls using given RNG
  rolls <- rng(num_rolls)

  # Use "apply" to calculate the revenue
  # To use "apply", we choose to convert vector "rolls" into a matrix
  revenue <- sum(apply(matrix(rolls, ncol = num_rolls), 2, function(row) {
    if (row %in% c(2,4,6)){
      return(row)
    } else{
      return(0)
    }
```

```
  }))

  # Calculate net income using net_income = revenue - cost
  cost <- 2*num_rolls
  return(revenue-cost)
}
```

### Task (b)

In this task, we will show that all versions work. Notice that we will pass a random number into the function `myRng`, since we do not need to fix the result at this moment.

```
for (t in c(3,3000)) {
  cat("Result for ", t, " using v1 is ", play_dice_v1(t, myRng(sample.int(1000, 1))), "\n"
  cat("Result for ", t, " using v2 is ", play_dice_v2(t, myRng(sample.int(1000, 1))), "\n"
  cat("Result for ", t, " using v3 is ", play_dice_v3(t, myRng(sample.int(1000, 1))), "\n"
  cat("Result for ", t, " using v4 is ", play_dice_v4(t, myRng(sample.int(1000, 1))), "\n"
}
```

```
Result for  3  using v1 is  -2
Result for  3  using v2 is  -2
Result for  3  using v3 is  -4
Result for  3  using v4 is  0
Result for  3000  using v1 is  -64
Result for  3000  using v2 is  256
Result for  3000  using v3 is  -42
Result for  3000  using v4 is  -106
```

### Task (c)

In this task, we will show that the four versions give the same result. We will control the randomization by putting the same seed to RNG.

```
seed <- 114
for (t in c(3,3000)) {
  cat("Result for ", t, " using v1 is ", play_dice_v1(t, myRng(seed)), "\n")
  cat("Result for ", t, " using v2 is ", play_dice_v2(t, myRng(seed)), "\n")
  cat("Result for ", t, " using v3 is ", play_dice_v3(t, myRng(seed)), "\n")
  cat("Result for ", t, " using v4 is ", play_dice_v4(t, myRng(seed)), "\n")
}
```

```
Result for  3  using v1 is  2
Result for  3  using v2 is  2
Result for  3  using v3 is  2
Result for  3  using v4 is  2
Result for  3000  using v1 is  -146
Result for  3000  using v2 is  -146
Result for  3000  using v3 is  -146
Result for  3000  using v4 is  -146
```

It is clear that the four versions give the same result.

```r
library(microbenchmark)
```

```r
# Benchmarking with low input (100)
seed <- 896
benchmark_low <- microbenchmark(
  v1 = play_dice_v1(100, myRng(seed)),
  v2 = play_dice_v2(100, myRng(seed)),
  v3 = play_dice_v3(100, myRng(seed)),
  v4 = play_dice_v4(100, myRng(seed)),
  times = 100
)

# Benchmarking with large input (10000)
benchmark_large <- microbenchmark(
  v1 = play_dice_v1(10000, myRng(seed)),
  v2 = play_dice_v2(10000, myRng(seed)),
  v3 = play_dice_v3(10000, myRng(seed)),
  v4 = play_dice_v4(10000, myRng(seed)),
  times = 100
)
```

```r
print(benchmark_low)
```

```
Unit: microseconds
 expr     min       lq      mean   median       uq      max neval
   v1  70.300  75.3515  83.31498  79.3010  83.5015  183.101   100
   v2  11.201  12.1510  31.14295  12.9015  14.6510 1556.801   100
   v3  68.500  75.6010  88.60103  82.3005  91.3510  223.500   100
   v4 146.401 155.3010 168.69605 160.5510 172.3510  290.702   100
```

```r
print(benchmark_large)
```

```
Unit: microseconds
 expr       min         lq       mean    median         uq       max neval
   v1  6622.201  7386.9010  9060.4390  8993.302 10192.0005 20111.801   100
   v2   486.101   514.0510   548.4701   537.152   568.5520   771.001   100
   v3   721.801   827.9515   876.0450   866.551   913.8005  1220.700   100
   v4 13538.702 15225.9010 17494.0810 17490.751 18986.8010 29273.600   100
```

**Problem 2**

```r
data <- read.csv("./cars.csv")
data <- data.frame(data)
```

```r
colnames(data) <- c("height", "length", "width", "driveLine", "engineType", "isHybrid", "n
```

```r
data <- data[which(data$fuelType == "Gasoline"),]
```

```r
M1 <- lm(highwayMPG~horsepower+numGears+cityMPG+torque, data = data)
summary(M1)
```

```
Call:
lm(formula = highwayMPG ~ horsepower + numGears + cityMPG + torque,
    data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-12.956  -1.029  -0.118   0.968 196.002

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.381273   0.573349  -2.409    0.016 *
horsepower   0.012339   0.001699   7.264 4.39e-13 ***
numGears     0.418666   0.065757   6.367 2.12e-10 ***
cityMPG      1.281144   0.019826  64.620  < 2e-16 ***
torque      -0.008215   0.001685  -4.877 1.12e-06 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.325 on 4586 degrees of freedom
Multiple R-squared:  0.6965,    Adjusted R-squared:  0.6963
F-statistic:  2632 on 4 and 4586 DF,  p-value: < 2.2e-16
```

```
#library(emmeans)
```

```
#M2 <- lm(highwayMPG~horsepower*torque, data = data)
#interact_plot(M2, pred = horsepower, modx = torque)
```

```
#library(imager)
```

```
#file_path <- system.file("./q2.jpg",package='imager')
#im <- load.image("./q2.jpg")
#plot(1:10,ty="n")
#rasterImage(im,2,1,10,10)
```