Problem Set 2

Brynn Woolley

Problem 1 - Modified Random walk

Consider a 1-dimensional random walk with the following rules:

1. Start at 0.
2. At each step, move +1 or -1 with 50/50 probability.
3. If +1 is chosen, 5% of the time move +10 instead.
4. If -1 is chosen, 20% of the time move -3 instead.
5. Repeat steps 2-4 (n) times.

(Note that if the +10 is chosen, it's not +1 then +10, it is just +10.)

Write a function to determine the end position of this random walk.

The input and output should be:
* Input: The number of steps
* Output: The final position of the walk

```
> random_walk(10)
 [1]  4

> random_walk(10)
 [1]  -11
```

We're going to implement this in different ways and compare them.

1. Implement the random walk in these three versions:
    - Version 1: using a loop.

- Version 2: using built-in R vectorized functions. (Using no loops.) (Hint: Does the order of steps matter?)

- Version 3: Implement the random walk using one of the "`apply`" functions.

```r
# Helper Function
random_walk_fixed <- function(n, draws = NULL) {
  if (is.null(draws)) {
    draws <- runif(2 * n)
  }

  pos <- 0
  for (i in 1:n) {
    dir_draw <- draws[(2 * i - 1)]
    size_draw <- draws[(2 * i)]

    if (dir_draw < 0.5) {
      pos <- pos + ifelse(size_draw < 0.95, 1, 10)
    } else {
      pos <- pos + ifelse(size_draw < 0.80, -1, -3)
    }
  }
  pos
}


# Method 1
random_walk1 <- function(n, draws) {
  pos <- 0
  for (i in 1:n) {
    dir_draw <- draws[(2 * i - 1)]
    size_draw <- draws[(2 * i)]
    if (dir_draw < 0.5) {
      pos <- pos + ifelse(size_draw < 0.95, 1, 10)
    } else {
      pos <- pos + ifelse(size_draw < 0.80, -1, -3)
    }
  }
  pos
}
```

```r
# Method 2
random_walk2 <- function(n, draws) {
  dirs <- draws[seq(1, 2 * n, by = 2)]
  sizes <- draws[seq(2, 2 * n, by = 2)]
  steps <- ifelse(
    dirs < 0.5,
    ifelse(sizes < 0.95, 1, 10),
    ifelse(sizes < 0.80, -1, -3)
  )
  sum(steps)
}

# Method 3
random_walk3 <- function(n, draws) {
  sum(sapply(1:n, function(i) {
    dir_draw <- draws[(2 * i - 1)]
    size_draw <- draws[(2 * i)]
    if (dir_draw < 0.5) {
      ifelse(size_draw < 0.95, 1, 10)
    } else {
      ifelse(size_draw < 0.80, -1, -3)
    }
  }))
}
```

Demonstrate that all versions work by running the following:

```
random_walk1(10)\
random_walk2(10)\
random_walk3(10)\
random_walk1(1000)\
random_walk2(1000)\
random_walk3(1000)
```

2. Demonstrate that the three versions can give the same result. Show this for both `n=10` and `n=1000`. (You will need to add a way to control the randomization.)

```r
set.seed(123)
draws <- runif(2 * 10)
random_walk1(10, draws)
```

```
[1] 7
```

```
random_walk2(10, draws)
```

```
[1] 7
```

```
random_walk3(10, draws)
```

```
[1] 7
```

```
set.seed(123)
draws <- runif(2 * 1000)
random_walk1(1000, draws)
```

```
[1] 78
```

```
random_walk2(1000, draws)
```

```
[1] 78
```

```
random_walk3(1000, draws)
```

```
[1] 78
```

3. Use the microbenchmark package to clearly demonstrate the speed of the implementations. Compare performance with a low input (1,000) and a large input (100,000). Discuss the results.

```
library(microbenchmark)

# low input: 1,000 steps
set.seed(123)
draws_1k <- runif(2 * 1000)
bench_1k <- microbenchmark(
  loop = random_walk1(1000, draws_1k),
  vectorized = random_walk2(1000, draws_1k),
  apply = random_walk3(1000, draws_1k),
  times = 100
)
print(bench_1k)
```

```
Unit: microseconds
       expr    min     lq     mean  median      uq    max neval
       loop 1668.7 1842.1 2066.049 1886.50 2007.45 6622.1   100
 vectorized  180.4  202.9  241.351  222.05  254.25  442.0   100
      apply 2824.0 2993.9 3420.793 3111.40 3269.95 9757.8   100
```

```r
# large input: 100,000 steps
set.seed(123)
draws_100k <- runif(2 * 100000)
bench_100k <- microbenchmark(
  loop = random_walk1(100000, draws_100k),
  vectorized = random_walk2(100000, draws_100k),
  apply = random_walk3(100000, draws_100k),
  times = 10
)
print(bench_100k)
```

```
Unit: milliseconds
       expr      min       lq      mean   median       uq      max neval
       loop 191.7225 249.1976 277.00601 273.2877 289.2033 356.0243    10
 vectorized  13.3627  17.0843  17.55125  17.6831  19.0742  22.1090    10
      apply 392.1293 445.2162 491.57884 464.8003 486.8995 718.5217    10
```

Discussion:

The benchmarking results show that the **vectorized implementation is the fastest**. For 1,000 steps, the vectorized method ran about 10 times faster than the loop and more than 10 times faster than the apply method. For 100,000 steps, the vectorized method was about 15 times faster than the loop and roughly 25 times faster than apply. The loop implementation performed reasonably well but was consistently slower than vectorized. The apply implementation was the slowest due to the overhead of repeated function calls.

In summary:

- Vectorized (fastest, most efficient for large n)

- Loop (moderate performance, simpler to read)

- Apply (slowest, avoid for performance)

4. What is the probability that the random walk ends at 0 if the number of steps is 10? 100? 1000? Defend your answers with evidence based upon a Monte Carlo simulation.

```r
set.seed(123)

# Monte Carlo simulation
simulate_prob_zero <- function(n, sims = 100000) {
  zeros <- replicate(sims, {
    draws <- runif(2 * n)
    random_walk2(n, draws)
  })
  mean(zeros == 0)
}

# Run for different step sizes
prob_10   <- simulate_prob_zero(10)
prob_100  <- simulate_prob_zero(100)
prob_1000 <- simulate_prob_zero(1000)

prob_10
```

```
[1] 0.1323
```

```r
prob_100
```

```
[1] 0.01921
```

```r
prob_1000
```

```
[1] 0.00575
```

Problem 2 - Mean of Mixture of Distributions

The number of cars passing an intersection is a classic example of a Poisson distribution. At a particular intersection, Poisson is an appropriate distribution most of the time, but during rush hours (hours of 8am and 5pm) the distribution is really normally distributed with a much higher mean.

Using a Monte Carlo simulation, estimate the average number of cars that pass an intersection per day under the following assumptions:

- From midnight until 7 AM, the distribution of cars per hour is Poisson with mean 1.