Lab 5: Simulating Discrete Probability Models

San Diego State University - STAT550

Aiden Jajo

Task 1: Roulette wheel simulation

A roulette wheel has 38 slots of which 18 are red, 18 are black, and 2 are green. If a ball spun on to the wheel stops on the color a player bets, the player wins. Consider a player betting on red. Winning streaks follow a Geometric($p = 20/38$) distribution in which we look for the number of red spins in a row until the first black or green. Use the derivation of the Geometric distribution from the Bernoulli distribution to simulate the game. Namely, generate Bernoulli($p = 20/38$) random variates (0 = red; 1 = black or green) until a black or green occurs.

Code set-up

A `while` loop allows us to count the number of spins until a loss. If we use indicator variable `lose` to note a win (1) or loss (0), the syntax is "while we have not lost (i.e., `lose==0`), keep spinning." Once you win, the while loop ends and the variable `streak` has counted the number of spins. Try running a few times.

```
streak = 0
lose = 0
p = 20/38

# While loop continues until we lose (land on black or green)
while(lose==0){
  lose = (runif(1) < p) # generate Bernoulli with probability p
  streak = streak + 1   # tally streak
}

# Display streak length
streak
```

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

The problem

The code chunk above performs the experiment once: spin the roulette wheel until you lose and record the number of spins. Simulate 1000 experiments. As usual, do this by wrapping the code chunk above within a for-loop and storing the number of spins `streak` in a vector.

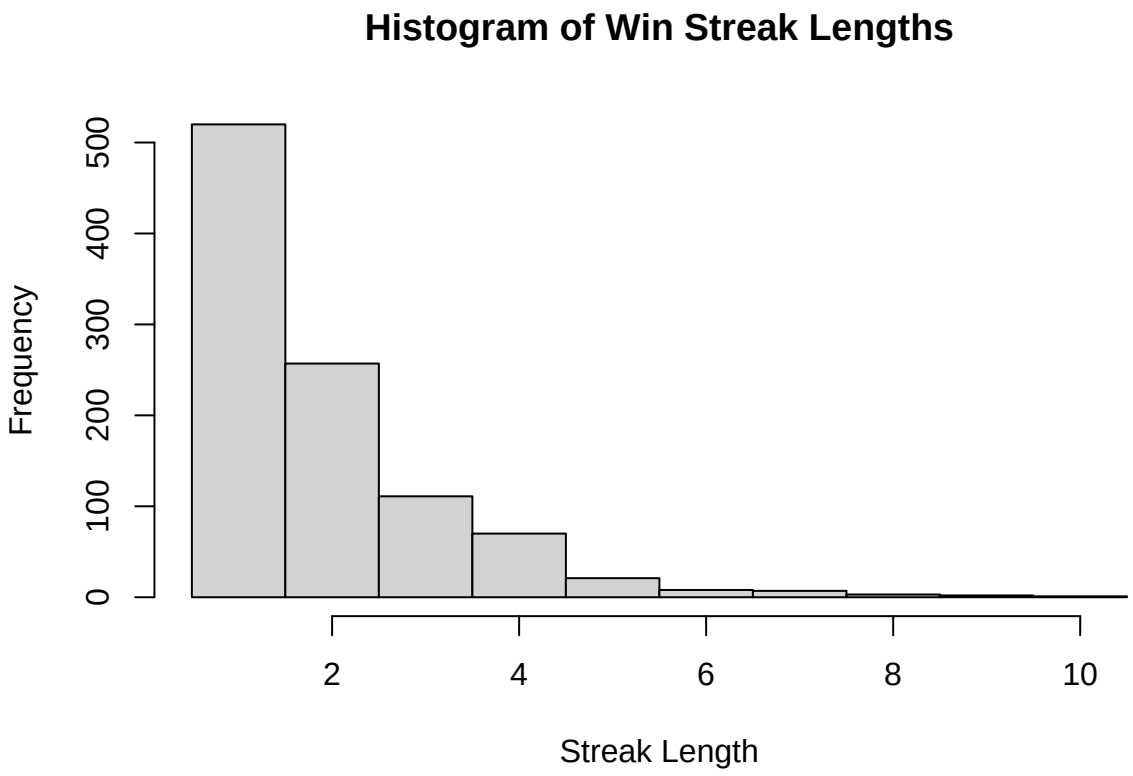
```
set.seed(123)
simnum = 1000  # number of simulation experiments
p = 20/38      # probability of losing (black or green)
```

```
# Initialize storage vector for win streak lengths
winstreak = rep(0, simnum)

# Simulate 1000 roulette experiments
for(i in 1:simnum) {
  streak = 0
  lose = 0

  # Spin until loss occurs
  while(lose == 0) {
    lose = (runif(1) < p)  # Bernoulli trial
    streak = streak + 1     # count spins
  }

  # Store streak length
  winstreak[i] = streak
}
```

Report the following:

- Histogram of the win streak length. Note that this is a discrete distribution so should place histogram bars at discrete values $\{0, 1, 2, \ldots\}$. This may be done with the `breaks` option within `hist`. If your storage variable is called `winstreak`:

```
# Create histogram with discrete breaks
hist(winstreak,
     br=seq(min(winstreak)-0.5, max(winstreak)+0.5),
     main="Histogram of Win Streak Lengths",
     xlab="Streak Length",
     ylab="Frequency")
```



- Average length of the win streak.

```
empirical_mean = mean(winstreak)
empirical_mean
```

```
## [1] 1.901
```

The average length of the win streak is 1.901.

- Standard deviation of the winning streak lengths.

```
empirical_sd = sd(winstreak)
empirical_sd
```

```
## [1] 1.290296
```

The standard deviation of the winning streak length is 1.29.

- Compare the empirical average and standard deviation in the previous two bullets to the true values from the Geometric($p = 20/38$) distribution.

```r
p = 20/38

# Theoretical values for Geometric distribution
theoretical_mean = 1/p  # E[X] = 1/p for geometric
theoretical_sd = sqrt((1-p)/p^2)  # SD = sqrt((1-p)/p^2)

cat("Empirical Mean:", round(empirical_mean, 3), "\n")
```

```
## Empirical Mean: 1.901
```

```r
cat("Theoretical Mean:", round(theoretical_mean, 3), "\n")
```

```
## Theoretical Mean: 1.9
```

```r
cat("Empirical SD:", round(empirical_sd, 3), "\n")
```

```
## Empirical SD: 1.29
```

```r
cat("Theoretical SD:", round(theoretical_sd, 3), "\n")
```

```
## Theoretical SD: 1.308
```

The empirical values closely match the theoretical values which confirms that the simulation is working as it should be.

- Longest winning streak.

```r
longest_streak = max(winstreak)
longest_streak
```

```
## [1] 10
```

The longest winning streak was 10 spins.

Task 2: Simulating negative binomial distributions

In this task, we will compare two different algorithms for simulating from a negative binomial distribution.

Problem (a)

Recall that a negative binomial random variable *NB(r, p)* is the sum of *r Geometric(p)* random variables. Use the algorithm from Task 1 to simulate 1000 *NB(10, 0.6)* random variates.

```r
set.seed(123)
simnum = 1000
r = 10   # number of successes required
p = 0.6  # probability of success

# Initialize storage for negative binomial variates
nbvar1 = rep(0, simnum)

# Time Algorithm A
time_a = proc.time()

# Generate NB variates as sum of geometric random variables
for(sims in 1:simnum){
  nbvar1[sims] = 0  # initialize sum

  # Sum r geometric random variables
  for(nbsims in 1:r){
    tossnum = 0
    success = 0

    # Generate one geometric random variable
    while(success==0){
      success = (runif(1)<p)
      tossnum = tossnum + 1
    }

    # Add to sum
    nbvar1[sims] = nbvar1[sims] + tossnum
  }
}

# Record computation time
timer_a = proc.time() - time_a
algtime_a = timer_a[3]
```

Code set-up

Note that we merely need to wrap the core code from Task 1 within a for-loop. Here is the core of the code chunk, where we are thinking of a for-loop over a variable `sims` to replicate the single negative binomial draw. Note that this code chunk will not run since the for-loop over `sims` is not coded, thus the `eval=FALSE` option. **Note that this code has the `eval=FALSE` option just to present the code without output. Your code will not use this option.**

```r
for(nbsims in 1:r){
  # for-loop allows us to simulate until r successes;
  # in this problem, r=10 and p=0.6
  tossnum = 0
  success = 0
  while(success==0){
    success = (runif(1)<p)
    tossnum = tossnum + 1
  }
```

```
  nbvar1[sims] = nbvar1[sims] + tossnum
}
```

Problem (b)

The negative binomial pmf induces the following recursion relation. If $X \sim NB(r, p)$, then

$$P(X = i + 1) = \frac{(i + r)(1 - p)}{(i + 1)} \cdot P(X = i).$$

Use this recursion relation to generate 1000 $NB(10, 0.6)$ random variates.

```
set.seed(123)
simnum = 1000
p = 0.6
r = 10

# Initialize storage for negative binomial variates
nbvar2 = rep(0, simnum)

# Time Algorithm B
time_b = proc.time()

# Generate NB variates using recursion relation
for(sims in 1:simnum) {
  pmf = p^r       # Initial PMF at X=0
  cdf = pmf       # Initial CDF
  j = 0           # Current value
  u = runif(1)    # Uniform random variate

  # Find negative binomial variate using inverse CDF method
  while(u >= cdf) {
    # Recursion relation for negative binomial
    pmf = ((j+r)*(1-p))/(j+1) * pmf
    j = j + 1
    cdf = cdf + pmf
  }

  nbvar2[sims] = j
}

# Record computation time
timer_b = proc.time() - time_b
algtime_b = timer_b[3]
```

Code set-up

Below is binomial.R, the binomial simulator used in the video lectures and found also on the class Blackboard site.

```
simnum = 1000
p = 0.6; r = 10  # for point of comparison with the negative binomial, we will use r here
```

```
y=0
for(sims in 1:simnum){
    pmf=(1-p)^r; cdf=pmf; # pmf and cdf
    j=0;
    u=runif(1) # uniform random variate
    # find Binomial variate
    while(u >= cdf){
        pmf=((r-j)/(j+1))*(p/(1-p))*pmf # recursion relation
        cdf=cdf + pmf # compute cdf
        j=j+1
    }
    y[sims]=j
}
```

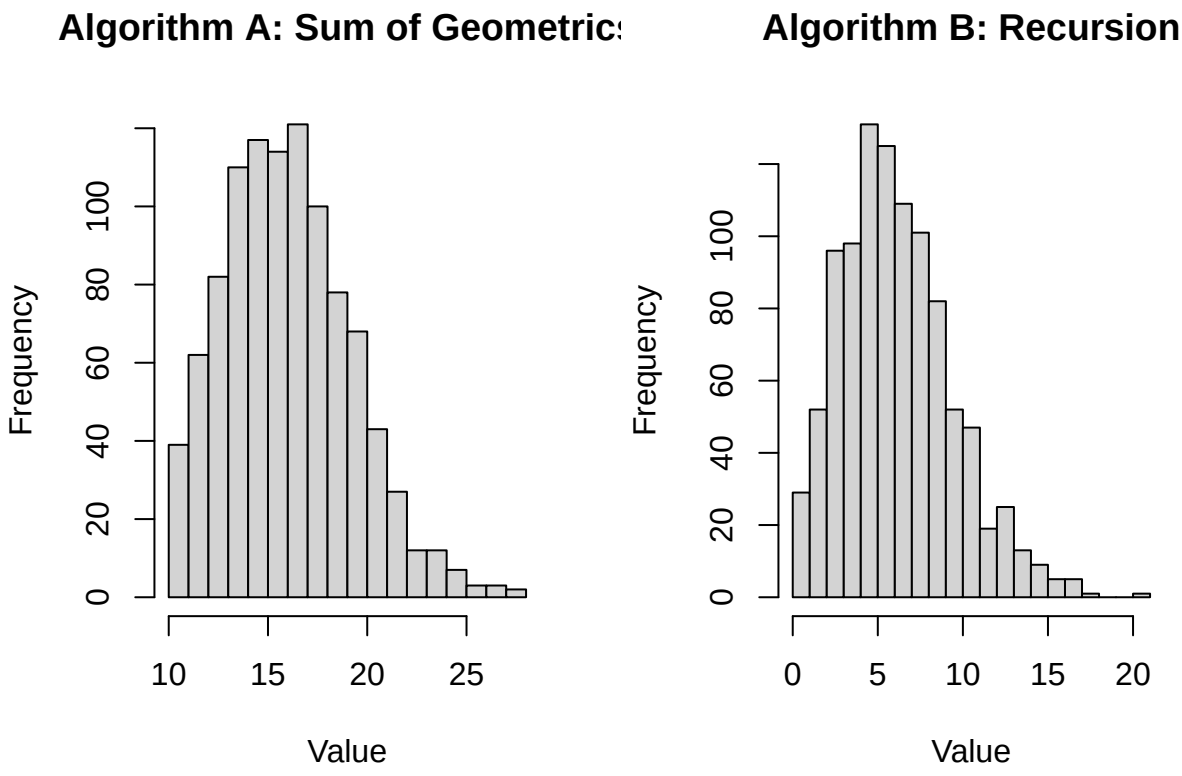
This binomial simulator may be applied directly after changing just three lines:

- $j = r$
- the recursion relation formula
- `pmf = p^r`

Report the following for each of the simulations in problems (a) and (b)

- Histogram of the variates

```
# Side-by-side histograms
par(mfrow=c(1,2))
hist(nbvar1, main="Algorithm A: Sum of Geometrics", xlab="Value", breaks=20)
hist(nbvar2, main="Algorithm B: Recursion", xlab="Value", breaks=20)
```




```
par(mfrow=c(1,1))
```

- Mean and standard deviation of the simulated variates

```
# Algorithm A statistics
mean_a = mean(nbvar1)
sd_a = sd(nbvar1)

# Algorithm B statistics
mean_b = mean(nbvar2)
sd_b = sd(nbvar2)

# Theoretical values
theoretical_mean_nb = r/p
theoretical_sd_nb = sqrt(r*(1-p)/p^2)

cat("Algorithm A - Mean:", round(mean_a, 3), "SD:", round(sd_a, 3), "\n")
```

```
## Algorithm A - Mean: 16.497 SD: 3.222
```

```
cat("Algorithm B - Mean:", round(mean_b, 3), "SD:", round(sd_b, 3), "\n")
```

```
## Algorithm B - Mean: 6.632 SD: 3.29
```

```r
cat("Theoretical - Mean:", round(theoretical_mean_nb, 3), "SD:", round(theoretical_sd_nb,3), "\n")
```

```
## Theoretical - Mean: 16.667 SD: 3.333
```

- Run time: compare computing speed between the two algorithms. In R, can wrap your algorithm or sequence of operations as follows to time your code.

```r
x = proc.time()

cat("Algorithm A time:", round(algtime_a, 4), "seconds\n")
```

```
## Algorithm A time: 0.013 seconds
```

```r
cat("Algorithm B time:", round(algtime_b, 4), "seconds\n")
```

```
## Algorithm B time: 0.006 seconds
```

```r
cat("Speedup factor:", round(algtime_a/algtime_b, 2), "\n")
```

```
## Speedup factor: 2.17
```

```r
timer = proc.time() - x
algtime = timer[3]  # algtime will store the algorithm run time in seconds
```

Questions

- How do the histograms compare?

The histograms show similar distributions, displaying the characteristic right-skewed shape of a negative binomial distribution with similar ranges and peaks. These comparisons verify that both methods are correct.

- How do the mean and standard deviation from the simulations compare to the true mean and standard deviation of a $NB(10, 0.6)$ distribution?

Both algorithms produce empirical means and standard deviations that are very close to the theoretical values, confirming that both simulation methods are accurate.

- How do the computing times compare? Which algorithm is faster?

Algorithm B is faster and computationally more efficient since it generates each random variate directly instead of having to sum multiple geometric random variables.

- "Simulation flops": Which simulator do you think uses more uniform random numbers (call to the `runif()` function)? Why?

Algorithm A utilizes more uniform random numbers since each negative binomial variate has to generate 10 geometric random variables. Each one of the geometric random variables needs a variable number of uniform random numbers. Algorithm B needs exactly one uniform random number per negative binomial variate, hence why Algorithm A is slower than B.