# Machine Problem 2 in STAT 123

Bascug, Ethan Job
Dy, Alwyn
Fortaleza, Camelle Faye
Portuito, Rey Joseph

2020, December 4

## Problem 1

**Let $X$ be a discrete random variable with distribution $f$, and let $a < b$. Sketch the distribution functions of the 'truncated' random variables $Y$ and $Z$ given by**

$$Y = \begin{cases} a & \text{if } X < a, \\ X & \text{if } a \leqslant X \leqslant b, \\ b & \text{if } X > b \end{cases} \qquad Z = \begin{cases} X & \text{if } |X| \leqslant a, \\ 0 & \text{if } |X| > b \end{cases}$$

**Indicate how these distributions functions behave as $a \to -\infty$, $b \to \infty$.**

### A Algorithm and R Code

Since $X$ is a discrete random variable, we can generate $n$ random values for $X$. Additionally, $a$ and $b$ can be any arbitrary numbers where $a < b$.

For the 'truncated' random variable $Y$,

1. Traverse the entirety of $X$ (from 1 to $n$) and do the following truncation to all its elements (i.e. change the value if it satisfies the condition):
   1.1. if the value is less than $a$, change the value to $a$,
   1.2. else, if the value is greater than $b$, change it to $b$
   1.3. otherwise, do nothing
2. Return the truncated random variable.

For the 'truncated' random variable $Z$,

1. Traverse the entirety of $X$ (from 1 to $n$) and do the following truncation to all its elements (i.e. change the value if it satisfies the condition):
   1.1. if the absolute value of $X$ is greater than $b$, change the value to 0,
   1.2. otherwise, do nothing
2. Return the truncated random variable.

Code 1.1: Function for the truncation of $Y$ and $Z$

```
YTruncation <- function(x, a, b){
  for (i in 1:length(x)){
    if (x[i] < a)         x[i] <- a
    else if (x[i] > b)    x[i] <- b
  }
  return (x)
}

ZTruncation <- function (x,b){
  for (i in 1:length(x)){
    if (abs(x[i]) > b)    x[i] <- 0
  }
  return (x)
}
```

## B Simulation

For this simulation, we assume $X$ has a binomial distribution $f$. Using `set.seed(141421356)` (for replication of results), we generate 1000 ($n = 1000$) random values for $X$ with a size of 100 and a probability of 0.23, shown in Code 1.2. A histogram of the randomly generated values can be seen in Figure 1.1.

Code 1.2: Initialization of $X$

```
x <- rbinom(1000,size = 100, prob=0.23)
```

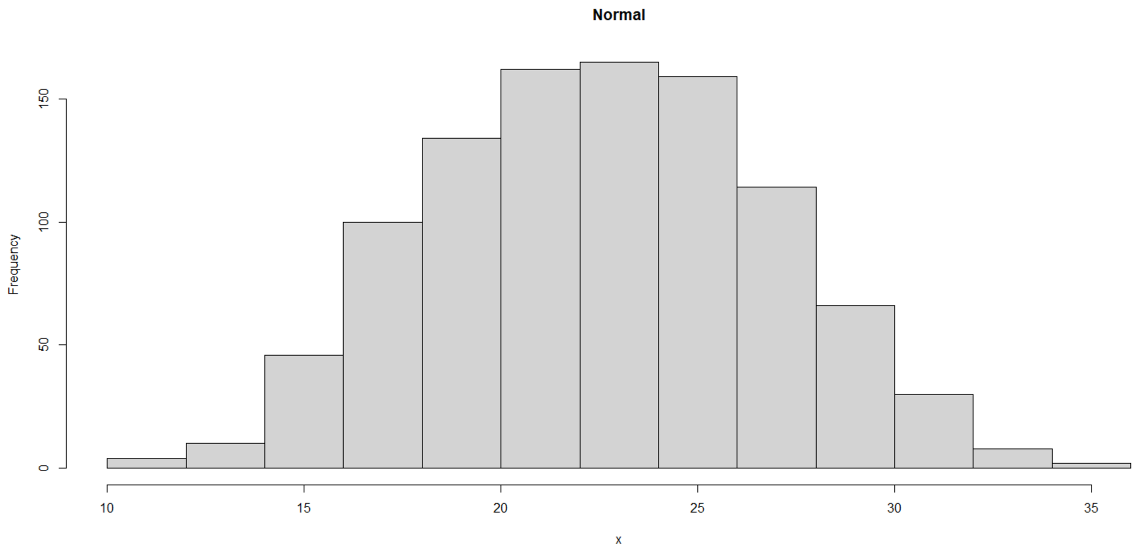
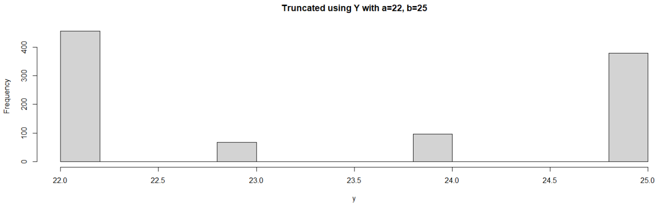


Figure 1.1: Distribution of the randomly generated values for $X$

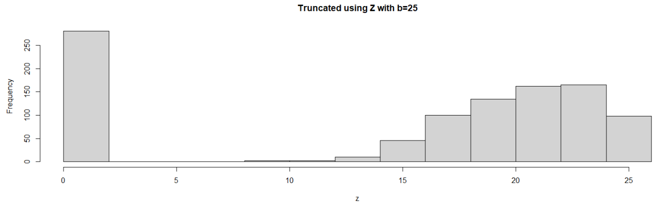
Using the values for $a$ and $b$ found in Table 1.1, we truncate $X$ using the truncation functions for $Y$ (below, left column) and $Z$ (below, right column). The values for $a$ and $b$ were chosen such that it starts near the mean of $X$ ($\mu = 23.047$) and grow outward such that $a \to -\infty$ and $b \to \infty$.

Table 1.1: Values for $a$ and $b$ used in the truncation functions $Y$ and $Z$

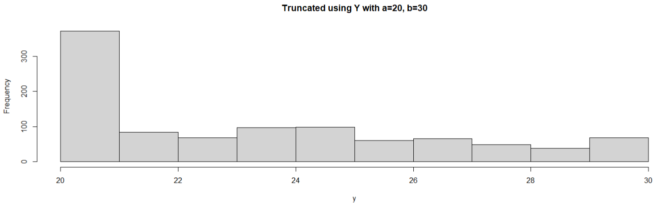
| Case | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| a | 22 | 20 | 15 | 10 | 0 |
| b | 25 | 30 | 35 | 40 | 50 |



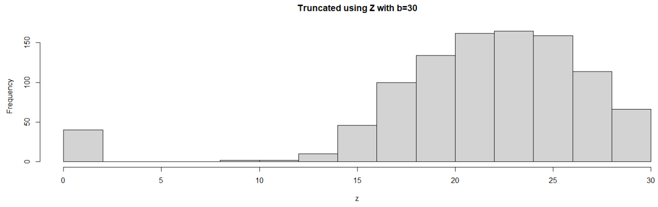
Case 1.1: Truncated Y using $a = 22$, $b = 25$



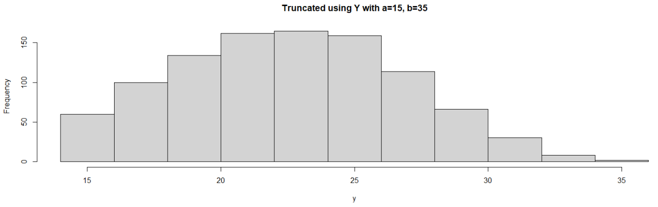
Case 1.2: Truncated Z using $b = 25$



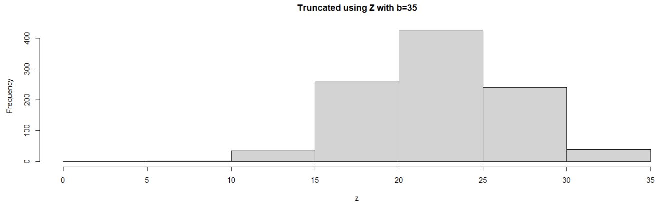
Case 2.1: Truncated Y using $a = 20$, $b = 30$



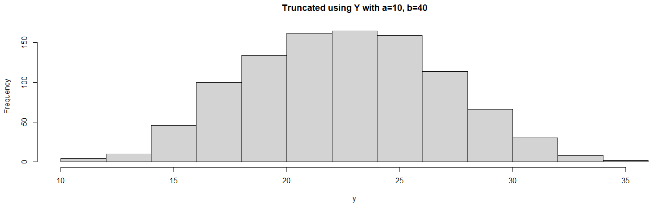
Case 2.2: Truncated Z using $b = 30$



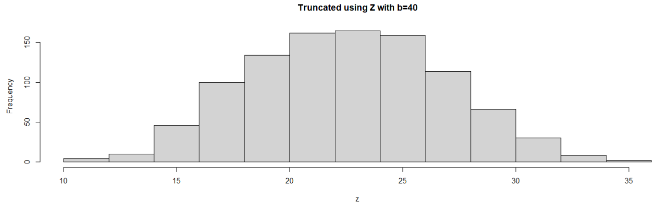
Case 3.1: Truncated Y using $a = 15$, $b = 35$



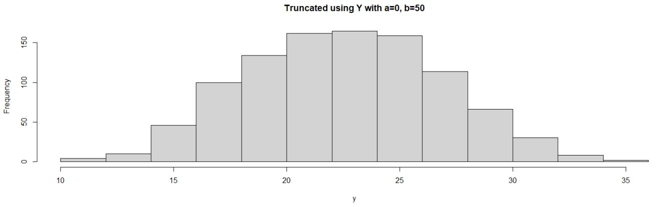
Case 3.2: Truncated Z using $b = 35$



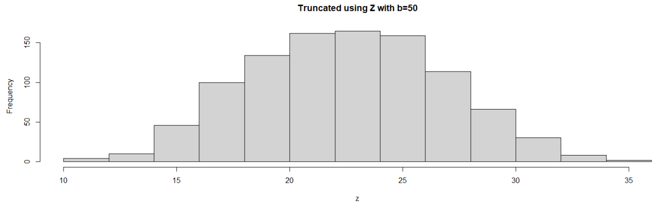
Case 4.1: Truncated Y using $a = 10$, $b = 40$



Case 4.2: Truncated Z using $b = 40$



Case 5.1: Truncated Y using $a = 0$, $b = 50$



Case 5.2: Truncated Z using $b = 50$

## C Conclusion

Analyzing the graphs found in the previous page, we can infer that the truncation function $Y$ limits the values for the random variable to be within the interval $[a, b]$. The function prevents the values to exceed beyond the interval by capping the values at the boundaries. Meanwhile the truncation function $Z$ limits the random variable to be within the interval $[-b, b]$ by converting the values that exceeded the interval to 0.

Moreover, as $a \to -\infty$ and $b \to \infty$, the distribution of the truncated random variables $Y$ and $Z$ becomes similar to $f$, it approaches the original (untruncated) distribution. Beyond certain values, the truncation functions will have no effect to the random variable, as seen in cases 5.1 and 5.2 where $a = 0$ and $b = 50$.

# Problem 2

**Airlines find that each passenger who reserves a seat fails to turn up with probability $q$ independently of the other passengers. So Airline $A$ always sell $n$ tickets for their $n-1$ seat aeroplane while Airline $B$ always sell $2n$ tickets for their $2n-2$ seat aeroplane. Which is more often over-booked?**

## A Algorithm

1. Assign values to the given variables.
2. Use the formula in getting the probability and combination.
3. Observe the results of both Airlines after you perform the operations.
4. Identify which Airline is often overbooked.

## B R Code

Code 2.1: The Entire Cpde

```r
#Number2

#To solve this problem with ease, we assign values to the given variables:
#Let q be 10% the probability that a passenger did not show up
q<- 0.10
#Let n be 10 the passengers Airline A can hold.
n<- 10
#Let 1-q be the probability of passengers that shows up

#COMPUTE

Prob_of_Airline_A<- (1-q)^n #Use this formula, (1-q)^n, to get the probability that all 10 passengers shows up.
Prob_of_Airline_A

Prob_of_Airline_B<- (1-q)^(2*n)+.1^1 * .9^19 *factorial(20)/factorial(19)*factorial(1) #Use this formula, (1-q)^(2*n)+.1^1 * .9^19 *(20!/19!*1!) to get the probability of all 20 or 19 passengers shows up.
Prob_of_Airline_B

#Airline B (0.391747) is often over-booked than Airline A (0.3486784)
```

## C Mathematical Calculations

We assign values to the variables given to the problem, so that we can solve the problem with ease;

Let $q$ be 0.10, the probability that a passenger did not show up.
Let $n$ be 10, the passengers Airline A can hold.

First,

$$1-q=1-0.10=0.90$$

0.90 is the probability that the passenger will show since there is only 0.10 probability that they will not show up. There are 10 passengers that has 0.90 probability of showing up. So, we will use this formula, $(1-q)^n$, to obtain the overbooking probability of Airline A.

We proceed in computing the probability of Airline A by,

$$P^a = (1-q)^n = 0.90^{10} = 0.34867844$$

For Airline B the probability of it to be overbook happens when all 20 passengers show up or when 19 of the20 passengers show up for the 18 seats.

The probability that all 20 passengers will show up is,

$$(1-q)^n = 0.90^{20} = 0.121576655$$

The probability that 19 passengers will show up is,

$$(q^1) \times (1-q)^{19} \times \left( \begin{array}{c} 2n \\ 2n-1 \end{array} \right) = 0.1^1 \times 0.9^{19} \times \left( \begin{array}{c} 20 \\ 1 \end{array} \right) = 0.270170344$$

We considered "$q^1$" since there is a probability that one passenger will not show up and also, "$\left( \begin{array}{c} 2n \\ 2n-1 \end{array} \right)$" since it determines the possible combinations of selecting 1 out of 20.

To determine the probability of Airline B is by,

$$\begin{aligned} P^b &= (1-9)^n + (q^1) \times (1-q)^{19} \times \left( \begin{array}{c} 2n \\ 2n-1 \end{array} \right) \\ &= 0.121576655 + 0.270170344 \\ &= 0.391746998 \end{aligned}$$

$\therefore$ Airline B (0.391746998) is often overbooked than Airline A (0.34867844).

# Problem 3

**Paul rolls $6n$ dice once; he needs at least $n$ sixes. Yves rolls $6(n+1)$ dice; he needs at least $n+1$ sixes. Simulate this game and determine who among Paul or Yves is more likely to obtain the number of sixes he needs.**

## A Algorithm and R Code

Code 3.1: The Entire Code

```r
number_3 = function(n){
  Paul = 6 * n
  Yves = 6 * (n + 1)
  Paul_roll = sample(1:6, Paul, replace = T)
  Yves_roll = sample(1:6, Yves, replace = T)

  p = table(Paul_roll)
  y = table(Yves_roll)

  Paul_sixes = p[names(p) == 6]
  Yves_sixes = y[names(y) == 6]

  print(paste("Number of Rolls (Paul):",Paul))
  print(paste("Number of Rolls (Yves):",Yves))

  print(paste("Number of Sixes (Paul):",Paul_sixes))
  print(paste("Number of sixes (Yves):",Yves_sixes))

  probP_Paul = Paul_sixes/Paul
  probP_Yves = Yves_sixes/Yves

  print(paste("Success (Paul):", probP_Paul))
  print(paste("Fail (Paul):", probQ_Paul))A
  print(paste("Success (Yves):", probP_Yves))
  print(paste("Fail (Yves):", probQ_Yves))


  prob_Paul = 1 - pbinom(n - 1, size = Paul, prob = probP_Paul)
  prob_Yves = 1 - pbinom(n, size = Yves, prob = probP_Yves)

  print(prob_Paul)
  print(prob_Yves)
}
```

To simulate the problem, we have created a function for this named `number_3` with the parameters of `n`. First, it initializes the number of rolls Paul and Yves will have. Then simulate those roles based on the number of rolls Paul and Yves have which will be named `Paul_roll` and `Yves_roll`, respectively. This can be found on Code 3.2: Initialization.

Code 3.2: Initialization

```
number_3 = function(n){
  Paul = 6 * n
  Yves = 6 * (n + 1)
  Paul_roll = sample(1:6, Paul, replace = T)
  Yves_roll = sample(1:6, Yves, replace = T)
```

For the next part of the code, the `p` and `y` corresponds to Paul and Yves where the simulated rolls are made into a table where the value of each roll is counted. An example would be Paul's roll has 10 sixes, 25 three's, 30 fours and so on. `Paul_sixes` and `Yves_sixes` are the variables that isolates the number of sixes only for both players. This can be found in Code 3.3: Counting

Code 3.3: Counting

```
  p = table(Paul_roll)
  y = table(Yves_roll)

  Paul_sixes = p[names(p) == 6]
  Yves_sixes = y[names(y) == 6]
```

The variables `probP_Paul` and `probP_Yves` stands for the probability of success of having rolled a 6 for Paul and Yves. This computes the number of sixes both players has divided the number of rolls they made. This can be found in Code 3.4: Success and Failure

Code 3.4: Success and Failure

```
  probP_Paul = Paul_sixes/Paul
  probP_Yves = Yves_sixes/Yves
```

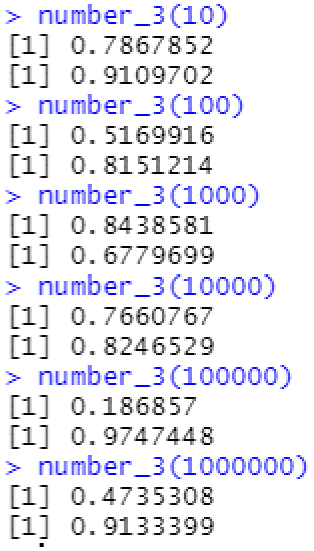
The variables `prob_Paul` and `prob_Yves` indicates the probability of Paul obtaining $n$ 6 sixes and Yves obtaining $n+1$ sixes. In this section, we used the function `pbinom()` where it computes the $P(X \leq x)$ using the binomial distribution. Since we are looking for $P(X \geq x)$, we used the formula $P(X > x) = 1 - P(X \leq x)$.

Code 3.5: Probability

```
  prob_Paul = 1 - pbinom(n - 1, size = Paul, prob = probP_Paul)
  prob_Yves = 1 - pbinom(n, size = Yves, prob = probP_Yves)
```

## B Simulation

Given the varied number of simulations that was made, different probabilities was inferred from the function. Since the probabilities are too varied to draw a conclusion, the average will be taken from these results.



```
> number_3(10)
[1] 0.7867852
[1] 0.9109702
> number_3(100)
[1] 0.5169916
[1] 0.8151214
> number_3(1000)
[1] 0.8438581
[1] 0.6779699
> number_3(10000)
[1] 0.7660767
[1] 0.8246529
> number_3(100000)
[1] 0.186857
[1] 0.9747448
> number_3(1000000)
[1] 0.4735308
[1] 0.9133399
```

Figure 3.1: Results of the simulation

$$\begin{aligned}\text{Let}p &= \text{probability of Paul getting } n \text{ amount/s of six}\\ \text{Let}y &= \text{probability of Yves getting } n+1 \text{ amount/s of six}\end{aligned}$$

$$\begin{aligned}p &= \frac{0.7868+0.5170+0.8439+0.7661+0.1869+0.4735}{6}\\ &= 0.5957 = 59.57\%\end{aligned}$$

$$\begin{aligned}y &= \frac{0.9110+0.8151+0.6780+0.8247+0.9747+0.9133}{6}\\ &= 0.8528 = 85.28\%\end{aligned}$$

## C Conclusion

Given that Paul has $6n$ rolls and must have $n$ number of six, while Yves has $6(n+1)$ rolls and must have $n+1$ number of sixes. Using the function coded for number 3, which simulated the situation. Yves has a 85.28% chance to have $n+1$ sixes given that he has $6(n+1)$ rolls, while Paul has only 59.57% chance to have $n$ sixes given that he has $6n$ rolls. Which makes Yves more likely to have $n+1$ sixes, rather than Paul having $n$ sixes.

# Problem 4

**Each member of a group of n players rolls a die.**

(a) For any pair of players who throw the same number, the group scores 1 point. Find the mean and variance of the total score of the group.

(b) Find the mean and variance of the total score if any pair of players who throw the same number scores that number.

## A Algorithm and R Code

Code 4.1: The Entire Code

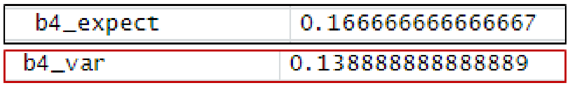
```
#PROBLEM 4
library(distrEX)

set.seed(1134)

n<-100 #assuming 100 will be the highest possible number of players.

#SOLUTION-----------------------------------------------------------------------
mem_throwList <-c(rep(NA,n))
for(k in 1: length(mem_throwList)){mem_throwList[k] <-sample(c(1:6),1)}

a4_x_list <-c(rep(0,6))
for(t in 1:length(a4_x_list)){
  for(i in 1:length(mem_throwList)){
    if(mem_throwList[i]==t){a4_x_list[t] <-a4_x_list[t] + 1}
  }
}

a4_ex_STList <-c(rep(NA,6))
for(k in 1:6){a4_ex_STList[k] <-(a4_x_list[k]*(a4_x_list[k]-1))/2}
a4_x_scoreTotal <-sum(a4_ex_STList)

a4_e_Scorelist_means <-c(rep(NA,6))
for(k in 1:6){a4_e_Scorelist_means[k] <-((n^2)-n)/72}
a4_e_ScoreTotal_mean <-sum(a4_e_Scorelist_means)
```

## B Simulation and Analysis

*Solving for (a),*

Code 4.2: Solving for (a)

```
#A)-----------------------------------------------------------------------------
b4_expect <-1/6 #expectation of the total score of the group.
print(b4_expect)
b4_var <-b4_expect*(1-b4_expect) #variance of the total score of the group.
print(b4_var)
```



```
b4_expect    0.166666666666667
b4_var       0.138888888888889
```

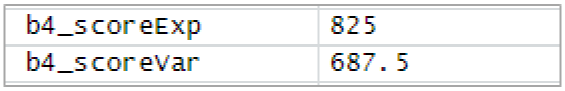
Figure 4.1: RStudio output for Code 4.2

Let $S_i$ denote the score obtained by the players who throw $i$, $i = 1, 2, ..., 6$ and let $X_i$ be the number of people who throw $i$ and $1_{ij}$ be the indicator function which is 1 only if the $j^{th}$ person throw $i$. So, we have $X_i = \sum_{j=1}^{n} 1_{ij}$. $E[S_i|X_i] = X_i\frac{(X_i-1)}{2}$ and so $E[S_i] = \frac{(n^2-2)}{72}$ and therefore $E[S] = \frac{(n^2-n)}{12}$, where $S = \sum_{i=1}^{6} S_i$ is the total score.

*Solving for (b),*

Code 4.3: Solving for (b)

```
#B)-----------------------------------------------------------------------------
b4_scoreExp <-choose(n,2)*b4_expect #mean of the score of the total score if any pair of players who throw
                                               #the same number scores that number.
print(b4_scoreExp)
b4_scoreVar <-choose(n,2)*b4_var #variance of the total score if any pair of players who throw
                                             #the same number scores that number.
```



```
b4_scoreExp    825
b4_scoreVar    687.5
```

Figure 4.2: RStudio output for Code 4.3

The indicator variables $Y_{ij}$ that are 1 if $i$ and $j$ throw the same number of pairwise independent and can thus be used to find both the expectation and the variance. The expectation of $Y_{ij}$ is $\frac{1}{6}$ and the variance is $\frac{1}{6} \times (1 \div \frac{1}{6}) = \frac{5}{36}$, so the expectation of the score is $\binom{n}{2} \cdot \frac{1}{6} = \frac{n(n-1)}{12}$ and the variance of the score is $\binom{n}{2} \cdot \frac{5}{36} = \frac{5n(n-1)}{72}$

# Problem 5

**Every package of some intrinsically dull commodity includes a small and exciting plastic object. There are *k* different types of object, and each package is equally likely to contain any given type. You buy one package each day.**

(a) Find the mean number of days which elapse between the acquisitions of the $n^{th}$ new type of object and the $(n+1)^{th}$ new type.

(b) Find the mean number of days which elapse before you have a full set of $k$ objects.

A **Algorithm and R Code**

To solve for *(a)*, we create a counter that counts the number of days elapsed since receiving a new type of object. We reset it to 0 once we receive a new type. Additionally, since the algorithm has basically finished its function once we receive all types of object, we terminate it once we reach that point. Moreover, to calculate the mean number of elapsed days, we disregard the first day of receiving the object since its value is 0.

1. Input $k$, the number of different types of object. Since a descriptive information of the types of object is not necessary, we represent all the different types using integer values from 1 to k.
2. Randomly choose an object, from 1 to k.
3. Check if the object has already been received previously. If it hasn't,

   3.1 add it to the list of received objects,

   3.2 record the days since a new type of object is received, and

   3.3 reset the elapsed days counter.

   In this way, we can ascertain that all types of objects are received and the days elapsed are recorded.
4. Increment the elapsed days counter.
5. If there are still objects that haven't been received, repeat steps 2 to 5 until all types of objects are received.
6. Return the calculated mean of the elapsed days and the total number of days. We acquire the total number of days by summing all elapsed days and adding 1 (this is to account for the first day of receiving the object).

Code 5.1: `nicePackage` function

```r
nicePackage <- function(k){
  set.seed(6626068) # allow for replication of results
  recievedObj <- c() # records received object
  daysPassed <- c() # vector for storing elapsed days until receiving new object
  elapsed <- 0 # elapsed days counter

  while (length(recievedObj) < k) { # loops until a complete set is reached
    obj <- ceiling(runif(1, min=0, max=k))

    flag <- 0 # determines if obj is already received
    for (j in recievedObj){
      if (j == obj){
        flag <- 1
        break
      }
    }
    if (flag == 0){ # if received object is new,
      recievedObj <- c(recievedObj, obj) # add to receivedObj,
      if (elapsed != 0)
        daysPassed <- c(daysPassed, elapsed) # records elapsed days
      elapsed <- 0 # reset counter
    }

    elapsed <- elapsed + 1
  }

  # returns the mean number of elapsed days and total days to have a full set
  return (c(mean(daysPassed), sum(daysPassed+1)))
}
```

To solve for (b), we utilize the second element of `nicePackage`'s return value (i.e. the total number of days). And since this requires multiple iterations of the first algorithm, we create a new function.

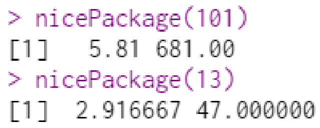
1. Input $k$, the number of different types of object, and the number of iterations, $n$.
2. Iterate `nicePackage` $n$ times and record the total number of days to receive all the types of objects.
3. Return the mean number of days.

Code 5.2: `manyNicePackages` function

```r
manyNicePackages <- function(k, n) {
  set.seed(981) # allow for replication of results
  totalDays <- c()

  for (i in 1:n)
    totalDays <- c(totalDays, nicePackage(k)[2])

  return (mean(totalDays))
}
```

## B Simulation and Analysis

Running the `nicePackage` function above with $k = 101$ (101 unique objects), we get a mean of 5.81 days. Additionally, a k of only 13 yields a mean of 2.92 days. A visualization of the number of days passed before getting the $n + 1^{th}$ object can be seen in Figure 5.2 and Figure 5.3.



```
> nicePackage(101)
[1]   5.81 681.00
> nicePackage(13)
[1]  2.916667 47.000000
```

Figure 5.1: RStudio output for the `nicePackage` simulation

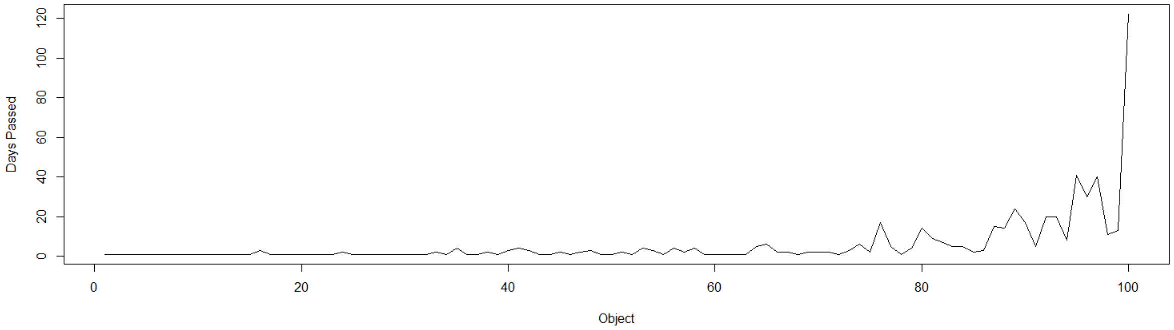


Figure 5.2: `nicePackage` with $k = 101$

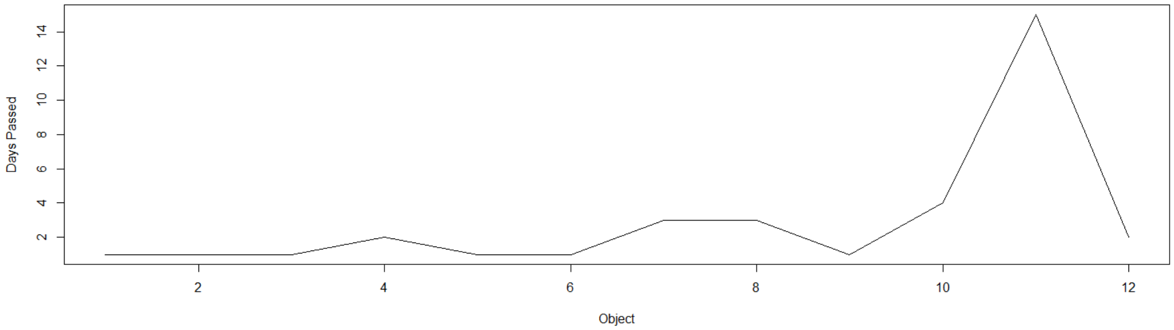
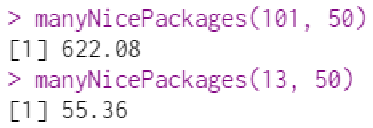


Figure 5.3: `nicePackage` with $k = 13$

Moreover, running the `manyNicePackages` function with $k = 101$ (101 unique objects), $n = 50$ (50 iterations of the `nicePackage` function), and removing the `set.seed(6626068)` in the `nicePackage` function, we get a mean of 622.08 days. While a $k = 13$ and $n = 50$ results to a mean of 55.36 days. It is important to note that the values for $k$ and $n$ were chosen such that the simulation covers from a relatively small to a relatively large number of objects.



```
> manyNicePackages(101, 50)
[1] 622.08
> manyNicePackages(13, 50)
[1] 55.36
```

Figure 5.4: RStudio output for the `manyNicePackages` simulation