Simulation and Optimization

Seth Berry

2021-09-20

Contents

1 Preface **5**

2 Introduction **7**
2.1 The Story 7

3 Linear Optimization **9**
3.1 Continuous Optimization 9

4 Process Simulation **19**
4.1 Discrete Event Simulation 20
4.2 Queueing Theory 20
4.3 Distributions 21
4.4 Performance 28
4.5 The Dispensary 28
4.6 Manufacturing 44

5 Integer Programming **51**
5.1 Troubling Solutions 51
5.2 ClassOverflow 53
5.3 Transportation Problems 54
5.4 Binary Integer Programming 57

6 Simulation **59**
6.1 Distributions 59
6.2 An Important Distinction 66
6.3 ClassOverflow 72

Chapter 1

Preface

You will find some form of this course in undergraduate and graduate business programs across the country. Instead of telling you what you should be looking for in such a class, it is easier to tell you what you shouldn't see:

1. Excel
2. Excel Add-ins
3. Palasaides

While the world runs on Excel, doing Simulation and Optimization in Excel will only ensure that you know Excel (can you use a VLookUp and sumproduct). Instead, the focus will be on understanding and implementing. You *will* be able to break problems down into the smallest parts and then roll those parts into objects. Throughout our time, we will get our hands dirty with some theory. These dives into theory will only serve to ease into greater understanding. In the end, however, the goal is application.

These techniques also serve as a nice introduction to programming in general, as they will allow us to scale from simple objects to very complex pipelines. Throughout our time here, we will mostly focus on using R. However, we don't live in a monolingual world anymore. To that end, we will see how some of these techniques translate to other languages (namely Python and Julia) and why we might want to consider one program over the other (speed, feature complete, ease, etc.).

Chapter 2

Introduction

In a world of exciting methods, simulation and optimization sit alone. Nobody touts how these things will change humanity. Nobody discusses how these methods can solve all of the world's problems. No... those conversations are reserved for things like statistical methods, machine learning, and the almighty AI! The secret, though, is that all of these fancy techniques do not exist without simulation and optimization.

Optimization is found in nearly every science: from nuclear medicine and biology, to electrical engineering and statistics, and beyond. While these fields are interesting on their own, our goal is to explore optimization in the context of business problem solving, with an occasional dive into how these techniques are used in techniques you learn in other courses.

Simulation is just as fundamental to the sciences as optimization. Nearly every event that happens is bound by some type of distribution and knowing that distribution allows us to test that event. What makes simulation so much fun is that you can program a version of the real world, run that program a few thousand times, and then generate a distribution of potential outcomes. This distribution will show us how common an event might be.

2.1 The Story

Meet Ali. Ali graduated from an Business Analytics program on the Atlantic coast in 2019 and made a smart career choice – accepting an offer to work as an analyst in the cannabis industry. The global cannabis industry has seen explosive growth during the last several years (topping 9 billion in 2020 and has a project compound annual growth rate of ~26% in America alone). While some of Ali's classmates (and family) questioned the decision, it was clear that it was an industry in need of some real analytics and Ali saw a real path towards making a difference for a business (after all, most people can't make a real difference in FAANGM). The Canadian cannabis industry has nearly a decade of maturity over the American cannabis industry, and American companies are looking to cover some of that lost ground. To that end, American companies are hiring people from all of the world to create the strongest possible teams. With diverse backgrounds and experiences, the general hope is that teams will function at the highest possible levels.

Ali belonged to a team with 3 other analysts: Alex, Jun, and Shashi. Ali was the youngest and least experienced of the entire group. What Ali lacked in experience, was more than made up by technical prowess. What Ali didn't know is that the analytics world has a dark secret. Throughout Ali's education, Python and R were touted as the most important languages in the world – they are, after all, where all of the exciting work happens. What Ali found, though, was that business analytics really runs on Excel and various add-ins.

Ali had a goal: to become the most valuable member of the team. Ali decided to take on anything the organization needed. It seemed like a good idea at first, but Ali found out that the Business Analytics program didn't really offer the proper preparation for what was to come.

Chapter 3

Linear Optimization

Ali's first task was to determine a marketing strategy. Both the Canadian and American cannabis industries are trying to normalize cannabis use (mainly through edibles and drinks) to women between the ages of 30 and 55. The working theory is that making cannabis use acceptable to this group will "allow" married men to also enjoy recreational cannabis use.

Ali's manager, Tolu, has asked to create a semi-automated system for determining advertisement spends. Thankfully, Tolu noted that Ali's coworker, Jun, has already been working in this space. Ali should be able to jump on Jun's work and make this system automated without much hassle.

3.1 Continuous Optimization

3.1.1 The Problem

What should have been an easy task became a nightmare. Ali didn't get a csv file with neatly defined columns and a clear outcome variable. No... Ali received this email (in which Jun was copied):

> Hi Ali,
>
> Here is what Rayan from Marketing needs:
>
> Instagram ads cost $50 dollars per hundred clicks
>
> TikTok ads cost $20 dollars per hundred clicks
>
> Over the last few weeks, we averaged about 1 female view for Instagram and 4 for TikTok. We need at least 80 female views in total for the coming week.
>
> We don't really do as well with men; we saw just about 1 average male view for both Instagram (.9) and TikTok (.8). We are really hoping to get at least 40 for the coming week.
>
> Where should we buy ads for the coming week?
>
> All my best,
>
> Tolu

3.1.2 From Words To Formulas

In typical analyst fashion, Ali responded with, "No problem!", and started digging through old course notes. Unfortunately, nothing looked like this problem. Ali decided that a cup of coffee with Jun was the way to go. Before a coffee invite even went out, Jun sent Ali a copy of the legacy Excel sheet... completely full of Excel equations and Solver boxes.

Ali had no idea what was going on in the sheet – there were *sumproduct* formulas, *vlookups*, and other strange things. Ali asked Jun to go over the sheet and the problem together, and Jun most graciously agreed.

Jun helped Ali break the problem down into small pieces. "The first question", Jun said, "is what are the variables and what are their values?"

Ali thought for a minute and decided that there are two variables to this problem: Instagram and TikTok. "Correct!", said Jun, "and what values do Instagram and TikTok have?" Ali went back to the email and saw:

> Instagram ads cost $50 dollars per hundred clicks
>
> TikTok ads cost $20 dollars per hundred clicks

"Awesome! The value for Instagram is 50 and the value for TikTok is 20.", Ali said. "And what specifically are those?", Jun asked. Ali wasn't sure, but said, "ad costs". "Now, let me show you something.", and Jun wrote this on a piece of paper:

$$\text{ad cost} = 50_{instagram} + 20_{tiktok}$$

"What else do we need?", asked Jun. Ali thought for a minute and said, "We need to get the rest of the information into the problem!"

> Over the last few weeks, we averaged about 1 female view for Instagram and 4 for TikTok.
>
> We don't really do as well with men; we saw just about 1 average male view for both Instagram (.9) and TikTok (.8)

"Let's put that into our problem", and Jun was back to writing:

$$\begin{aligned}\text{ad cost} = 50_{instagram} &+ 20_{tiktok}\\ 1_{instagram} &+ 4_{tiktok}\\ .9_{instagram} &+ .8_{tiktok}\end{aligned}$$

"Did Rayan have an specific needs for those men and women?", asked Jun. Again, Ali looked at the email and saw:

> We need at least 80 female views in total for the coming week.
>
> We are really hoping to get at least 40 for the coming week.

"So,", Ali began, "We need at least 80 views for women and 40 views for men. We could have more for both, though... it is just the baseline."

"Excellent! Check this out", and Jun added the following:

$$\begin{aligned}\text{ad cost} &= 50_{x1} + 20_{x2}\\ \text{women} &= 1_{instagram} + 4_{tiktok} \geq 80\\ \text{men} &= .9_{instagram} + .8_{tiktok} \geq 40\end{aligned}$$

"And we are almost there!", Jun smiled and then asked, "What would we want to do with cost: spend as much as possible or as little as possible?" "Oh", Ali said, "that's easy: we definitely want to minimize our cost."

$$\begin{aligned}&\text{Minimize:}\\ &\text{ad cost} = 50_{instagram} + 20_{tiktok}\\ &\text{Subject to:}\\ &\text{women} = 1_{instagram} + 4_{tiktok} \geq 80\\ &\text{men} = .9_{instagram} + .8_{tiktok} \geq 40\end{aligned}$$

"Here's the last question", Jun said, "Could we buy a negative number of ads?". "Absolutely not", Ali said.

"We have this now", and Jun showed Ali his paper

$$\begin{aligned} &\text{Minimize:} \\ &\text{ad cost} = 50_{instagram} + 20_{tiktok} \\ &\text{Subject to:} \\ &\text{women} = 1_{instagram} + 4_{tiktok} \geq 80 \\ &\text{men} = .9_{instagram} + .8_{tiktok} \geq 40 \\ &\text{instagram, tiktok} \geq 0 \end{aligned}$$

"Now that we have this put completely together, we need to break it down", Jun laughed.

"We know that this is a **minimization** problem", Jun said and continued, "and we know that we have two **variables**: Instagram and TikTok. You may hear people call these **objective values**."

Jun carried on, "We also know that we have some rules to follow for our problem. These rules are called **constraints**. Think of these constraints as rules that reflect reality".

"Got it!", exclaimed Ali.

"Let's see them again", Jun said:

$$\begin{aligned} &\text{Subject to:} \\ &\text{women} = 1_{instagram} + 4_{tiktok} \geq 80 \\ &\text{men} = .9_{instagram} + .8_{tiktok} \geq 40 \end{aligned}$$

"This whole thing is the **constraint matrix** and we can break it down into its component parts!", beamed Jun.

"Let's start with the **left-hand side** of the constraint matrix, which you might hear referred to as the **A matrix**.

$$\begin{aligned} 1_{instagram} + 4_{tiktok} \\ .9_{instagram} + .8_{tiktok} \end{aligned}$$

"We have 4 values, spread across 2 columns and 2 rows.", Jun said, "Just like a normal table". Jun continued, "Next we come to the **directions**... those things look like inequalities, but we will also probably encounter some equalities too".

"Here, we just have a simple **vector** of those signs.", Jun wrote:

$$\begin{aligned} \geq \\ \geq \end{aligned}$$

"Last thing, I promise.", Jun said: "The **right hand side**, those values that we need to achieve, are referred to as the **marginal values**."

$$\begin{aligned} 80 \\ 40 \end{aligned}$$

"Tolu said that you were going to program these in Q or Boa, or something like that. I can't help you there, but let me know if I can do anything else for you", Jun said and walked back to the office.

Ali felt better, but getting all of that information into R was going to be a little bit tricky.

3.1.3 Application

Ali was feeling pretty good after all of this! As soon as the computer was unlocked, StackOverlow came to the rescue – a user called Not_Prof_Berry had answered a few questions about linear programming with R.

It seemed like Ali was going to need a package called `linprog`:

```
# install.packages('linprog')

library(linprog)
```

The specific function is `solveLP`, but Ali saw that it needed some objects to be created first: `cvec`, `bvec`, `Amat`, and `const.dir`. Ali remembered a common mantra among professors – "Read the flipping manual!". After reading the helpfile, Ali determined that the cvec object needed to contain the objective values:

```
objective_values <- c(50, 20)
```

Ali then figured out that bvec came from the right-hand side of the constraint matrix (the values out in the margin of the constraint matrix):

```
constraint_values <- c(80, 40)
```

The Amatrix felt a little bit tricky. It definitely needed to be the constraint matrix, but it was somewhat tough to get into the right shape. Ali tried a few things:

```
constraint_matrix <- rbind(c(1, 4),
                           c(.9, .8))

constraint_matrix2 <- matrix(c(1, 4, .9, .8),
                             ncol = 2, nrow = 2,
                             byrow = TRUE)
```

Both returned a matrix:

```
str(constraint_matrix)
```

```
 num [1:2, 1:2] 1 0.9 4 0.8
```

```
str(constraint_matrix2)
```

```
 num [1:2, 1:2] 1 0.9 4 0.8
```

Which Ali knew was needed for function to work properly.

Finally, Ali made a character vector of const.dir (i.e., the constraint directions):

```
constraint_directions <- c(">=", ">=")
```

With those 4 objects, Ali was ready to solve the problem!

```
solved_model <- solveLP(cvec = objective_values,
                        bvec = constraint_values,
                        Amat = constraint_matrix,
                        maximum = FALSE,
                        const.dir = constraint_directions)
```

With the model solved, Ali needed to grab some information: the recommended values for Instagram and TikTok, and how much money it was going to cost.

First, how much money was this going to cost:

```
solved_model$opt
```

```
[1] 1000
```

Got it. $1000 is the total spend.

Second, what is the marketing mix:

```
solved_model$solution
```

```
 1  2
 0 50
```

That gives 0 ads for Instagram and 50 ads for TikTok! There is no way that is correct. Everyone knows that a 0 for an answer means that there has to be a problem. How could Ali take this solution to Tolu? Ali figured the best course of action would be to check the solution with Jun.

3.1.4 Theory

Like all early-career analysts, Ali was feeling beaten – going back to Jun so quickly felt like a failure. Like all experienced analysts, Jun was only too happy to help and explain what was happening.

Jun reminded Ali of the complete notation they had created together.

$$\begin{aligned} &\text{Minimize:} \\ &\text{ad cost} = 50_{instagram} + 20_{tiktok} \\ &\text{Subject to:} \\ &\text{women} = 1_{instagram} + 4_{tiktok} \geq 80 \\ &\text{men} = .9_{instagram} + .8_{tiktok} \geq 40 \\ &\text{instagram, tiktok} \geq 0 \end{aligned}$$

“Let’s break this down a little bit”, and turning to the whiteboard, Jun wrote:

$$1_{instagram} + 4_{tiktok} = 80$$

Can be solved with:

$$(instagram = 0, tiktok = 20) \text{ or } (instagram = 80, tiktok = 0)$$

And:

$$.9_{instagram} + .8_{tiktok} = 40$$

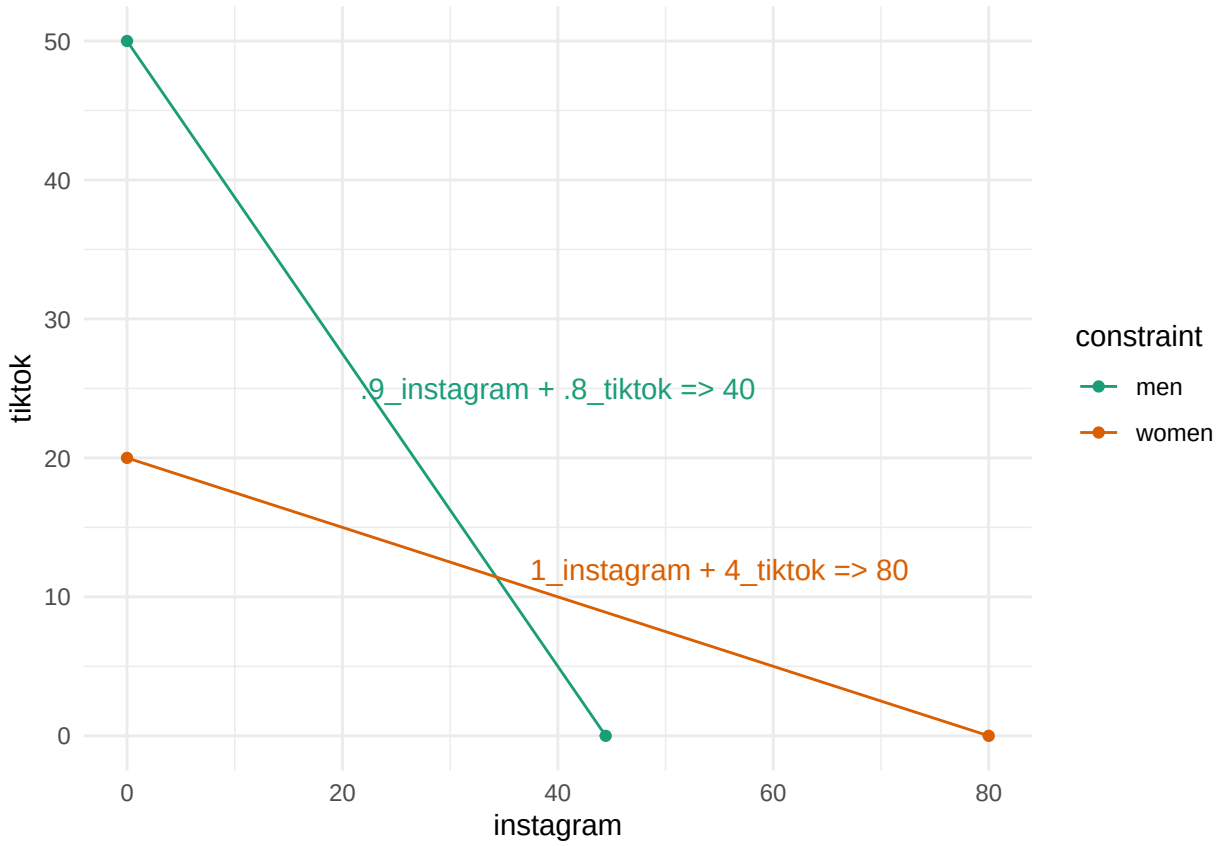
Can be solved with:

$$(instagram = 44.44, tiktok = 0) \text{ or } (instagram = 0, tiktok = 50)$$

“It’s okay if you don’t remember or didn’t take linear algebra”, Jun noted, “just know that we are solving these equations to obtain a set of points.”

“Okay”, Ali nodded, “but what do we do with those points?”

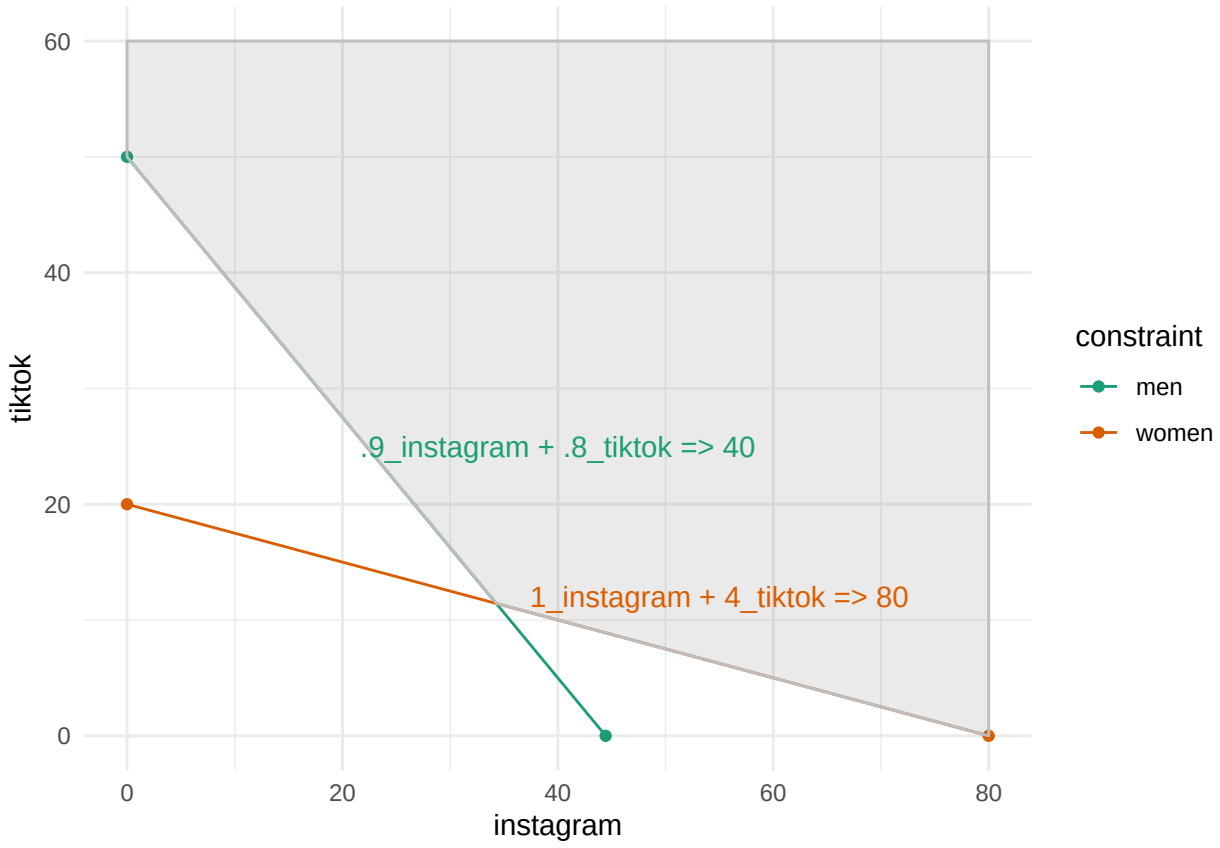
“Plot them”, and Jun went back to writing:



"Now that we have those lines plotted, we can clearly see where we might find our answer!", beamed Jun.

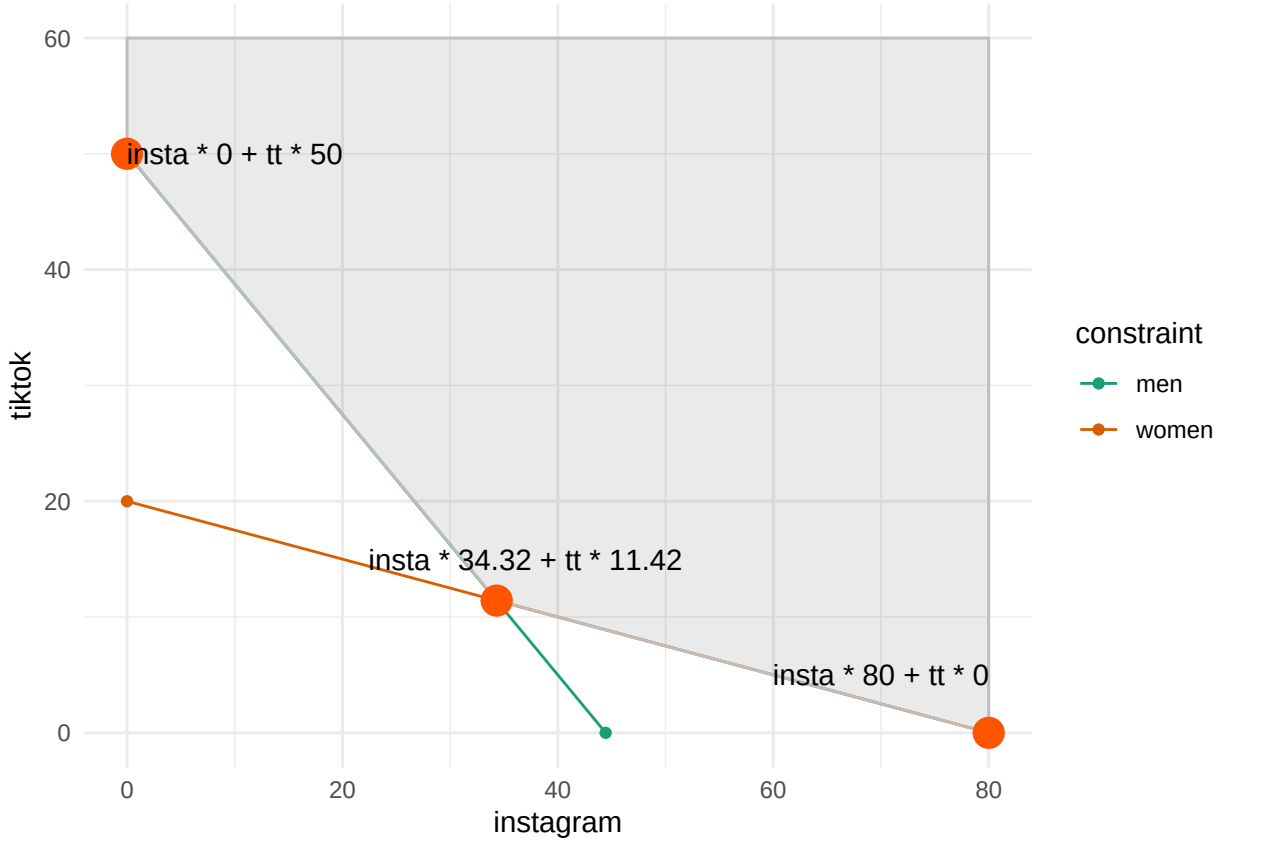
"Umm. . . it's a little fuzzy", admitted a puzzled Ali.

"Completely to be expected", Jun smiled. "Let's find the **feasible region** – this is the place where our answer lives!"



"We could go through and try every single set of points in this shaded area, but we would never finish and we would be wasting our time", Jun chuckled and continued, "We already know that we are looking for the values that minimize our solution, so we can completely ignore every point that doesn't sit on our lines."

Jun made a few circles on the plot and said, "These points will give us our answer!"



"This is called the **extreme point theorem** and it basically says that our **optimal** solution has to rest somewhere in the extreme points of the feasible region", said Jun, "and it makes life so much easier for finding our answer."

"We can just do the simple math now", and Jun wrote"

```
insta_cost = 50

tt_cost = 20

insta_cost * 0 + tt_cost * 50
```

```
[1] 1000
```

```
insta_cost * 34.32 + tt_cost * 11.42
```

```
[1] 1944.4
```

```
insta_cost * 80 + tt_cost * 0
```

```
[1] 4000
```

"Which of those is the smallest value?", Jun asked.

The conference room glowed with Ali's excitement! "I got it now!", Ali exclaimed. "We can get our optimal value of 1000 by purchasing 50 ads on TikTok and 0 on Instagram! The solution was correct!"

"Ahhh!", Jun started laughing, "it is definitely the optimal solution, but do you *really* think that it's the correct solution?" Jun shook his head and continued laughing, "How do you think Tolu is going to take the advice to not put anything at all on Instagram?"

via GIPHY

Ali's mind was sufficiently wrecked. How could an optimal answer not be the correct answer? Stupid analytics – nothing can ever be easy.

"What's the best path forward, then?", Ali asked Jun. "Simple", Jun replied, "ask how much they want to put on Instagram and that becomes a constraint!"

This was to be an important lesson for Ali: analytics tasks are never a one-shot deal. Clarity needs to be sought before most work can actually happen.

After a quick email exchange with Rayan from Marketing, Ali found out that at least 10 ads were needed for Instagram.

3.1.5 ClassOverflow

Let's spend some time helping Ali. We need to do two things: 1) specify an appropriate model and 2) solve it.

We will do this two different ways; both are good to know, but I'd imagine that you will find one to be more valuable than the other.

3.1.6 Using Python

A great chunk of Ali's coursework was in R, with just some excursions into Python. For statistics, R reigns supreme (but statsmodels in Python is really pretty solid). For machine learning, take your pick (only fanboys speak in absolutes about one being better than the other – both have their pros and cons). Linear programming is a bit different. R has some clear advantages in terms of flexibility, but Google has put effort towards implementing their GLOP solver in Python (among other languages).

The `pulp` package is going to look different than what we saw in R, but there are some definite improvements in expressing our model:

```
from pulp import *

model = LpProblem(name = "test-model",
  sense = LpMinimize)

x = LpVariable(name = "instagram", lowBound = 0)
y = LpVariable(name = "tiktok", lowBound = 0)

model += (1 * x + 4 * y >= 80, "women")
model += (.9 * x + .8 * y >= 40, "men")

obj_func = 50 * x + 20 * y
model += obj_func

model

status = model.solve()

model.objective.value()

x.value()
y.value()

for var in model.variables():
  print(f"{var.name}: {var.value()}")
```

We really aren't breaking our problem down into small objects here. Instead, all we really need to do is to take our math form and pop that into our `model` – pretty easy stuff.

Finally, here is Google's OR tools. It is the current SoTA for optimization (how do you think Google gets people navigated). You'll notice that we aren't really doing anything too different than what we saw with pulp:

```
from ortools.linear_solver import pywraplp

solver = pywraplp.Solver.CreateSolver('GLOP')
x = solver.NumVar(0, solver.infinity(), 'instagram')
y = solver.NumVar(0, solver.infinity(), 'tiktok')

solver.NumVariables()
```

```
2
```

```
solver.Add(1 * x + 4 * y >= 80)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type 'operations_research::MPConst
```

```
solver.Add(.9 * x + .8 * y >= 40)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type 'operations_research::MPConst
```

```
solver.NumConstraints()
```

```
2
```

```
solver.Minimize(50 * x + 20 * y)

status = solver.Solve()

solver.Objective().Value()
```

```
999.9999999999999
```

```
x.solution_value()
```

```
0.0
```

```
y.solution_value()
```

```
49.99999999999999
```

Chapter 4

Process Simulation

Ali was absolutely smoked (no pun intended) after handling all of that optimization – hopefully some more standard modeling would come through. High hopes are always short lived, though, and Ali was thrown right back into the dark arts of Operations-based research.

During an "all-hands" meeting, Ali had the chance to listen to Rene, the Director of Retail Analytics, talk about store efficiency. Rene explained the process that typically happens.

"Our average store front is pretty small and we need to be careful about how many people are inside at any one time", Rene started. "We can't have more than 8 people waiting in the lobby before their ID's are checked", and Rene continued, "we don't want to turn people away, so we need to get more efficient in ID checks and bud-tending".

Ali was feeling good after some success and wasn't afraid to ask some questions – "Can you explain the process to me?", Ali asked.

"Sure", Rene said. "It is pretty easy, people walk in the door and wait for their ID to be checked."

"Once their ID has been checked they can meet with one of our bud-tenders – they pick their poison and then pay."

Ali was drawing the process out and made sure that it was correct:

"Seem about right?", Ali asked.

```
library(DiagrammeR)

grViz("
digraph {
  graph [overlap = true, fontsize = 10, rankdir = LR]

  node [shape = box, style = filled, color = black, fillcolor = aliceblue]
  A [label = 'ID Check Line']
  B [label = 'ID Check']
  C [label = 'Bud Tender Line']
  D [label = 'Bud Tender']
  E [label = 'Pay']

  A->B B->C C->D D->E
}
")
```

"That's right", Rene said.

"How many people are checking IDs and how many bud tenders do you have?", Ali asked.

"It kinda depends", Rene said, "but it is usually just one person checking IDs and usually two bud tenders."

Ali had one more question: "How long do each of those steps take?"

"Uhhhhh... I'll have to ask around and get back with you", Rene noted.

"Most excellent", Ali thought, "that will give me some time to chat with Alex." Alex was the resident expert of simulations of all kind, so Ali knew an ally was there.

4.1 Discrete Event Simulation

When Ali finally caught up with Alex, Alex was only too happy to share some of the finer points on process simulation. First, Alex made note that the particular type of simulation under conversation wasn't just a process simulation, but was something called *discrete event simulation* (DES) – events are individual processes and some items goes through a series of those individual processes.

"It has roots in manufacturing, but some many things in life can be modeled through DES", Alex said.

"Let's start with something horribly boring... lines."

4.2 Queueing Theory

"There is a whole field of study regarding lines", Alex said, "and it is called *queueing theory*."

"We don't have to get crazy, but there is this thing called Kendall's Notation... it just describes the particular parts of how lines form and the distributions that guide them."

Ali thought, "The theory of lines... how do some people ever find love."

Alex could almost feel Ali's thought and said, "It really is more interesting than it sounds."

"Check this out!", and Alex was off to the races.

$A/B/C/D$

Where:

$A = arrival\,process$

$B = service\,process$

$C = server\,number$

$D = que\,capacity$

$M/D/k$

$M/M/k$

M generally stands for Markov or Exponential

D is deterministic: all jobs require a fixed amount of time.

k is the number of servers/workers/etc.

"Both of these are generally assumed to have an **infinite queue**... that is important to remember."

"If a queue is $M/D/k$, we can easily compute some helpful statistics."

λ = arrival rate

μ = service rate

$\rho = \frac{\lambda}{\mu}$ = utilization

Average number of entities in the system is:

$$L = \rho + \frac{1}{2}\left(\frac{\rho^2}{1-\rho}\right)$$

Average number in queue:

$$L_Q = \frac{1}{2}\left(\frac{\rho^2}{1-\rho}\right)$$

Average system waiting time:

$$\omega = \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)}$$

Average waiting time in queue:

$$\omega_Q = \frac{\rho}{2\mu(1-\rho)}$$

"The equations are not the important part here", Alex said, "but the idea that the equation captures is critical."

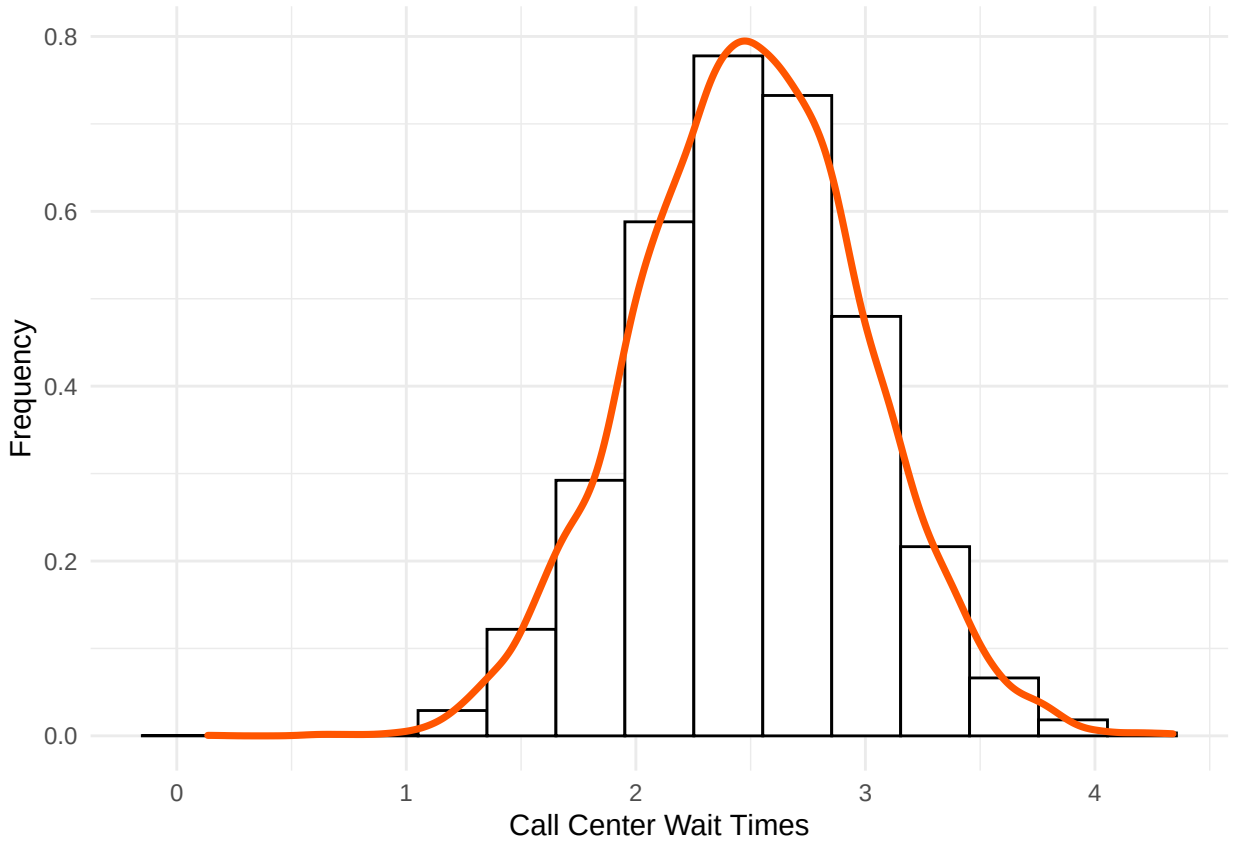
4.3 Distributions

"Distributions drive every single part of DES – every event that you can ever imagine comes from some type of distribution."

4.3.1 Normal Distribution

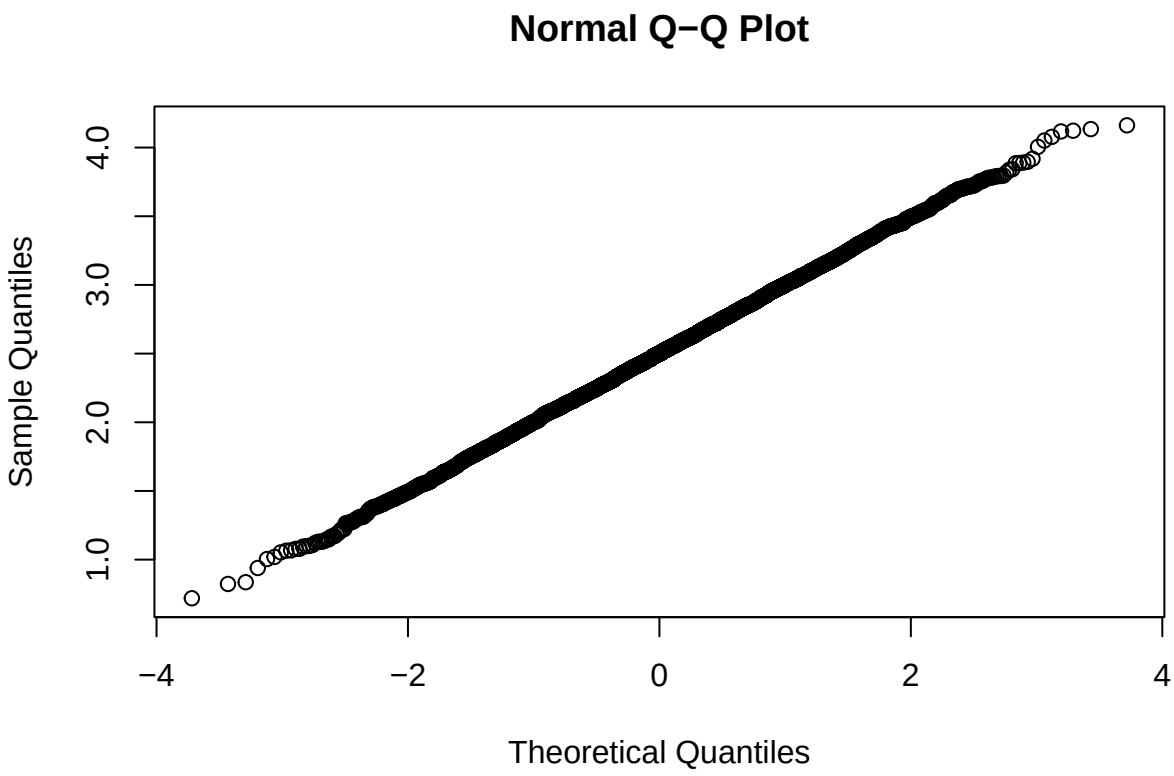
"For our normal distribution, we know the μ and σ."

"It is definitely the most common and you'll find that a lot of processes are normally distributed."

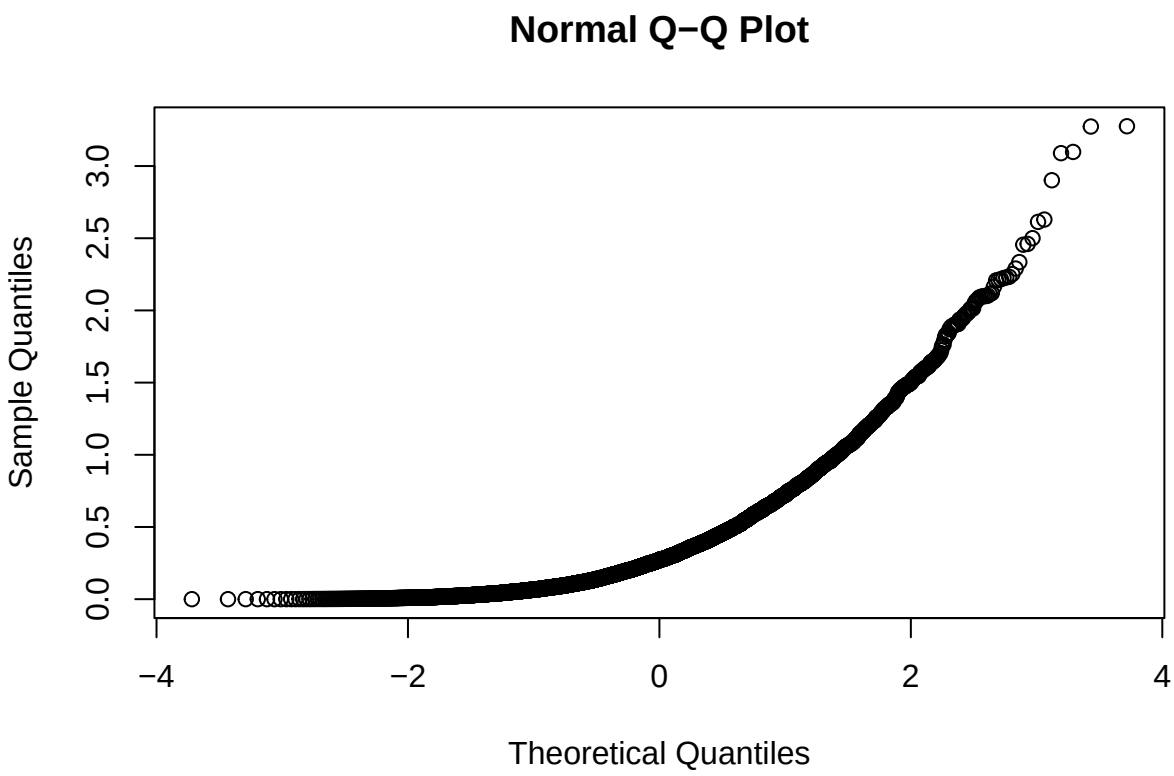


"You might get process data and want to test if a variable is normally distributed."

```
# Just creating data from a normal distribution:

normal_variable <- rnorm(n = 5000, mean = 2.5, sd = .5)

# And an exponential distribution:

exponential_variable <- rexp(n = 5000, rate = 2.5)

# We can check both with a qq plot for normality:
qqnorm(normal_variable)
```




```
qqnorm(exponential_variable)
```




```
# You'll notice the normal distribution
# just plots a straight, diagonal line.
```

```
# The shapiro test will give a test statistic:
shapiro.test(normal_variable)
```

```
    Shapiro-Wilk normality test

data:  normal_variable
W = 0.99978, p-value = 0.9141
```

```
shapiro.test(exponential_variable)
```

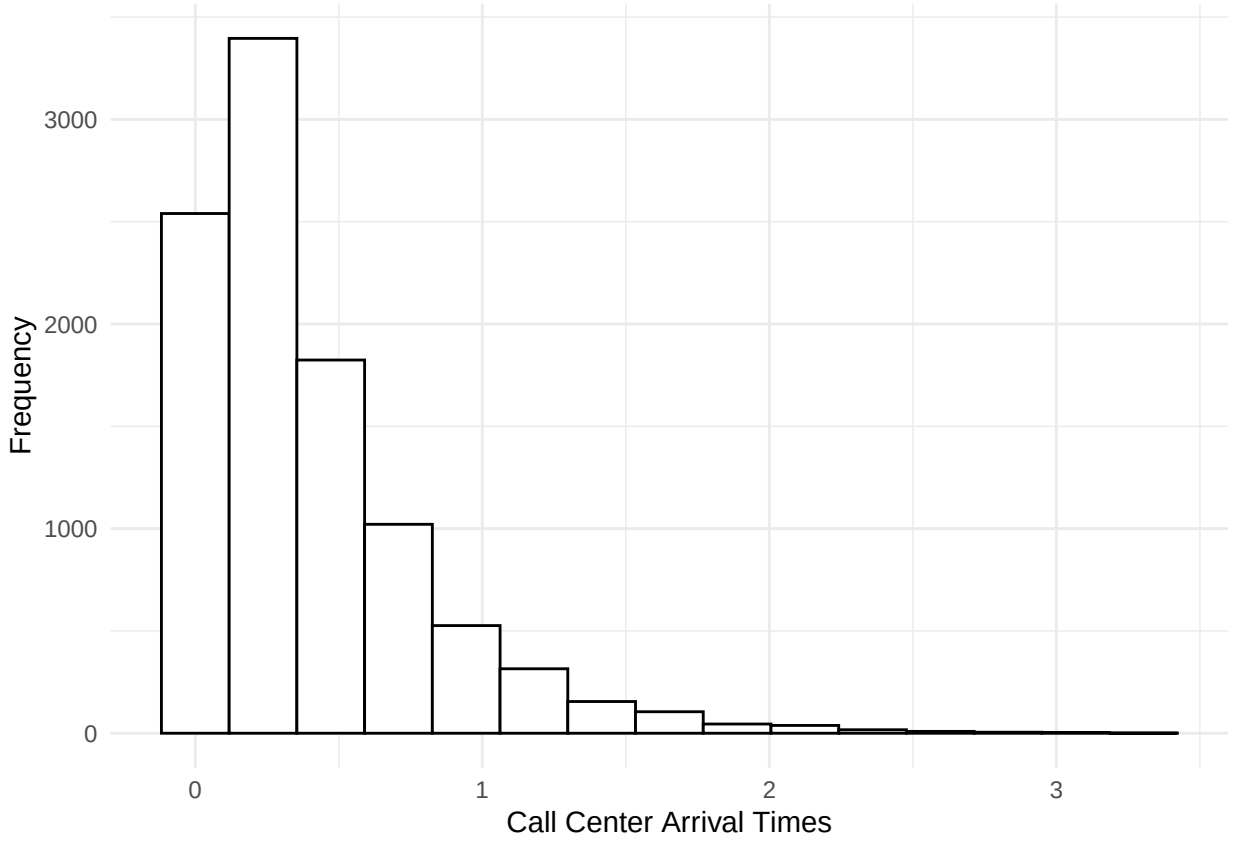
```
    Shapiro-Wilk normality test

data:  exponential_variable
W = 0.81317, p-value < 2.2e-16
```

```
# An insignificant p-value would indicate
# no difference from normality.
```

4.3.2 Exponential Distribution

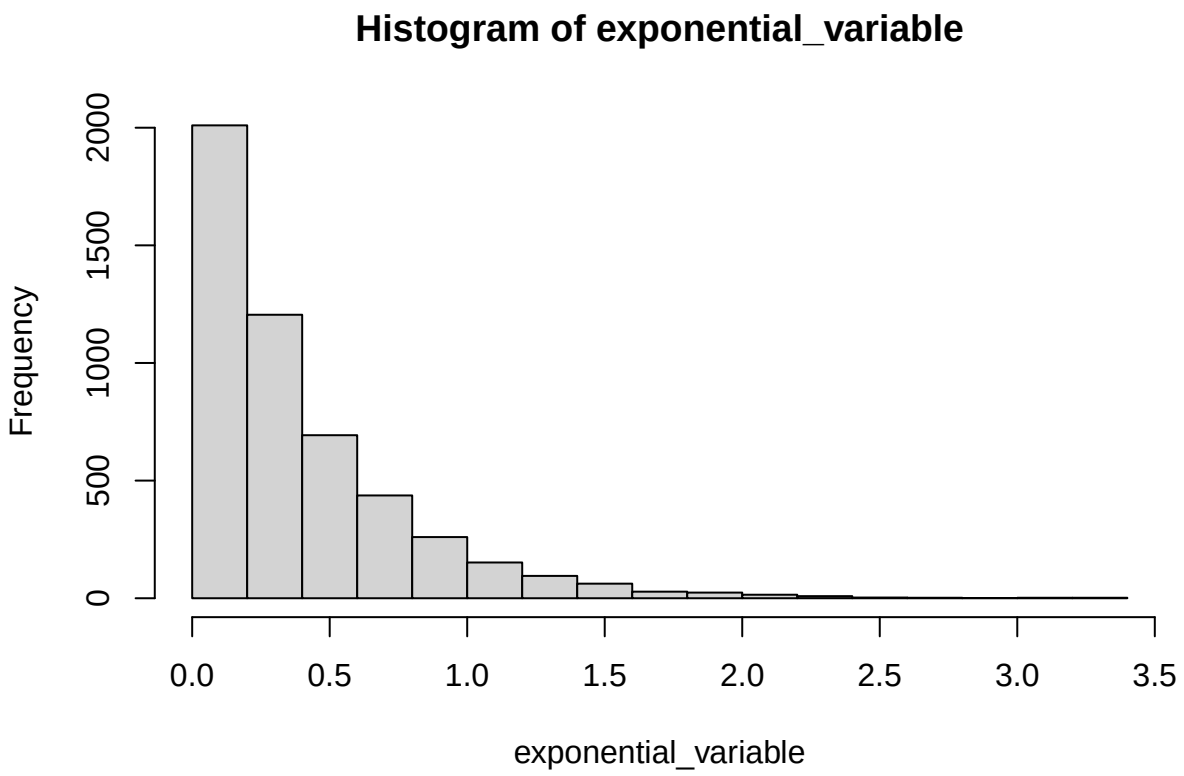
"We can only know one thing about the exponential distribution: μ (also expressed as a rate)."

"Just about any arrival process can be approximated by an exponential distribution."



"Wanna test it?"

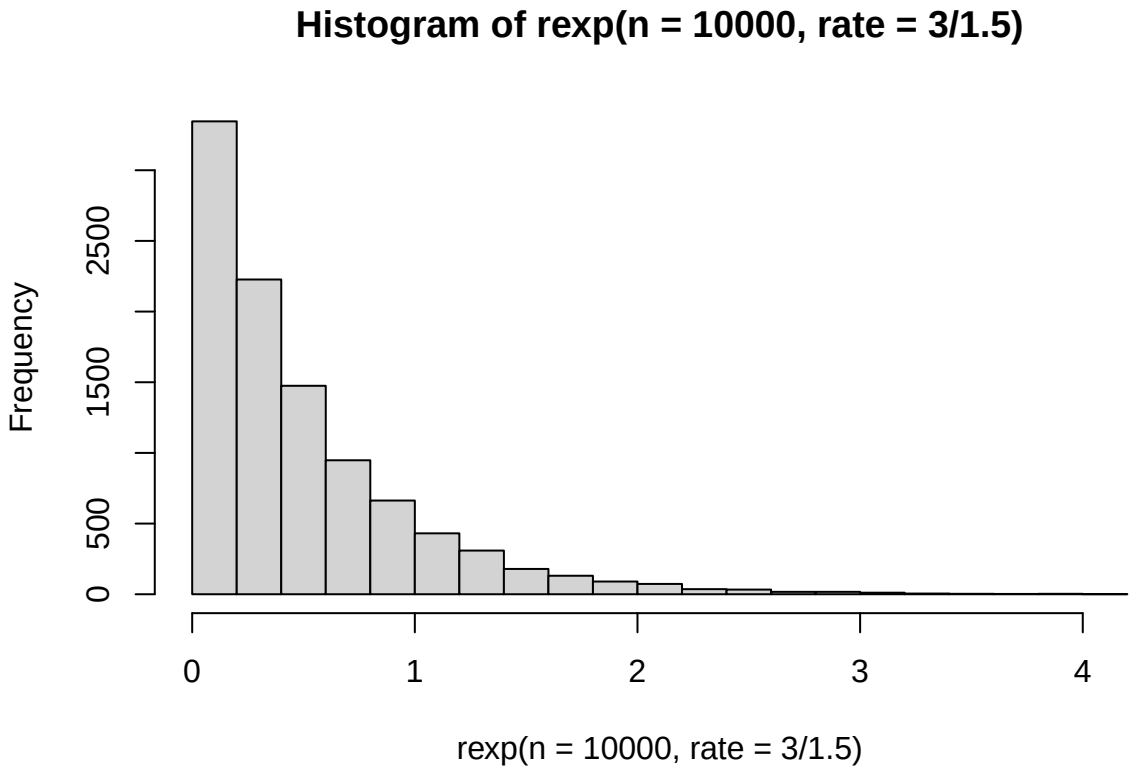
```
hist(exponential_variable)
```



"That rate parameter is a bit confusion. . . the easiest way to express it is as a ratio."

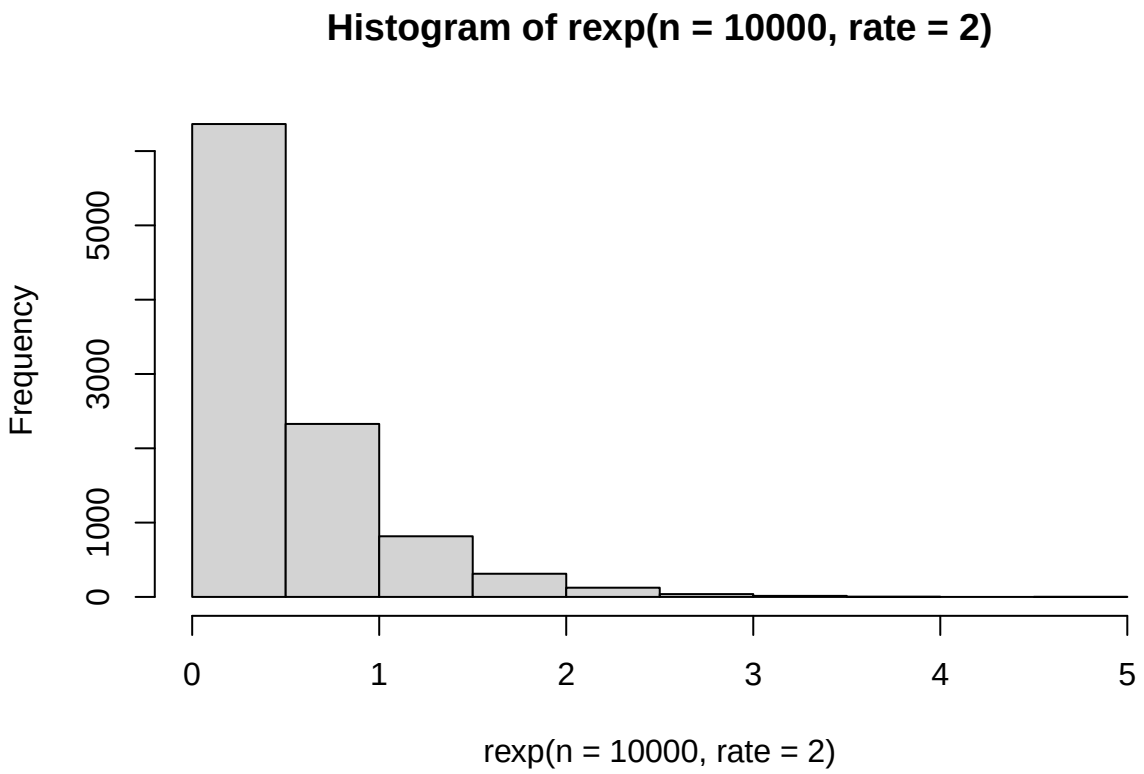
"If 3 people enter the line every minute and a half, than it would be a rate of 3/1.5".

```
hist(rexp(n = 10000, rate = 3 / 1.5))
```



"You could also just put the average rate in."

```
hist(rexp(n = 10000, rate = 2))
```

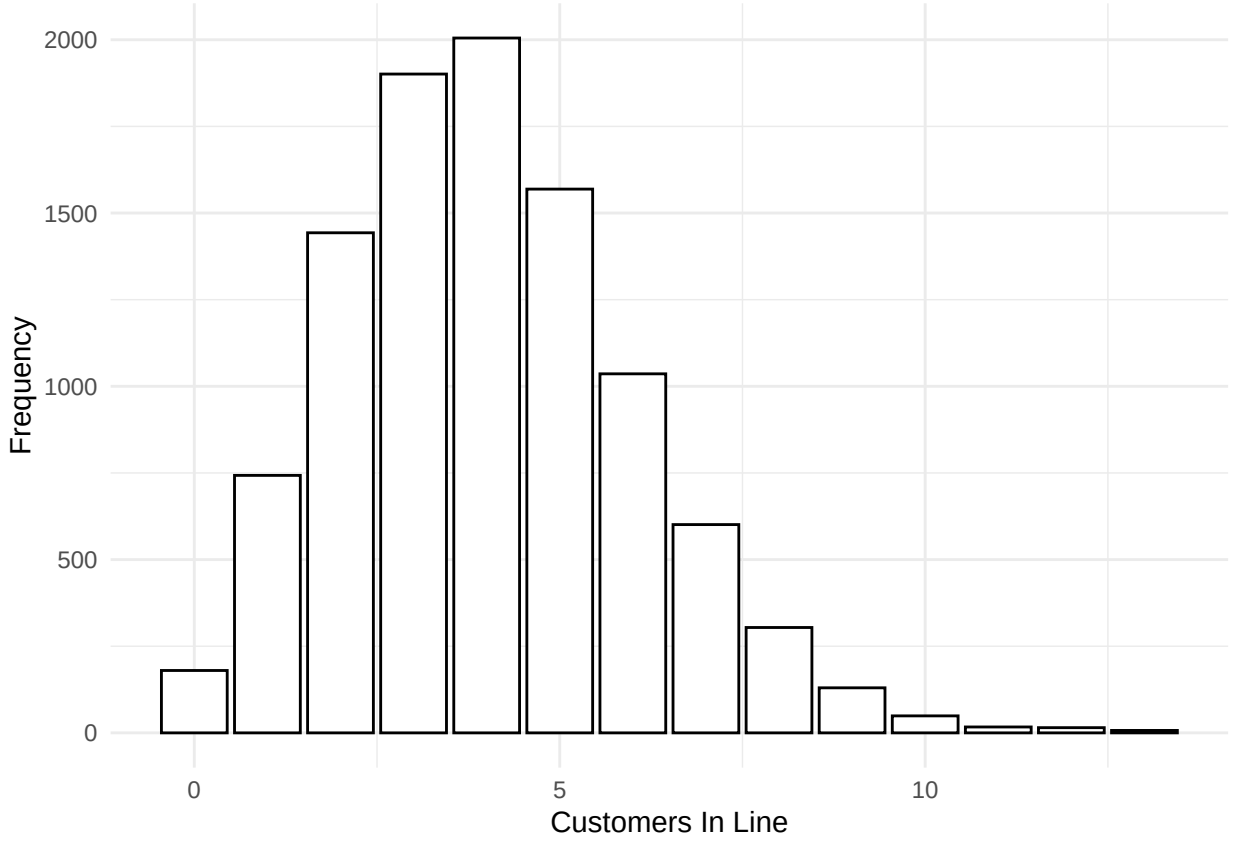


"They are really the same thing."

4.3.3 Poisson Distribution

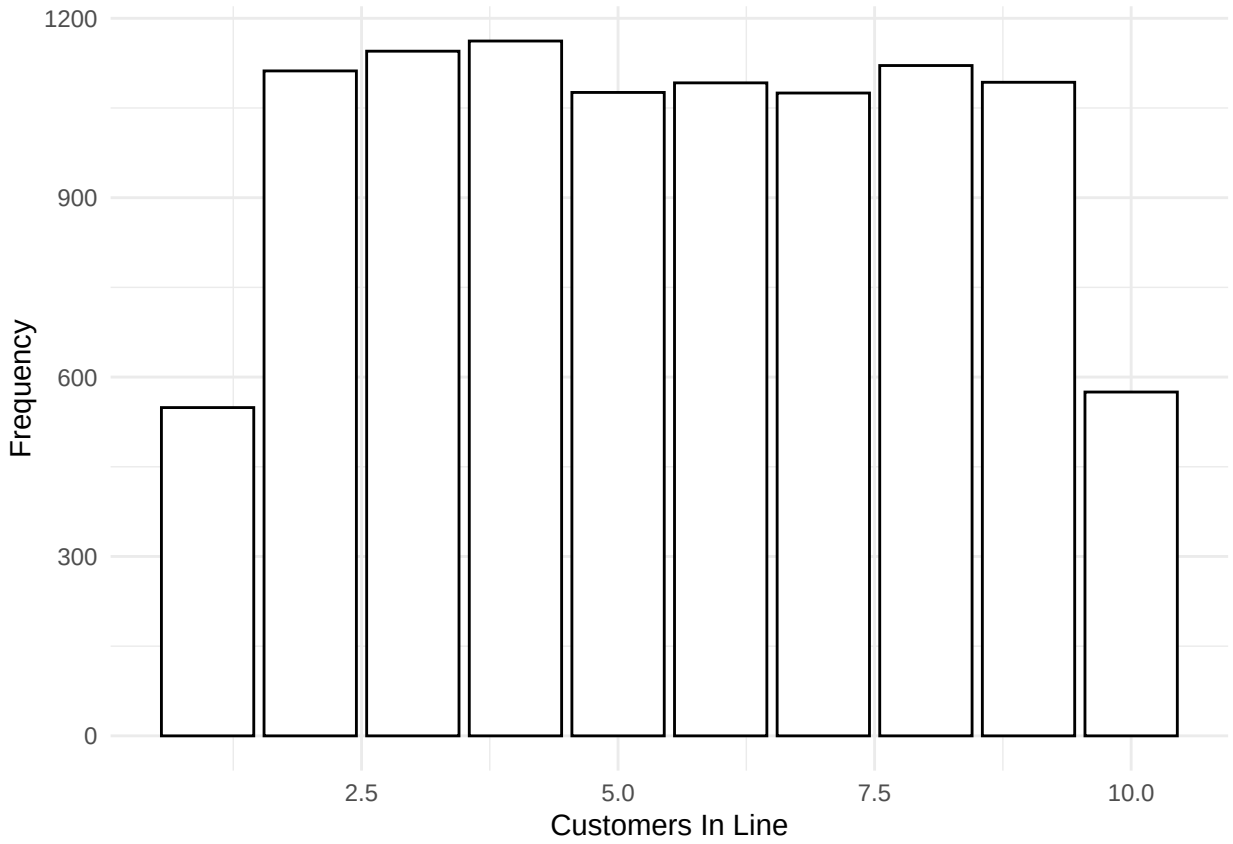
"The poisson is an interesting distribution – it tends to deal with count-related variables. It tells us the probability of a count occurring. We know its λ (again, a rate)."

"The λ is just a fancy way of saying the average number of events, or the incidence rate."



4.3.4 Uniform Distribution

"While people tend to think about the Gaussian distribution as the most vanilla of all distributions, it really is not – I would say that distinction belongs to the uniform distribution. We don't even get any fancy Greek letters, just a minimum and a maximum. Why? Because knowing the min and max will tell us that there is an equal probability of drawing a value anywhere within that range."



4.4 Performance

The *service level* for each simulation is the fraction of the demand that is satisfied.

$$Entrance\ Service\ Level = \frac{Objects\ Entering}{Objects\ Entering + Objects\ Unable\ To\ Enter}$$

Here, we are looking at the number of people who wanted to join the process, but could not. If we have a service level of 1, then 100% of objects were able to get into the process. A service level of .5 would indicate that only 50% of objects were able to enter.

The *overall mean service level* of the process is the mean of the service levels calculated from each simulation.

The *mean cycle time* at a buffer is the mean amount of time an object takes to move through the buffer during a simulation.

The *overall mean cycle time* at a buffer is the mean of the mean cycle time of the buffer for each simulation.

You will see different words for lines: buffers and queues. Just know that they are used interchangeably.

4.5 The Dispensary

The interarrival times for customers follows an exponential distribution with a rate of 1 person every 1.5 minutes.

The dispensary cannot hold any more than 8 people, for safety reasons. If a person arrives when the line is full, that person will not get in line.

The ID check is approximately normal with $\mu = 15\,seconds$ and $\sigma = 3\,seconds$. Once a person has their ID checked, they can sit in the lobby and there are 10 seats in the lobby.

The bud tender's service time is $\mu = 2.4\,minutes$ and $\sigma = .5\,minutes$.

Paying generally follows a uniform distribution, with a minimum of 5 seconds and a maximum of 15 seconds.

```
grViz("
digraph {
  graph [overlap = true, fontsize = 10, rankdir = LR]

  node [shape = box, style = filled, color = black, fillcolor = aliceblue]
  A [label = 'ID Check Line']
  B [label = 'ID Check']
  C [label = 'Bud Tender Line']
  D [label = 'Bud Tender']
  E [label = 'Pay']

  A->B B->C C->D D->E
}
")
```

4.5.1 ClassOverflow

We will need the `simmer` package for our simulation:

```
install.packages("simmer")
```

Once we have `simmer` installed, we need to load it:

```
library(simmer)
```

Let's start by defining a customer's trajectory. First, we will provide a name for `trajectory()`.

```
customer <- trajectory("Customer path")
```

Next, we need to initiate a start time with `set_attribute()` – we will use `now()` to specify our not-yet-created dispensary object.

```
customer <- trajectory("Customer path") |>
  set_attribute("start_time", function() {now(dispensary)})
```

After establishing our time, the next step for a customer is to `seize()` the "teller" (which we will define later).

```
customer <- trajectory("Customer path") |>
  set_attribute("start_time", function() {now(dispensary)}) |>
  seize("id_check")
```

Now things start to get tricky. We need to use `timeout()` to specify how long a customer is using the id check – this is the check's average working time.

We can specify how long an id check is seized (i.e., how long the check is working) – we provide a distribution with the appropriate values.

```
customer <- trajectory("Customer path") %>%
  set_attribute("start_time", function() {now(dispensary)}) |>
  seize("id_check") |>
  timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)})
```

After a customer spends time with the teller, the customer releases the counter.

```
customer <- trajectory("Customer path") %>%
  set_attribute("start_time", function() {now(dispensary)}) |>
  seize("id_check") |>
  timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
  release("id_check")
```

From there, we can add the additional resources to our model:

```
customer <- trajectory("Customer path") %>%
  set_attribute("start_time", function() {now(dispensary)}) |>
  seize("id_check") |>
  timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
  release("id_check") |>
  seize("bud_tender") |>
  timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
  release("bud_tender") |>
  seize("payment") |>
  timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
  release("payment")
```

This is all we need to do for a customer, so now we can turn our attention to the dispensary.

Our dispensary is going to provide the environment that houses our trajectory. So, we can start by creating an environment with `simmer()`:

```
dispensary <- simmer("dispensary")
```

Once we have our simulation environment defined, we can add resources to it with the aptly-named `add_resources()` function. This is where we will specify what is being seized by our customer. We need to provide some additional information to our resource: `capacity` and `queue_size`.

```
dispensary <- simmer("dispensary") |>
  add_resource("id_check", capacity = 1, queue_size = 8) |>
  add_resource("bud_tender", capacity = 2, queue_size = 10) |>
  add_resource("payment", capacity = 2)
```

To this point, we have our customer behavior (how they move through our process) and information about our work stations. The last detail is the inter-arrival time, which we can specify with `add_generator()`. It works in very much the same way that `timeout()`, in that we are specifying a distribution. The `rexp` function in R takes a rate. If we remember that, on average, one person comes into the dispensary every two minutes, we can define our rate as $\frac{1}{2}$.

Try this: mean(rexp(n = 10000, rate = 1/2))

```
dispensary <- simmer("dispensary") |>
  add_resource("id_check", capacity = 1, queue_size = 8) |>
  add_resource("bud_tender", capacity = 2, queue_size = 10) |>
  add_resource("payment", capacity = 2) |>
  add_generator("Customer", customer, function() {
    c(0, rexp(n = 100, rate = 1/1.5), -1)
  })
```

Now we can run our simulation; we just need to provide a time value for the `until` argument. Let's say we want to run this simulation for 2 hours.

```
run(dispensary, until = 120)
```

If we put it together, here is what we have:

```
customer <- trajectory("Customer path") %>%
  set_attribute("start_time", function() {now(dispensary)}) |>
  seize("id_check") |>
  timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
  release("id_check") |>
  seize("bud_tender") |>
  timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
  release("bud_tender") |>
  seize("payment") |>
  timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
  release("payment")

dispensary <- simmer("dispensary") |>
  add_resource("id_check", capacity = 1, queue_size = 8) |>
  add_resource("bud_tender", capacity = 2, queue_size = 10) |>
  add_resource("payment", capacity = 2) |>
  add_generator("Customer", customer, function() {
    c(0, rexp(n = 100, rate = 1/1.5), -1)
  })

simmer::run(dispensary, until = 120)
```

```
simmer environment: dispensary | now: 120 | next: 120.84365617859
{ Monitor: in memory }
{ Resource: id_check | monitored: TRUE | server status: 0(1) | queue status: 0(8) }
{ Resource: bud_tender | monitored: TRUE | server status: 0(2) | queue status: 0(10) }
{ Resource: payment | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
{ Source: Customer | monitored: 1 | n_generated: 101 }
```

Finally, we can start to look at our data:

```
result <- get_mon_arrivals(dispensary)

head(result)
```

```
       name start_time  end_time activity_time finished replication
1 Customer0   0.000000  2.697552      2.697552     TRUE           1
2 Customer1   1.570410  4.294481      2.724070     TRUE           1
3 Customer2   3.933397  6.976134      3.042737     TRUE           1
4 Customer3   4.298961  7.730810      3.431849     TRUE           1
5 Customer4   9.878084 13.641008      3.762924     TRUE           1
6 Customer5  12.540194 15.151201      2.611007     TRUE           1
```

Let's calculate a few things. First, let's how many people made it through:

```
nrow(result[result$finished == TRUE, ])
```

```
[1] 85
```

Now we can check our service level:

```
nrow(result[result$finished == TRUE, ]) / nrow(result)
```

```
[1] 1
```

The `nrow` function will tell us how many rows are in the data. In the numerator, we filtered those rows were finished was equal to TRUE (giving us the number of people who made it into the system).

Now we need to calculate how long each person was in line.

```
result$wait_time <- result$end_time - result$start_time - result$activity_time
```

Now, we can find the average wait time. We only want to do it for those who actually made it into the system though!

```
completeOnly <- result[result$finished == TRUE, ]

mean(completeOnly$wait_time)
```

```
[1] 3.817454
```

That gives us all of the information that we need for this dispensary configuration.

But... that is just one simulation. We really need to run this many times to get an idea about the distribution of outcomes.

We have a few choices. One choice is that we just replicate our procedure a certain number of times:

```
sim50Runs <- replicate(50, expr = {
  customer <- trajectory("Customer path") %>%
    set_attribute("start_time", function() {now(dispensary)}) |>
    seize("id_check") |>
    timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
    release("id_check") |>
    seize("bud_tender") |>
    timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
    release("bud_tender") |>
    seize("payment") |>
    timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
    release("payment")

  dispensary <- simmer("dispensary") |>
    add_resource("id_check", capacity = 1, queue_size = 8) |>
    add_resource("bud_tender", capacity = 2, queue_size = 10) |>
    add_resource("payment", capacity = 2) |>
    add_generator("Customer", customer, function() {
      c(0, rexp(n = 100, rate = 1/1.5), -1)
    })

  simmer::run(dispensary, until = 120)

  result <- get_mon_arrivals(dispensary)

}, simplify = FALSE)
```

We can extend this idea into something a bit more complex:

```
purrr::map_df(1:100, ~{
  customer <- trajectory("Customer path") %>%
    set_attribute("start_time", function() {now(dispensary)}) |>
    seize("id_check") |>
    timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
    release("id_check") |>
    seize("bud_tender") |>
    timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
    release("bud_tender") |>
```

```
    seize("payment") |>
    timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
    release("payment")

  dispensary <- simmer("dispensary") |>
    add_resource("id_check", capacity = 1, queue_size = 8) |>
    add_resource("bud_tender", capacity = 2, queue_size = 10) |>
    add_resource("payment", capacity = 2) |>
    add_generator("Customer", customer, function() {
      c(0, rexp(n = 100, rate = 1/1.5), -1)
    })

  simmer::run(dispensary, until = 120)

  result <- get_mon_arrivals(dispensary)

  result$run <- .x

  result
})
```

```
          name start_time   end_time activity_time finished replication run
1    Customer0   0.000000   2.517919      2.517919     TRUE           1   1
2    Customer2   2.391802   4.889366      2.497564     TRUE           1   1
3    Customer1   2.056590   5.748084      3.691494     TRUE           1   1
4    Customer3   4.777644   7.594081      2.816437     TRUE           1   1
5    Customer4   8.619556  11.439209      2.819653     TRUE           1   1
6    Customer5   8.870385  11.484980      2.549597     TRUE           1   1
7    Customer7  10.326079  13.337011      2.339395     TRUE           1   1
8    Customer6   9.212045  14.140854      3.110043     TRUE           1   1
9    Customer8  13.093383  17.099195      4.005812     TRUE           1   1
10   Customer9  14.181439  17.299386      3.117947     TRUE           1   1
11  Customer10  14.806823  19.217344      2.622868     TRUE           1   1
12  Customer11  15.916431  20.299764      3.274686     TRUE           1   1
13  Customer12  17.435814  21.690620      3.001048     TRUE           1   1
14  Customer13  23.090952  25.647531      2.556579     TRUE           1   1
15  Customer14  23.967826  26.734810      2.766984     TRUE           1   1
16  Customer15  24.756136  27.734450      2.506461     TRUE           1   1
17  Customer16  24.897015  28.178113      1.861899     TRUE           1   1
18  Customer18  25.927867  29.890360      2.122821     TRUE           1   1
19  Customer17  25.411131  30.200346      2.997034     TRUE           1   1
20  Customer19  26.711244  32.036761      2.673791     TRUE           1   1
21  Customer20  28.281161  32.762561      3.060201     TRUE           1   1
22  Customer21  28.895971  34.514335      2.972451     TRUE           1   1
23  Customer22  29.029609  34.655927      2.322018     TRUE           1   1
24  Customer23  30.094487  36.251005      2.172245     TRUE           1   1
25  Customer24  31.765812  36.823337      2.598897     TRUE           1   1
26  Customer25  32.751092  38.577855      2.744814     TRUE           1   1
27  Customer26  32.885852  39.227217      2.810723     TRUE           1   1
28  Customer27  33.081719  41.002669      2.856904     TRUE           1   1
29  Customer28  35.836650  41.517033      2.677681     TRUE           1   1
30  Customer29  37.675268  42.809176      2.245678     TRUE           1   1
31  Customer30  38.358458  43.952251      2.838248     TRUE           1   1
32  Customer31  38.588501  45.676118      3.392906     TRUE           1   1
```

33	Customer32	38.617271	46.552860	3.117738	TRUE	1	1
34	Customer33	42.665632	48.973201	3.869382	TRUE	1	1
35	Customer34	45.611595	49.183152	3.066589	TRUE	1	1
36	Customer36	48.394606	51.362488	2.643506	TRUE	1	1
37	Customer35	46.396039	51.368339	2.749032	TRUE	1	1
38	Customer37	48.603911	53.400046	2.593212	TRUE	1	1
39	Customer39	53.010828	54.868490	1.857662	TRUE	1	1
40	Customer38	51.951265	55.210735	3.259470	TRUE	1	1
41	Customer40	53.664858	56.469537	2.031890	TRUE	1	1
42	Customer41	53.940514	57.990110	3.300438	TRUE	1	1
43	Customer42	54.181671	58.710481	2.723133	TRUE	1	1
44	Customer43	56.335824	59.560362	1.933482	TRUE	1	1
45	Customer44	56.562583	61.412858	3.033548	TRUE	1	1
46	Customer45	60.440314	62.420248	1.979934	TRUE	1	1
47	Customer46	60.545993	64.100864	3.123065	TRUE	1	1
48	Customer47	61.002743	64.559726	2.535130	TRUE	1	1
49	Customer48	61.215522	66.458466	2.853272	TRUE	1	1
50	Customer49	61.991989	67.195283	2.958176	TRUE	1	1
51	Customer50	62.894419	68.281069	2.200741	TRUE	1	1
52	Customer51	63.195680	68.921004	2.185241	TRUE	1	1
53	Customer53	65.949447	71.183520	2.725354	TRUE	1	1
54	Customer52	63.833538	71.240630	3.403084	TRUE	1	1
55	Customer54	66.700056	73.228353	2.405080	TRUE	1	1
56	Customer55	68.224566	73.938408	2.993188	TRUE	1	1
57	Customer56	71.041170	75.757211	2.983506	TRUE	1	1
58	Customer57	71.242927	77.314544	3.742103	TRUE	1	1
59	Customer58	74.760795	78.368294	2.941379	TRUE	1	1
60	Customer59	76.696582	79.056711	2.223895	TRUE	1	1
61	Customer60	78.353684	81.065287	2.711603	TRUE	1	1
62	Customer61	81.495413	83.668820	2.173407	TRUE	1	1
63	Customer63	84.303041	86.911747	2.483208	TRUE	1	1
64	Customer62	84.143600	87.209568	3.065968	TRUE	1	1
65	Customer64	85.231000	88.857746	2.206904	TRUE	1	1
66	Customer65	85.298960	90.509111	3.734202	TRUE	1	1
67	Customer66	85.887583	92.382193	3.906745	TRUE	1	1
68	Customer67	92.164476	94.232074	2.067598	TRUE	1	1
69	Customer68	93.970179	97.121444	3.151266	TRUE	1	1
70	Customer69	95.086855	97.156429	2.069575	TRUE	1	1
71	Customer70	96.508986	99.486409	2.830302	TRUE	1	1
72	Customer71	97.752620	100.210828	2.458209	TRUE	1	1
73	Customer72	97.914228	101.726496	2.572188	TRUE	1	1
74	Customer73	98.395615	102.727295	2.954917	TRUE	1	1
75	Customer74	98.679470	104.026720	2.675566	TRUE	1	1
76	Customer75	101.092632	105.146659	2.758796	TRUE	1	1
77	Customer76	101.948921	106.202479	2.584796	TRUE	1	1
78	Customer78	106.123372	108.364100	2.240728	TRUE	1	1
79	Customer77	105.672270	108.625495	2.953225	TRUE	1	1
80	Customer79	106.923698	111.057852	3.141883	TRUE	1	1
81	Customer80	114.307387	116.785482	2.478095	TRUE	1	1
82	Customer81	114.558097	117.033270	2.475173	TRUE	1	1
83	Customer82	116.342318	119.250968	2.908650	TRUE	1	1
84	Customer0	0.000000	2.938786	2.938786	TRUE	1	2
85	Customer1	4.747101	8.110066	3.362964	TRUE	1	2
86	Customer2	6.369435	9.856833	3.487398	TRUE	1	2

87	Customer3	6.785318	11.281288	3.551750	TRUE	1	2
88	Customer4	7.941389	12.485964	2.978557	TRUE	1	2
89	Customer5	8.744060	13.404590	2.412298	TRUE	1	2
90	Customer6	8.981213	15.221019	3.195894	TRUE	1	2
91	Customer7	10.228239	15.619870	2.683081	TRUE	1	2
92	Customer8	10.833512	18.197808	3.389090	TRUE	1	2
93	Customer9	11.068130	18.213913	3.041646	TRUE	1	2
94	Customer10	15.020210	19.757265	2.019294	TRUE	1	2
95	Customer11	16.059310	20.472753	2.520753	TRUE	1	2
96	Customer12	16.355498	21.811186	2.494479	TRUE	1	2
97	Customer13	17.927817	23.264079	3.149958	TRUE	1	2
98	Customer14	20.479853	23.688285	2.344045	TRUE	1	2
99	Customer15	20.480455	25.958064	3.050562	TRUE	1	2
100	Customer16	22.588624	26.692652	3.339099	TRUE	1	2
101	Customer17	23.889068	28.562228	2.954405	TRUE	1	2
102	Customer18	24.893866	29.377618	3.163475	TRUE	1	2
103	Customer19	25.397790	30.019838	1.852826	TRUE	1	2
104	Customer20	27.328336	32.062741	3.105385	TRUE	1	2
105	Customer21	30.438134	32.667001	2.228867	TRUE	1	2
106	Customer23	34.503384	37.149547	2.608759	TRUE	1	2
107	Customer22	34.268311	37.588271	3.319960	TRUE	1	2
108	Customer24	35.556360	39.759686	3.066658	TRUE	1	2
109	Customer25	36.414954	41.062988	3.789153	TRUE	1	2
110	Customer26	36.686714	42.060794	2.796981	TRUE	1	2
111	Customer27	38.265236	43.966698	3.265701	TRUE	1	2
112	Customer28	39.977143	44.891054	3.192318	TRUE	1	2
113	Customer29	40.149318	46.301815	2.915314	TRUE	1	2
114	Customer30	41.432476	46.540788	2.109768	TRUE	1	2
115	Customer31	45.956673	48.605433	2.648760	TRUE	1	2
116	Customer32	47.293416	49.708781	2.415365	TRUE	1	2
117	Customer33	50.536973	53.494090	2.957117	TRUE	1	2
118	Customer35	53.719270	55.709477	1.990207	TRUE	1	2
119	Customer34	52.739863	56.314639	3.574776	TRUE	1	2
120	Customer36	53.729955	58.436464	3.138726	TRUE	1	2
121	Customer37	53.936882	58.439423	2.523811	TRUE	1	2
122	Customer38	54.370644	60.304550	2.284099	TRUE	1	2
123	Customer39	54.819103	60.782521	2.753823	TRUE	1	2
124	Customer41	56.777548	62.426223	2.031296	TRUE	1	2
125	Customer40	56.560274	62.677503	2.855098	TRUE	1	2
126	Customer43	58.374203	64.024516	1.674488	TRUE	1	2
127	Customer42	58.103787	65.204812	3.195748	TRUE	1	2
128	Customer44	61.959408	67.010737	3.302950	TRUE	1	2
129	Customer45	62.754448	67.542910	2.777262	TRUE	1	2
130	Customer47	63.679984	69.750052	2.667228	TRUE	1	2
131	Customer46	63.035314	69.964115	3.354332	TRUE	1	2
132	Customer48	64.133470	71.862085	2.533114	TRUE	1	2
133	Customer49	68.005321	72.502703	2.787598	TRUE	1	2
134	Customer51	69.976708	74.199893	2.187689	TRUE	1	2
135	Customer50	69.425835	74.574594	3.072368	TRUE	1	2
136	Customer52	75.374653	78.769682	3.395029	TRUE	1	2
137	Customer53	78.109770	80.529242	2.419472	TRUE	1	2
138	Customer54	79.774247	82.628857	2.854610	TRUE	1	2
139	Customer55	79.973801	83.486510	3.361120	TRUE	1	2
140	Customer56	80.537115	85.303366	3.041666	TRUE	1	2

```
141 Customer57  81.549156  85.320359      2.151661     TRUE           1   2
142 Customer58  82.593484  87.520535      2.613718     TRUE           1   2
 [ reached 'max' / getOption("max.print") -- omitted 7718 rows ]
```

Next, we can see what things might look like if we change parts of the process:

```
purrr::map2_df(.x = 1:100, .y = runif(100, min = 1, max = 2), ~{
    customer <- trajectory("Customer path") %>%
    set_attribute("start_time", function() {now(dispensary)}) |>
    seize("id_check") |>
    timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
    release("id_check") |>
    seize("bud_tender") |>
    timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
    release("bud_tender") |>
    seize("payment") |>
    timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
    release("payment")

  dispensary <- simmer("dispensary") |>
    add_resource("id_check", capacity = 1, queue_size = 8) |>
    add_resource("bud_tender", capacity = 2, queue_size = 10) |>
    add_resource("payment", capacity = 2) |>
    add_generator("Customer", customer, function() {
      c(0, rexp(n = 100, rate = 1/.y), -1)
    })

  simmer::run(dispensary, until = 120)

  result <- get_mon_arrivals(dispensary)

  result$run <- .x

  result$arrival <- .y

  result
})
```

```
          name   start_time   end_time activity_time finished replication run
1    Customer0   0.00000000   3.092014     3.0920145     TRUE           1   1
2    Customer1   0.02167911   3.294545     3.0916472     TRUE           1   1
3    Customer2   1.02110865   5.261299     2.5703333     TRUE           1   1
4    Customer3   3.29153289   5.689652     2.3981187     TRUE           1   1
5    Customer4   4.99737235   7.149883     2.1525111     TRUE           1   1
6    Customer5   6.94864559   9.716435     2.7677895     TRUE           1   1
7    Customer6   7.55729372  10.673184     3.1158901     TRUE           1   1
8    Customer7   8.96537684  11.820995     2.4698104     TRUE           1   1
9    Customer8   9.71335431  13.947218     3.5838826     TRUE           1   1
10   Customer9  12.23295992  14.543168     2.3102081     TRUE           1   1
11  Customer11  17.68871577  20.767998     2.9328871     TRUE           1   1
12  Customer10  17.63924085  21.541235     3.9019943     TRUE           1   1
13  Customer12  21.50887292  24.542224     3.0333508     TRUE           1   1
14  Customer13  22.58095058  24.872345     2.2913947     TRUE           1   1
15  Customer14  24.22295266  27.188333     2.9653800     TRUE           1   1
16  Customer15  25.56068720  28.428086     2.8673990     TRUE           1   1
```

17	Customer16	31.78503035	34.019869	2.2348391	TRUE	1	1
18	Customer17	33.87507529	36.749189	2.8741140	TRUE	1	1
19	Customer18	35.33870950	38.117312	2.7786022	TRUE	1	1
20	Customer19	37.71997279	39.693982	1.9740095	TRUE	1	1
21	Customer21	39.57411629	42.215117	2.6410009	TRUE	1	1
22	Customer20	39.15244768	42.467641	3.3151931	TRUE	1	1
23	Customer23	41.62012124	45.374854	3.1951877	TRUE	1	1
24	Customer22	39.64306735	45.609844	3.7425238	TRUE	1	1
25	Customer24	44.51849022	47.999688	3.0850496	TRUE	1	1
26	Customer25	45.99356808	48.365688	2.3721203	TRUE	1	1
27	Customer26	47.50265789	50.085805	2.4566298	TRUE	1	1
28	Customer27	48.50895848	51.986741	3.4777826	TRUE	1	1
29	Customer28	51.30854288	54.059567	2.7510238	TRUE	1	1
30	Customer29	52.63612407	55.019704	2.3835801	TRUE	1	1
31	Customer30	54.58587954	57.183388	2.5975089	TRUE	1	1
32	Customer31	54.99788880	58.232706	3.2348175	TRUE	1	1
33	Customer32	57.45870199	59.737566	2.2788639	TRUE	1	1
34	Customer33	57.47209219	60.941620	3.1119068	TRUE	1	1
35	Customer34	57.86619880	62.081705	2.7351814	TRUE	1	1
36	Customer35	57.93867436	63.572725	3.0695576	TRUE	1	1
37	Customer36	58.33579353	63.816719	2.1525433	TRUE	1	1
38	Customer37	61.92364598	65.497216	2.3553373	TRUE	1	1
39	Customer38	61.97487412	66.316499	2.9445774	TRUE	1	1
40	Customer39	62.48749100	67.673434	2.6305495	TRUE	1	1
41	Customer40	63.84401402	68.197143	2.3820251	TRUE	1	1
42	Customer41	63.89806683	70.168394	2.8711797	TRUE	1	1
43	Customer42	64.40478647	71.158309	3.3936234	TRUE	1	1
44	Customer43	64.82427214	72.683784	2.9287608	TRUE	1	1
45	Customer44	66.61210708	74.210739	3.5325844	TRUE	1	1
46	Customer45	67.50153415	75.330789	3.1187366	TRUE	1	1
47	Customer46	68.01040761	76.337743	2.5919771	TRUE	1	1
48	Customer47	70.20318448	77.308389	2.3552406	TRUE	1	1
49	Customer48	72.37396781	78.814094	2.9082223	TRUE	1	1
50	Customer49	73.09329510	79.882919	3.0262203	TRUE	1	1
51	Customer50	73.09896207	80.699311	2.3884951	TRUE	1	1
52	Customer51	74.50211879	82.627177	3.1249792	TRUE	1	1
53	Customer52	77.49431171	83.255462	3.0749869	TRUE	1	1
54	Customer53	77.51796360	85.056608	2.7693331	TRUE	1	1
55	Customer54	78.08391815	85.160265	2.4162182	TRUE	1	1
56	Customer56	79.11822517	87.423303	2.8215691	TRUE	1	1
57	Customer55	78.10772080	87.572539	2.9899411	TRUE	1	1
58	Customer58	80.38138227	90.136201	2.9919105	TRUE	1	1
59	Customer57	80.11466603	90.261792	3.2267946	TRUE	1	1
60	Customer59	82.71284101	92.190514	2.4086833	TRUE	1	1
61	Customer60	82.93177441	92.282357	2.4301966	TRUE	1	1
62	Customer62	84.84966594	94.107263	2.2338122	TRUE	1	1
63	Customer61	84.13480990	94.566240	2.8107173	TRUE	1	1
64	Customer63	85.60262204	95.871482	2.1821022	TRUE	1	1
65	Customer64	85.72275365	96.495941	2.2985461	TRUE	1	1
66	Customer65	86.26149430	98.568331	3.1073561	TRUE	1	1
67	Customer66	88.65798471	99.251816	3.0193875	TRUE	1	1
68	Customer79	98.94375239	99.455784	0.2737005	FALSE	1	1
69	Customer67	89.00412366	100.323124	2.1285099	TRUE	1	1
70	Customer68	90.13222729	101.472444	2.6240438	TRUE	1	1

71	Customer69	90.28480364	102.724908	2.8287059	TRUE	1	1
72	Customer70	91.74647685	103.506221	2.4831231	TRUE	1	1
73	Customer71	92.07634319	105.295681	2.9929956	TRUE	1	1
74	Customer72	92.85902779	106.222717	3.1554700	TRUE	1	1
75	Customer73	94.83030361	107.675617	2.8749868	TRUE	1	1
76	Customer74	95.81026114	107.787744	2.2142685	TRUE	1	1
77	Customer75	97.57634050	108.334600	1.0330029	TRUE	1	1
78	Customer76	97.66335506	110.017893	2.6846301	TRUE	1	1
79	Customer77	98.62150476	111.001373	3.1580922	TRUE	1	1
80	Customer78	98.89847350	112.628652	3.0407000	TRUE	1	1
81	Customer80	100.29417693	113.651977	3.2289362	TRUE	1	1
82	Customer82	107.51205684	115.444022	2.1374834	TRUE	1	1
83	Customer81	102.78867100	115.545281	3.3238912	TRUE	1	1
84	Customer83	107.67843625	117.606914	2.7226491	TRUE	1	1
85	Customer84	111.10913204	117.947164	2.7843006	TRUE	1	1
86	Customer85	112.86584203	119.050288	1.8872721	TRUE	1	1
87	Customer0	0.00000000	2.466134	2.4661342	TRUE	1	2
88	Customer1	0.97009477	3.833609	2.8635144	TRUE	1	2
89	Customer2	1.34910747	5.441126	3.3707208	TRUE	1	2
90	Customer3	1.77880324	6.482126	3.2497533	TRUE	1	2
91	Customer4	4.47666671	7.241000	2.2419095	TRUE	1	2
92	Customer5	5.36345673	9.001791	3.0078611	TRUE	1	2
93	Customer6	6.94177597	10.070343	3.1285671	TRUE	1	2
94	Customer8	8.34206692	11.566192	1.9361377	TRUE	1	2
95	Customer7	7.60537857	12.140200	3.5396232	TRUE	1	2
96	Customer10	9.30801794	14.040348	2.2364464	TRUE	1	2
97	Customer9	9.30552133	14.574826	3.3232686	TRUE	1	2
98	Customer12	10.49607625	16.857973	2.7837907	TRUE	1	2
99	Customer11	9.95157271	17.098938	3.4869126	TRUE	1	2
100	Customer14	12.54721109	19.438919	2.7146953	TRUE	1	2
101	Customer13	12.38252066	19.905336	3.4749167	TRUE	1	2
102	Customer15	12.61474735	21.599261	2.4369988	TRUE	1	2
103	Customer16	12.73037175	22.033801	2.7234686	TRUE	1	2
104	Customer17	17.98258653	23.631200	2.4054495	TRUE	1	2
105	Customer18	18.10333090	24.741764	3.1575036	TRUE	1	2
106	Customer19	18.44219370	25.928302	2.8014562	TRUE	1	2
107	Customer20	19.58327558	27.377587	3.0384530	TRUE	1	2
108	Customer21	19.88575257	28.238540	2.8341337	TRUE	1	2
109	Customer22	20.88935621	29.845382	2.9080887	TRUE	1	2
110	Customer23	22.81204246	30.122439	2.3758929	TRUE	1	2
111	Customer25	25.38710052	32.748317	3.0761962	TRUE	1	2
112	Customer24	24.05533891	33.162588	3.6641676	TRUE	1	2
113	Customer27	28.43788009	34.729585	2.0460589	TRUE	1	2
114	Customer26	28.08179132	36.040353	3.5772467	TRUE	1	2
115	Customer28	30.03139507	37.278240	3.0262753	TRUE	1	2
116	Customer29	30.15099103	38.645508	3.0503516	TRUE	1	2
117	Customer30	37.00393543	40.388523	3.3845876	TRUE	1	2
118	Customer31	46.20671973	48.740023	2.5333028	TRUE	1	2
119	Customer32	46.22811105	49.444000	2.9487223	TRUE	1	2
120	Customer33	46.81997918	51.566425	3.2722751	TRUE	1	2
121	Customer34	47.68394812	51.950729	2.9260537	TRUE	1	2
122	Customer35	47.92009977	53.774351	2.7168045	TRUE	1	2
123	Customer36	48.20263677	54.286013	2.8147295	TRUE	1	2
124	Customer37	49.19817037	56.375290	3.0577469	TRUE	1	2

```
125 Customer38  51.33023953  58.597815     4.7310608     TRUE           1    2
     arrival
1   1.325281
2   1.325281
3   1.325281
4   1.325281
5   1.325281
6   1.325281
7   1.325281
8   1.325281
9   1.325281
10  1.325281
11  1.325281
12  1.325281
13  1.325281
14  1.325281
15  1.325281
16  1.325281
17  1.325281
18  1.325281
19  1.325281
20  1.325281
21  1.325281
22  1.325281
23  1.325281
24  1.325281
25  1.325281
26  1.325281
27  1.325281
28  1.325281
29  1.325281
30  1.325281
31  1.325281
32  1.325281
33  1.325281
34  1.325281
35  1.325281
36  1.325281
37  1.325281
38  1.325281
39  1.325281
40  1.325281
41  1.325281
42  1.325281
43  1.325281
44  1.325281
45  1.325281
46  1.325281
47  1.325281
48  1.325281
49  1.325281
50  1.325281
51  1.325281
52  1.325281
```

```
53  1.325281
54  1.325281
55  1.325281
56  1.325281
57  1.325281
58  1.325281
59  1.325281
60  1.325281
61  1.325281
62  1.325281
63  1.325281
64  1.325281
65  1.325281
66  1.325281
67  1.325281
68  1.325281
69  1.325281
70  1.325281
71  1.325281
72  1.325281
73  1.325281
74  1.325281
75  1.325281
76  1.325281
77  1.325281
78  1.325281
79  1.325281
80  1.325281
81  1.325281
82  1.325281
83  1.325281
84  1.325281
85  1.325281
86  1.325281
87  1.690627
88  1.690627
89  1.690627
90  1.690627
91  1.690627
92  1.690627
93  1.690627
94  1.690627
95  1.690627
96  1.690627
97  1.690627
98  1.690627
99  1.690627
100 1.690627
101 1.690627
102 1.690627
103 1.690627
104 1.690627
105 1.690627
106 1.690627
```

```
107 1.690627
108 1.690627
109 1.690627
110 1.690627
111 1.690627
112 1.690627
113 1.690627
114 1.690627
115 1.690627
116 1.690627
117 1.690627
118 1.690627
119 1.690627
120 1.690627
121 1.690627
122 1.690627
123 1.690627
124 1.690627
125 1.690627
 [ reached 'max' / getOption("max.print") -- omitted 7834 rows ]
```

And a function to show:

```
employee_change_mod <- purrr::map2_df(.x = 1:100, .y = sample(1:3, 100, TRUE, c(.1, .7, .2)), ~{
  customer <- trajectory("Customer path") %>%
    set_attribute("start_time", function() {now(dispensary)}) |>
    seize("id_check") |>
    timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
    release("id_check") |>
    branch(function() {sample(1:2, 1, prob = c(.8, .2))},
           # The "continue" indicates if the branches should continue through
           # the trajectory once the branch trajectory is completed.
           continue = c(TRUE, TRUE),
           trajectory() %>%
             seize("bud_tender") |>
             timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
             release("bud_tender"),
           trajectory() |>
             seize("devices") |>
             timeout(function() {rnorm(1, 5, 1)}) |>
             release("devices")) |>
    seize("payment") |>
    timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
    release("payment")

  dispensary <- simmer("dispensary") |>
    add_resource("id_check", capacity = 1, queue_size = 8) |>
    add_resource("bud_tender", capacity = .y, queue_size = 10) |>
    add_resource("devices", capacity = 1, queue_size = 10) |>
    add_resource("payment", capacity = 2) |>
    add_generator("Customer", customer, function() {
      c(0, rexp(n = 100, rate = 1/.y), -1)
    })

  simmer::run(dispensary, until = 120)
```

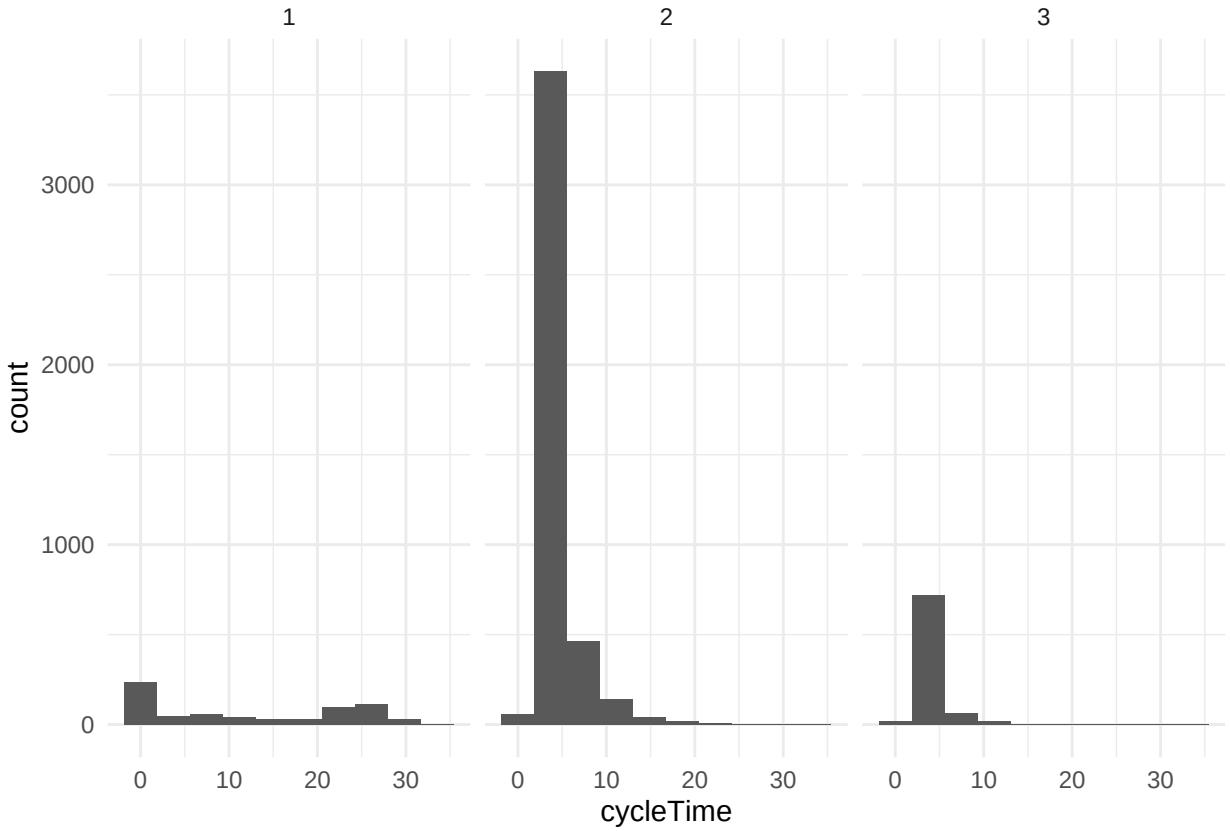
```
  result <- get_mon_arrivals(dispensary)

  result$run <- .x

  result$employees <- .y

  result
})

sapply(split(employee_change_mod, employee_change_mod$employees), function(x) {
  finished <- x[x$finished == TRUE, ]
  nrow(finished) / nrow(x)
})
```

```
        1         2         3
0.6559297 1.0000000 1.0000000
```

```
employee_change_mod$cycleTime <- employee_change_mod$end_time - employee_change_mod$start_time

aggregate(employee_change_mod$cycleTime,
          by = list(employee_change_mod$employees),
          mean)
```

```
  Group.1         x
1       1 12.051752
2       2  4.105586
3       3  3.579114
```

```
library(ggplot2)

ggplot(employee_change_mod, aes(cycleTime)) +
  geom_histogram(bins = 10) +
  facet_wrap(vars(employees)) +
  theme_minimal()
```



4.5.2 Something Silly

```
library(parallel)

cl <- makeCluster(detectCores() - 1)

clusterEvalQ(cl, library(simmer))
```

```
[[1]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[2]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[3]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[4]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[5]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

```
[[6]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"

[[7]]
[1] "simmer"    "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
```

```
allResults = parLapply(cl, seq(1:50), function(the_seed) {
 customer <- trajectory("Customer path") %>%
    set_attribute("start_time", function() {now(dispensary)}) |>
    seize("id_check") |>
    timeout(function() {rnorm(n = 1, mean = 15/60, sd = 3/60)}) |>
    release("id_check") |>
    seize("bud_tender") |>
    timeout(function() {rnorm(n = 1, mean = 2.4, sd = .5)}) |>
    release("bud_tender") |>
    seize("payment") |>
    timeout(function() {runif(n = 1, min = 5/60, max = 15/60)}) |>
    release("payment")

  dispensary <- simmer("dispensary") |>
    add_resource("id_check", capacity = 1, queue_size = 8) |>
    add_resource("bud_tender", capacity = 2, queue_size = 10) |>
    add_resource("payment", capacity = 2) |>
    add_generator("Customer", customer, function() {
      c(0, rexp(n = 100, rate = 1/1.5), -1)
    })

  simmer::run(dispensary, until = 120)

  result <- get_mon_arrivals(dispensary)

  result$run <- the_seed

  result

})

stopCluster(cl)
```

4.6 Manufacturing

In manufacturing, one of the more important metrics we deal with is *throughput* (the total number of units produced).

Variability, while not something we can always measure/predict, can come from a few sources.

- Machine/worker processing times
- Output quality
- Demand
- Supply

Controlling variability can help to increase throughput.

4.6.1 An Example

- In a factory, we have five work stations arranged in a line (WS1 through WS5).
- Each work station is a machine with one operator.
  - WS1 (bandsaw): $\mu = 10; \sigma = 1$ (normal)
  - WS2 (contouring): $\mu = 5; \sigma = 2$ (normal)
  - WS3 (rough sanding): $min = 5; max = 15$ (uniform)
  - WS4 (finish sanding): $min = 10; max = 15$ (uniform)
  - WS5 (finish): $\mu = 10; \sigma = 2.5$ (normal)
- The work stations process one product that must move sequentially.
- Each work station has its own processing time.
- Product cannot be *stacked*
  - If WS3 has finished a unit, it cannot be passed onto WS4 if WS4 is still working on a product.
- We are looking at an 8 hour shift.

4.6.2 Improvements

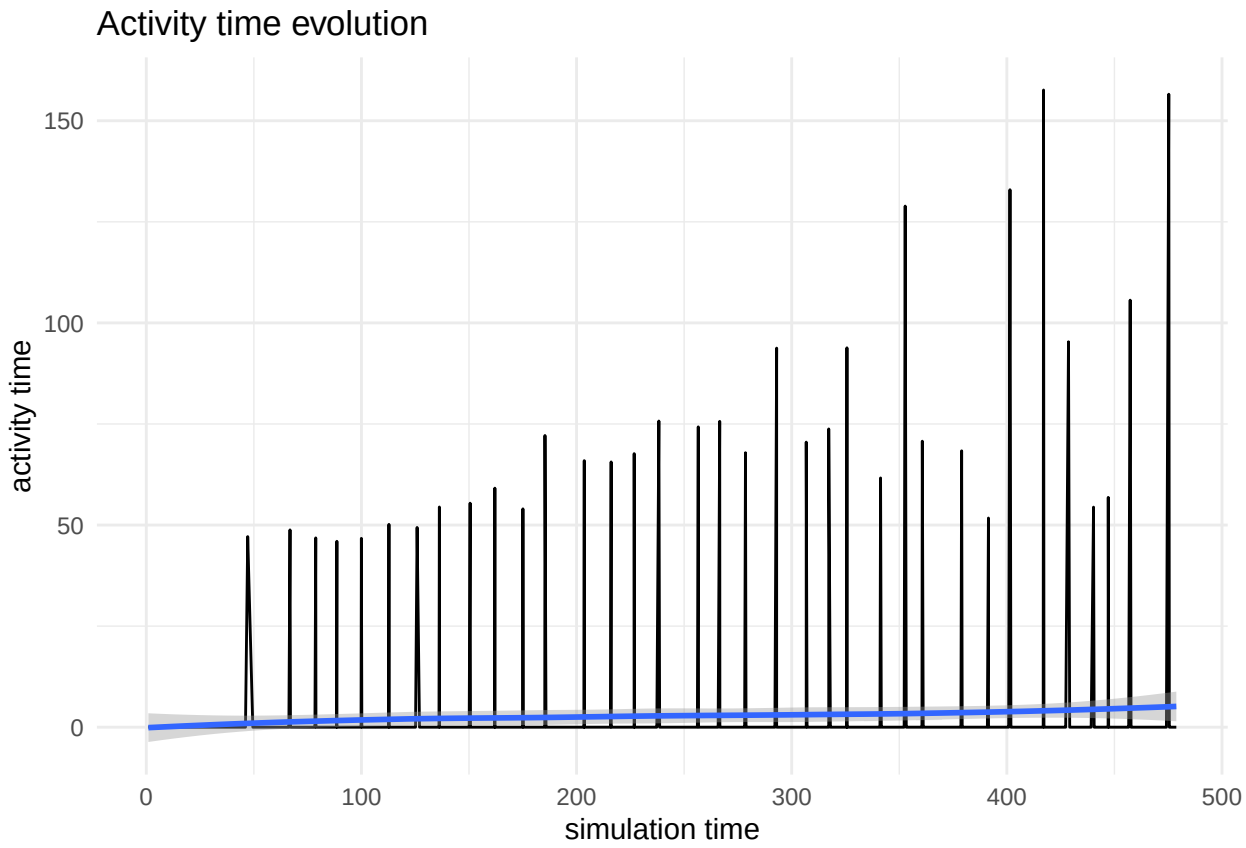
What can we do to improve our throughput?

```
library(simmer.plot)
make_parts <- trajectory("parts") %>%
  set_attribute("start_time", function() {now(machineShop)}) %>%
  seize("machine1") %>%
  timeout(function() {rnorm(n = 1, mean = 10, sd = 1)}) %>%
  release("machine1") %>%
  seize("machine2", continue = FALSE,
        reject = trajectory() %>%
          timeout(1) %>%
          rollback(amount = 2, times = Inf)) %>%
  timeout(function() {rnorm(n = 1, mean = 5, sd = 2)}) %>%
  release("machine2") %>%
  seize("machine3", continue = FALSE,
        reject = trajectory() %>%
          timeout(1) %>%
          rollback(amount = 2, times = Inf)) %>%
  timeout(function() {runif(n = 1, min = 5, max = 15)}) %>%
  release("machine3") %>%
  seize("machine4", continue = FALSE,
        reject = trajectory() %>%
          timeout(1) %>%
          rollback(amount = 2, times = Inf)) %>%
  timeout(function() {runif(n = 1, min = 10, max = 15)}) %>%
  release("machine4") %>%
  seize("machine5", continue = FALSE,
        reject = trajectory() %>%
          timeout(1) %>%
          rollback(amount = 2, times = Inf)) %>%
```

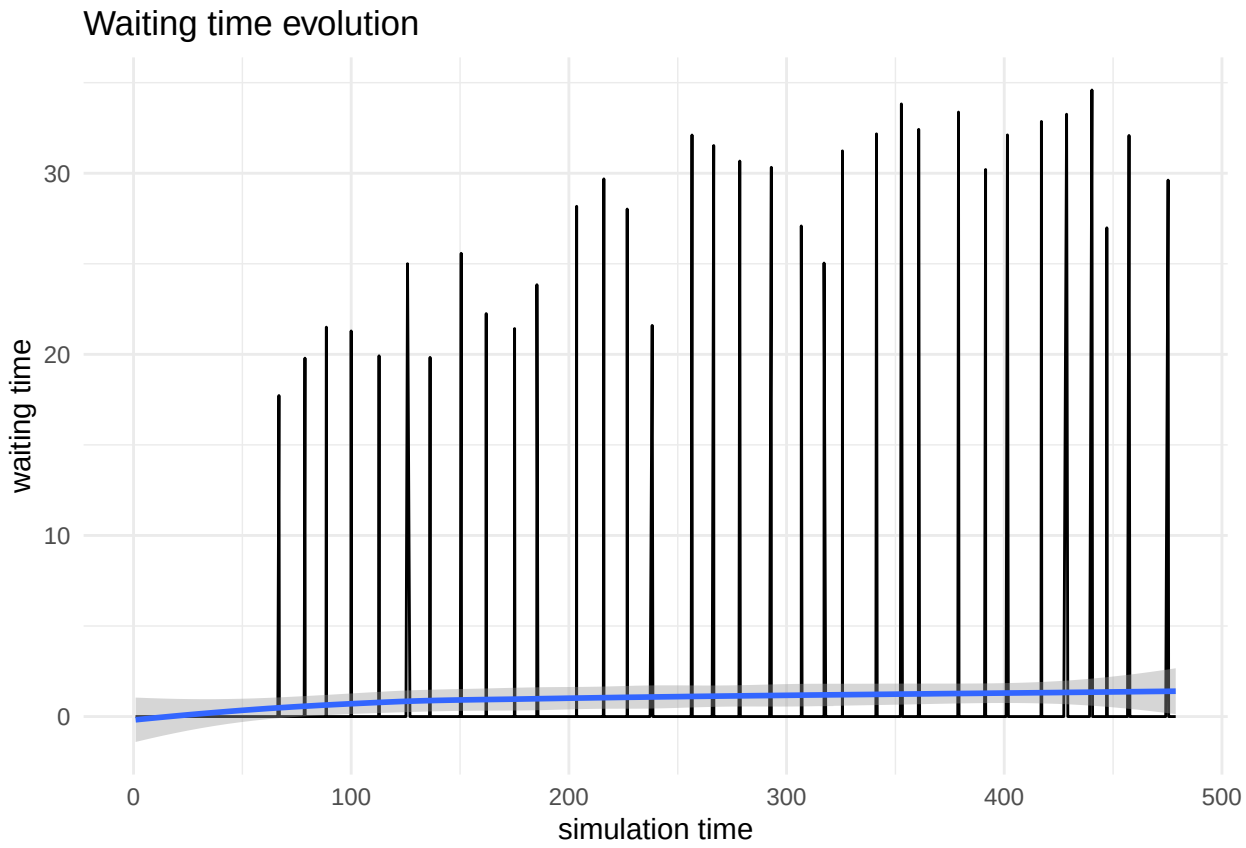
```
  timeout(function() {rnorm(n = 1, mean = 10, sd = 2.5)}) %>%
  release("machine5")

machineShop <- simmer("machineShop") %>%
  add_resource("machine1", capacity = 1, queue_size = 1) %>%
  add_resource("machine2", capacity = 1, queue_size = 1) %>%
  add_resource("machine3", capacity = 1, queue_size = 1) %>%
  add_resource("machine4", capacity = 1, queue_size = 1) %>%
  add_resource("machine5", capacity = 1, queue_size = 1) %>%
  add_generator("part", make_parts, mon = 1, function() {c(0, rexp(1000, 1/.5), -1)})

simmer::run(machineShop, 480)
```

```
simmer environment: machineShop | now: 480 | next: 480.039052840364
{ Monitor: in memory }
{ Resource: machine1 | monitored: TRUE | server status: 1(1) | queue status: 1(1) }
{ Resource: machine2 | monitored: TRUE | server status: 0(1) | queue status: 0(1) }
{ Resource: machine3 | monitored: TRUE | server status: 1(1) | queue status: 1(1) }
{ Resource: machine4 | monitored: TRUE | server status: 1(1) | queue status: 1(1) }
{ Resource: machine5 | monitored: TRUE | server status: 1(1) | queue status: 0(1) }
{ Source: part | monitored: 1 | n_generated: 1001 }
```

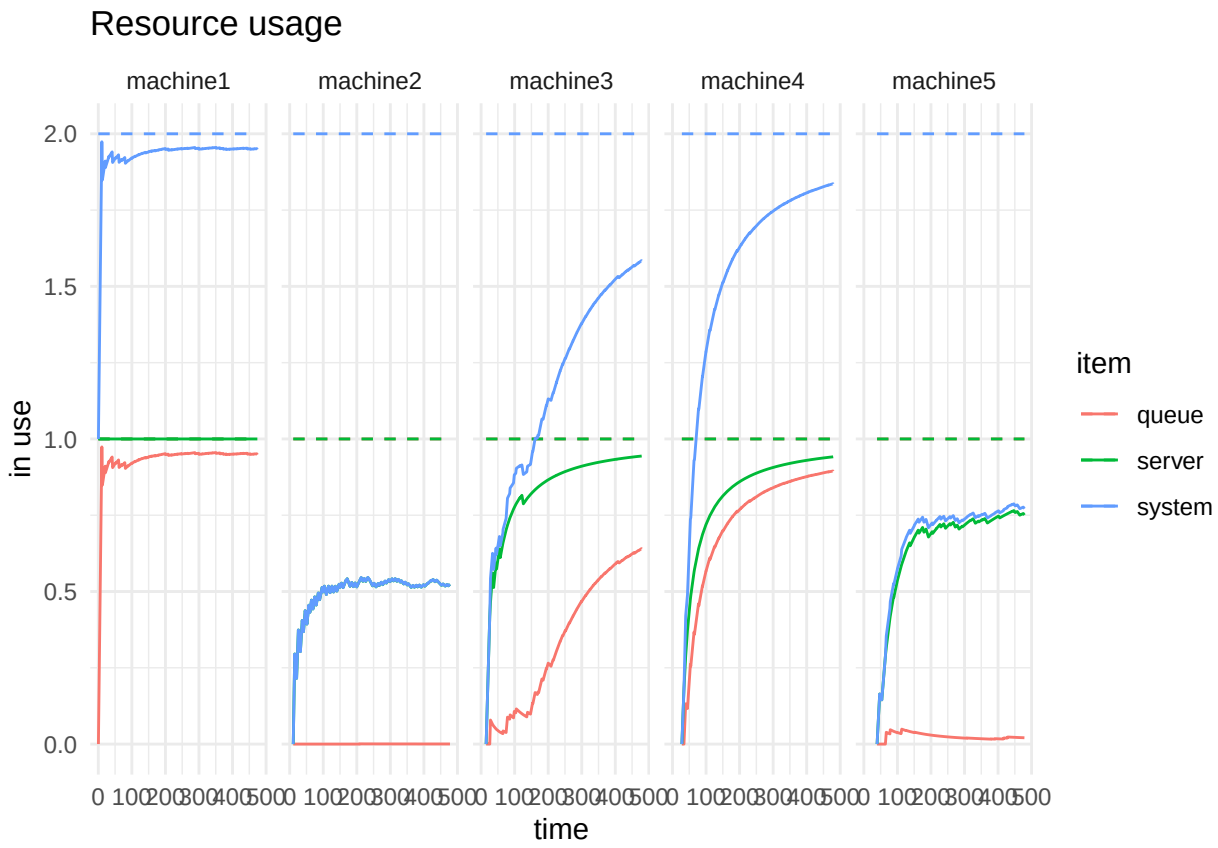
```
result <- get_mon_arrivals(machineShop)

resourceResult <- get_mon_resources(machineShop)

plot(result, metric = "activity_time") +
  theme_minimal()
```



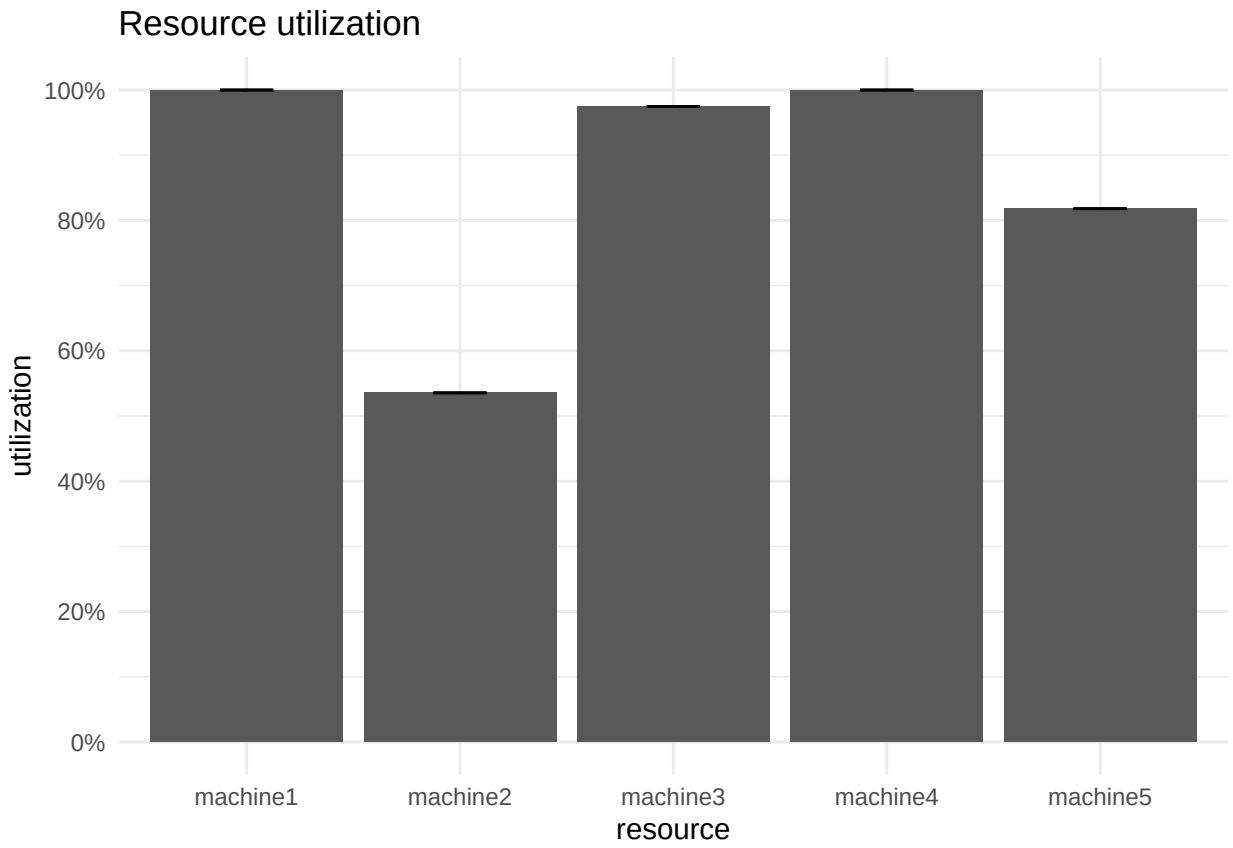

```
plot(result, metric = "waiting_time") +
  theme_minimal()
```




```
plot(resourceResult, metric="usage") +
  theme_minimal()
```




```
plot(resourceResult, metric="utilization") +
  theme_minimal()
```




```
sum(result$finished[result$finished == TRUE])
```

```
[1] 35
```

Chapter 5

Integer Programming

5.1 Troubling Solutions

Ali had figured out the whole linear programming thing and all felt good – until a troubling solution came back.

As per usual, trouble began with an email:

> What's good Ali? I'm Bao, from our retail operations group! We heard that you are a wizard with this stuff, so we are hoping that you can help us out.
>
> We have an issue in many of our stores: we often find that we are overstocked on certain edibles, but we can't keep others on the shelf. We'd love to balance that out, but aren't really sure if saying, "Just change production", is the answer. No matter what we make for batches, we would obviously like to do it for as cheap as possible.
>
> Here is the basics of what we have to deal with:
>
> 1. We have two primary classes of edibles: gummies and candy bars.
> 2. We aren't really worried about ingredients other than what is grown in our greenhouses. Food supplies are easy, but green supplies aren't.
> 3. To make a batch of candy, it requires 4 grams of raw flower, 1 gram of distillates, and 1 gram of pressed trichromes.
> 4. To make a batch of gummies, it requires 3 grams of raw flower and 1 gram of distillates.
> 5. Every month, we start with 200 grams of raw flower, 500 grams of distallate, and 100 grams of pressed trichromes.
> 6. If a store needs more, they can purchase a standard bag of raw flower for 80 dollars per bag.
> 7. A bag of raw flower can produce 10 grams of distallate, 20 grams of raw flower, and 2 grams of pressed trichromes
> 8. Producing candy costs 30 dollars per batch; producing gummies costs 40 dollars a batch
> 9. We need at least 1000 selling units per month.
>
> We owe you one!
>
> Bao

"Oh yeah", though Ali, "this is going to be a walk in the park."

```
library(linprog)

# Remember...the c vector is the top part of the whole problem:
# 30candy + 40gummies + 80bag

cvec <- c(candy = 30,
          gummies = 40,
          bag = 80)

# The b vector is the margins of the problem:
# What comes on the right hand side of the
# equality sign.

bvec <- c(distillate = 500,
          flower = 200,
          trichromes = 100,
          batch_need = 1000)

# These are the directions -- hopefully not
# much of a mystery.

constDirs <- c("<=", "<=", "<=", ">=")

# The a matrix comprises all of the rows of
# the constraint matrix (just not the margins
# or the directions).

# 1candy + 1gummies - 10bag
# 4candy + 3gummies - 20bag
# 1candy + 0gummies - 2bag
# 1candy + 1gummies + 0bag

aMat <- rbind(dis_const = c(1, 1, -10),
              flow_const = c(4, 3, -20),
              trich_const = c(1, 0, -2),
              batch_const = c(1, 1, 0))

# All together, we have:
# 30candy + 40gummies + 80bag
# 1candy + 1gummies - 10bag <= 500
# 4candy + 3gummies - 20bag <= 200
# 1candy + 0gummies - 2bag <= 100
# 1candy + 1gummies + 0bag >= 1000

solveLP(cvec, bvec, aMat, maximum = FALSE,
        const.dir = constDirs)
```

```
Results of Linear Programming / Linear Optimization

Objective function (Minimum): 48666.7

Iterations in phase 1: 2
Iterations in phase 2: 1
```

```
Solution
            opt
candy   422.222
gummies 577.778
bag     161.111

Basic Variables
                  opt
candy         422.222
gummies       577.778
bag           161.111
S distillate 1111.111

Constraints
             actual dir bvec    free     dual dual.reg
distillate -611.111  <=  500 1111.11  0.00000  1111.11
flower      200.000  <=  200    0.00  3.33333  5200.00
trichromes  100.000  <=  100    0.00  6.66667   380.00
batch_need 1000.000  >= 1000    0.00 50.00000      Inf

All Variables (including slack variables)
                  opt cvec      min.c max.c     marg marg.reg
candy         422.222   30  -60.00000    36       NA       NA
gummies       577.778   40  -46.00000    70       NA       NA
bag           161.111   80 -140.00000   200       NA       NA
S distillate 1111.111    0   -6.00000    12  0.00000       NA
S flower       0.000     0   -3.33333   Inf  3.33333     5200
S trichromes   0.000     0   -6.66667   Inf  6.66667      380
S batch_need   0.000     0  -50.00000   Inf 50.00000      Inf
```

That is great! Make 422.222 batches of candy, 577.778 batches of gummies, and buy 161.111 bags...

The first thing Ali though was, "What in the actual... those numbers cannot be correct. How do you make .222 of something? Is that acceptable? On second thought, could you ever buy a tenth of a bag?"

Just to be sure, Ali checked every value and everything matched just fine. So what could be causing the problem and how can it be fixed?

Looks like another visit to Jun... who just so happens to be on vacation.

5.2 ClassOverflow

Let's spend some time exploring some package functionality:

```
library(lpSolve)

test <- lpSolve::lp(direction = "min", objective.in = cvec,
            const.mat = aMat, const.dir = constDirs,
            const.rhs = bvec, all.int = TRUE)
```

Remember ROI from last time? It has some pretty handy functionality.

```
library(ROI)
library(ROI.plugin.glpk)
```

We can use all of the objects that we have already created

```
# When working within ROI, we string together the entire constraint
# matrix (A matrix, directions, and the b vector) using the L_constraint
# function

model_constraints <- L_constraint(L = aMat,
                                  dir = constDirs,
                                  rhs = bvec)

# After we string those together, we can throw them OP;
# this just creates the model, but doesn't actually
# solve the problem.
# The big difference below is in the types column;
# this changes it from continuous ("C") to
# integer ("I").

model_creation <- OP(objective = cvec,
                     constraints = model_constraints,
                     types = rep("I", length(cvec)),
                     maximum = FALSE)

model_solved <- ROI_solve(model_creation)

solution(model_solved, "primal")
```

```
  candy gummies     bag
    420     580     161
```

```
solution(model_solved, "objval")
```

```
[1] 48680
```

Now, Ali knows to make 420 units of candy (coincidence?), 580 units of gummies, and 161 bags.

All we have done in either case is to constrain the possible solution set to only include integer values. The "how" of this is substantially more complicated.

What follows is purely a cursory glance – if you are interested in knowing more, you can check out the resources!

1. The most naive approach is to solve as a linear programming problem (i.e., LP relaxation) and then round the results
2. Some matrices are *totally unimodular* (no big concern for us) and will always return an integer value.
3. Cutting planes solve the LP relaxation, test if the optimal value is integer. A non-integer solution will get reworked as a constraint and this process continues until an optimal integer solution is found.
4. Branch and bound algorithms produces possible solution sets and transverses subsets of those sets to find an answer.
5. Branch and cut algorithms combine the previous 2 approaches.

5.3 Transportation Problems

After sending an updated solution to Bao, Ali felt some sense of relief – learning was happening and solutions were getting easier to come by. Unfortunately for Ali, stories of success spread wildly and it wasn't long until more people came knocking. The first request was from Castel, the director of grow operations; the second request was from Blaise, the director of Human Resources.

Castel's problem seemed pretty simple: there are greenhouses that contain raw product and that raw product needs to be shipped to different processing facilities. Each processing facility has a capacity need and there is a cost for moving products between the different facilities:

Castel drew a map for the cost to move product between greenhouses and processing facilities:

And then added the following notes:

Greenhouse 1 can only contribute 1200 pounds

Greenhouse 2 can only contribute 1000 pounds

Greenhouse 3 can only contribute 800 pounds

Processing facility 1 needs at least 1100 pounds

Processing facility 2 needs at least 400 pounds

Processing facility 3 needs at least 750 pounds

Processing facility 4 needs at least 750 pounds

How much should each greenhouse send to each processing facility and do it as cheaply as possible?

Ali had all of the necessary information, so took Castel's words and translated them into an expression:

$$\begin{aligned} M = 35_{x_{11}} + 30_{x_{12}} + 40_{x_{13}} + 32_{x_{14}} + 37_{x_{21}} + 40_{x_{22}} + \\ 42_{x_{23}} + 25_{x_{24}} + 40_{x_{31}} + 15_{x_{32}} + 20_{x_{33}} + 28_{x_{34}} \\ subject\ to \\ X_{11} + X_{12} + X_{13} + X_{14} \leq 1200 \\ X_{21} + X_{22} + X_{23} + X_{24} \leq 1000 \\ X_{31} + X_{32} + X_{33} + X_{34} \leq 800 \\ X_{11} + X_{21} + X_{31} \geq 1100 \\ X_{12} + X_{22} + X_{23} \geq 400 \\ X_{13} + X_{23} + X_{33} \geq 750 \\ X_{14} + X_{24} + X_{34} \geq 750 \\ X_{ij} \geq 0 \end{aligned}$$

```
cMat <- c(35, 30, 40, 32, 37, 40,
     42, 25, 40, 15, 20, 28)

b <- c(1200, 1000, 800, 1100, 400, 750, 750)

A <- rbind(c(1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0),
           c(0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0),
           c(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1),
           c(1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0),
           c(0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0),
           c(0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0),
           c(0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1))

# Remember that the rep function below only serves to replicate whatever
# you put there. So this:
# rep("<=", 12)
# Is the same as:
# c("<=", "<=", "<=", "<=", "<=", "<=", "<=", "<=", "<=", "<=", "<=", "<=")
# One is clearly an easier thing to code.
```

```
constraints <- L_constraint(A,
                            c(rep("<=", 3), rep(">=", 4)),
                            b)

# The length function returns how many items are in the vector.

model <- OP(objective = cMat,
            constraints = constraints,
            types = rep.int("I", length(cMat)),
            maximum = FALSE)

result <- ROI::ROI_solve(model, "glpk", verbose = TRUE)
```

```
<SOLVER MSG>  ----
GLPK Simplex Optimizer, v4.65
7 rows, 12 columns, 24 non-zeros
      0: obj =   0.000000000e+00 inf =   3.000e+03 (4)
      6: obj =   1.049000000e+05 inf =   0.000e+00 (0)
*    10: obj =   8.400000000e+04 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
GLPK Integer Optimizer, v4.65
7 rows, 12 columns, 24 non-zeros
12 integer variables, none of which are binary
Integer optimization begins...
Long-step dual simplex will be used
+    10: mip =     not found yet >=              -inf        (1; 0)
+    10: >>>>>   8.400000000e+04 >=   8.400000000e+04   0.0% (1; 0)
+    10: mip =   8.400000000e+04 >=     tree is empty   0.0% (0; 1)
INTEGER OPTIMAL SOLUTION FOUND
<!SOLVER MSG> ----
```

```
result$objval
```

```
[1] 84000
```

```
transportation_solution <- solution(result)

# Just setting names to make life easier.

names(transportation_solution) <- c("g1_p1", "g1_p2", "g1_p3", "g1_p4",
                                    "g2_p1", "g2_p2", "g2_p3", "g2_p4",
                                    "g3_p1", "g3_p2", "g3_p3", "g3_p4")
```

“That’s a nice solution!”, Ali remarked and then emailed Castel the results:

> Hey Castel,
>
> From greenhouse 1, send 850 pounds to processing 1 and 350 pounds to processing 2.
>
> From greenhouse 2, send 250 pounds to processing 1 and 750 pounds to processing 4.
>
> From greenhouse 3, send 50 pounds to processing 50 and 750 to processing 3.
>
> All those moves will cost 84000 dollars.
>
> Let me know if I can help you with anything else,
>
> Ali

5.4 Binary Integer Programming

The *transportation problem* proved to be an easy one, once it was broken down into it's mathematical expression. Blaise's request, though, proved to be a bit more challenging.

> How's life, my unmet friend?
>
> We've got a situation in our *connoisseur's cabinet* – it is where we keep the expensive stuff.
>
> We only have room for a single feature cabinet, so we can't put everything that we have into it.
>
> I'd like to put things in there that won't take up a ton of space, but will also bring in the cash.
>
> The list of products, prices, and space is attached. I can't use any more than 10 spaces.
>
> Any ideas?
>
> Blaise

Turns out, that Blaise was just asking for some version of a *knapsack* problem:

```
# Nothing tricky below -- all we are doing is creating a data frame.
# The data frame contains 3 columns: item, space, and value.

special_items <- data.frame(item = c("cannabis_caviar", "oracle", "fruity_pebbles",
                                     "loud_dream", "white_fire", "j1",
                                     "hammerhead", "sista", "goblin",
                                     "fishermen", "cloud", "paradise"),
                            space = c(1.5, 1, 1,
                                      1.25, 1, 1,
                                      6, 5, 6,
                                      7, 3, 2),
                            value = c(800, 450, 400,
                                      400, 500, 350,
                                      1600, 2325, 1005,
                                      750, 250, 875))

# Below is just a little bit different from what we have seen:
# instead of specifying a whole vector on the RHS, we just
# have a single number. We are really only rocking with
# a single constraint.

constraints <- L_constraint(special_items$space, "<=", 10)

# Going to set those variables to be integers.

model <- OP(objective = special_items$value,
            constraints = constraints,
            types = rep.int("I", 12),
            maximum = TRUE)

solved_model <- ROI_solve(model)

# The line below looks weird, but we are literally saying
# to take our solution and then set the names of that
# data frame to the item names from our special_items
# data frame. The |> is R's native pipe operator.
# The only reason we are doing this is to make the solution
# easier to see (i.e., which itmes should we select).
```

```
solution(solved_model) |>
  setNames(special_items$item)
```

```
cannabis_caviar          oracle  fruity_pebbles      loud_dream      white_fire
              6               0               0               0               1
             j1      hammerhead           sista          goblin       fishermen
              0               0               0               0               0
          cloud        paradise
              0               0
```

```
solution(solved_model, "objval")
```

```
[1] 5300
```

Ali wondered, "What would happen if I changed that constraint to only be a 0 or a 1?"

```
model <- OP(objective = special_items$value,
            constraints = constraints,
            types = rep.int("B", 12),
            maximum = TRUE)

solved_model <- ROI_solve(model)

solution(solved_model) |>
  setNames(special_items$item)
```

```
cannabis_caviar          oracle  fruity_pebbles      loud_dream      white_fire
              0               1               1               0               1
             j1      hammerhead           sista          goblin       fishermen
              0               0               1               0               0
          cloud        paradise
              0               1
```

```
solution(solved_model, "objval")
```

```
[1] 4550
```

What should Ali do?

Chapter 6

Simulation

Ali was hearing a lot of people talk about "what if" scenarios and frankly, the requests for new work was coming in too fast. After dealing with marketing and retail problems, they would keep coming back to ask more questions. "What if we would purchase this many items?", Boa and Blaise both asked. Even Castel was asking for information when the greenhouse situations would change. It seemed like Ali really needed to turn all of the code into functions... but a thought started chewing away at Ali brain.

"What if all of those things are just variables from different distributions".

"If I know the past, I can figure out the distribution, and spin up some tables".

The final straw was this email from Chaman, the head of grow operations:

> Good afternoon Ali,
> We have an interesting problem in our grow operations right now. Our plants are very successful, but a new strain isn't really behaving how we would expect, given the expected probability of full maturation. Nearly 99.9% of these plants live and the maturation cycle runs about 65 days. We started with a 1000 plants on our last run, but we only ended up with 750 plants at the end of the growing cycle. We also had some weird pump failures; we'd think it would only happen about 1/100 times. Realistically, how many times would something like this happen? Better said, is 750 the best we can get with 1000 plants? Thanks for the help,
> Chaman

Ali knew that two other analysts might be able to offer some insight: Shashi knew everything statistical and Alex was the simulation wizard.

Starting at the beginning is always important, so Ali swung by Shashi's cubicle.

6.1 Distributions

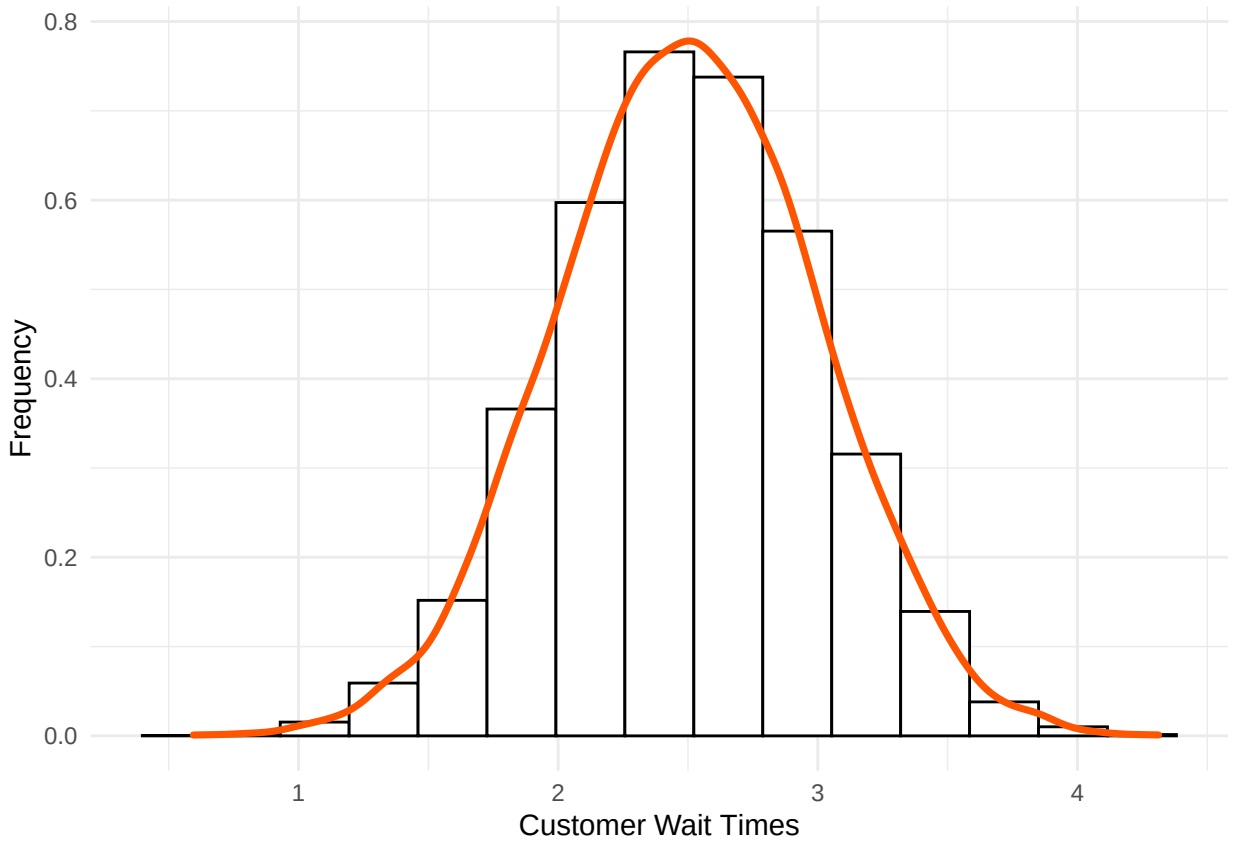
"You were right to come to me first", Shashi said, "I'll fill you in on a few distributions that might be helpful to you."

"Distributions drive every single part of simulation – every event that you can ever imagine comes from some type of distribution.", Shashi said.

6.1.1 Normal Distribution

"For our normal distribution, we know the μ and σ."

"It is likely the most common and you'll find that a lot of processes are normally distributed."



"You might get customer data and want to test if a variable is normally distributed."

"Let's start by creating a normal distribution."

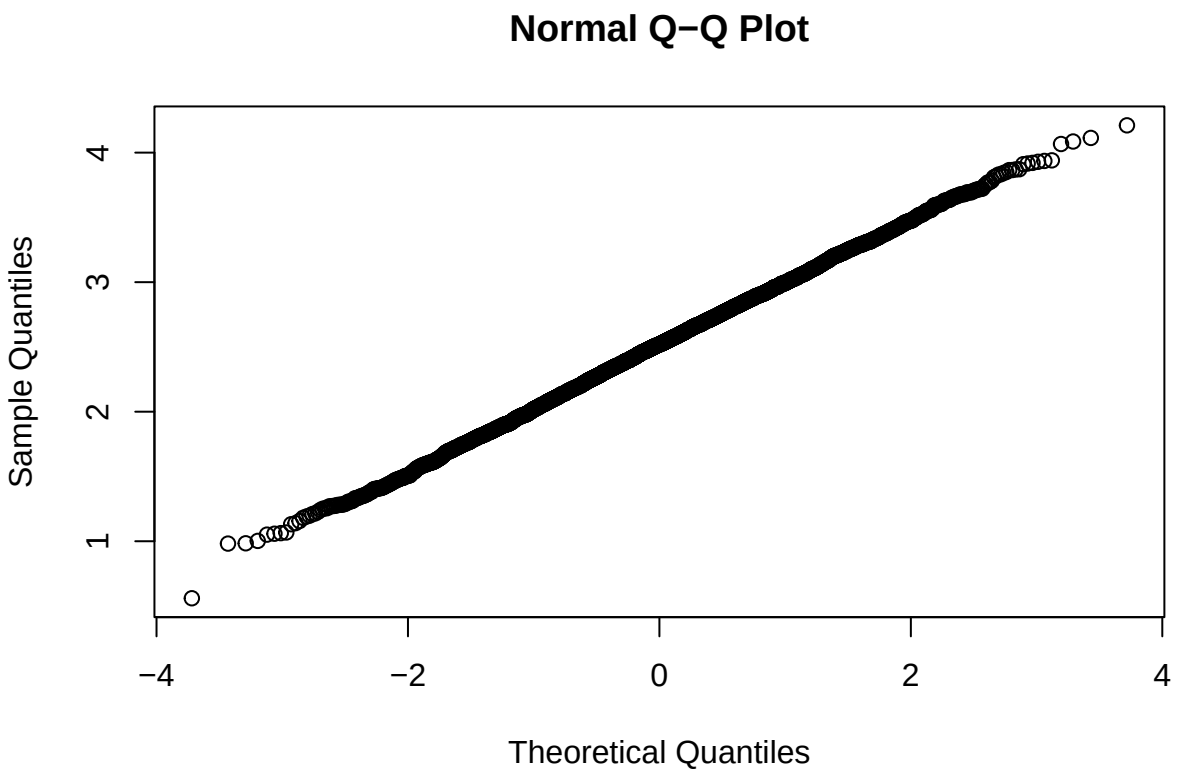
```
normal_variable <- rnorm(n = 5000, mean = 2.5, sd = .5)
```

"And an exponential distribution. . . I'll tell you more about that in a few minutes."

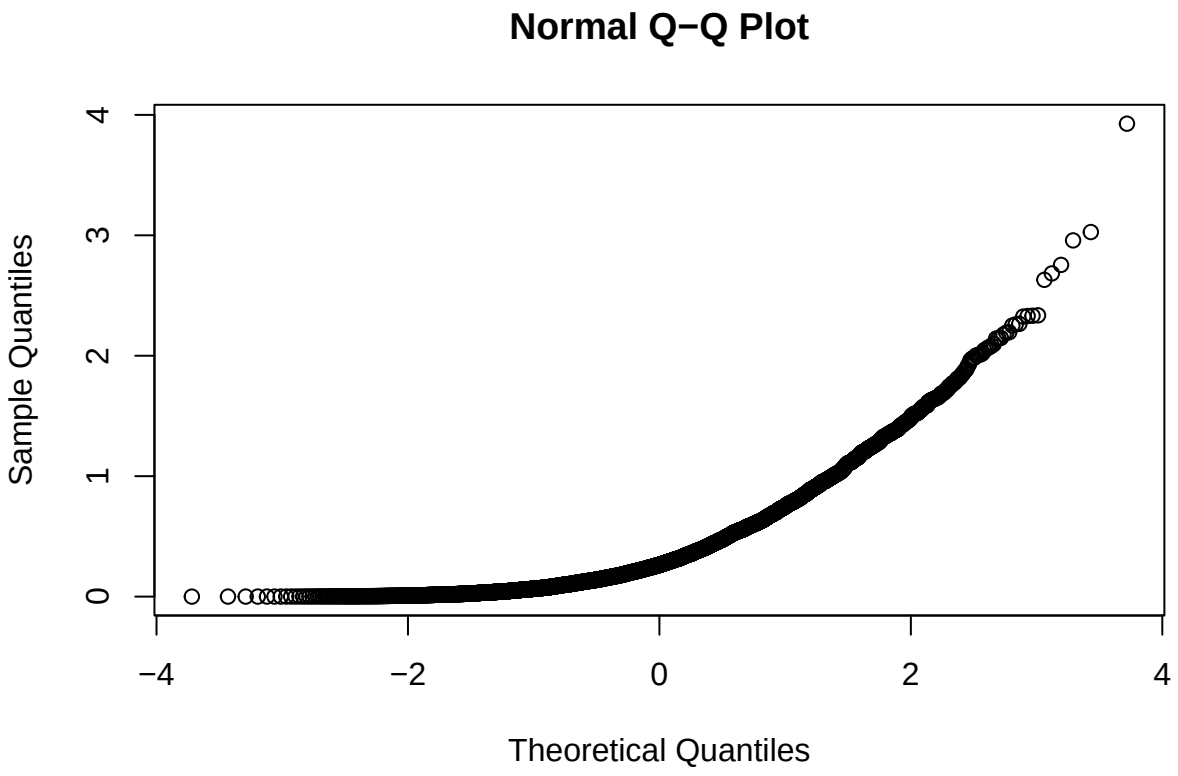
```
exponential_variable <- rexp(n = 5000, rate = 2.5)
```

"We can check both with a qq plot for normality."

```
qqnorm(normal_variable)
```




```
qqnorm(exponential_variable)
```



"The normal distribution will follow the diagonal perfectly, but the exponential looks like it curves down".

"If you ever want to test for the normal distribution, you can just use the Shapiro test."

```
shapiro.test(normal_variable)
```

```
    Shapiro-Wilk normality test

data:  normal_variable
W = 0.99965, p-value = 0.5551
```

```
shapiro.test(exponential_variable)
```

```
    Shapiro-Wilk normality test

data:  exponential_variable
W = 0.82426, p-value < 2.2e-16
```

"Think about it like this", Shashi said, "We are trying to test if our observed distribution is significantly different than a normal distribution."

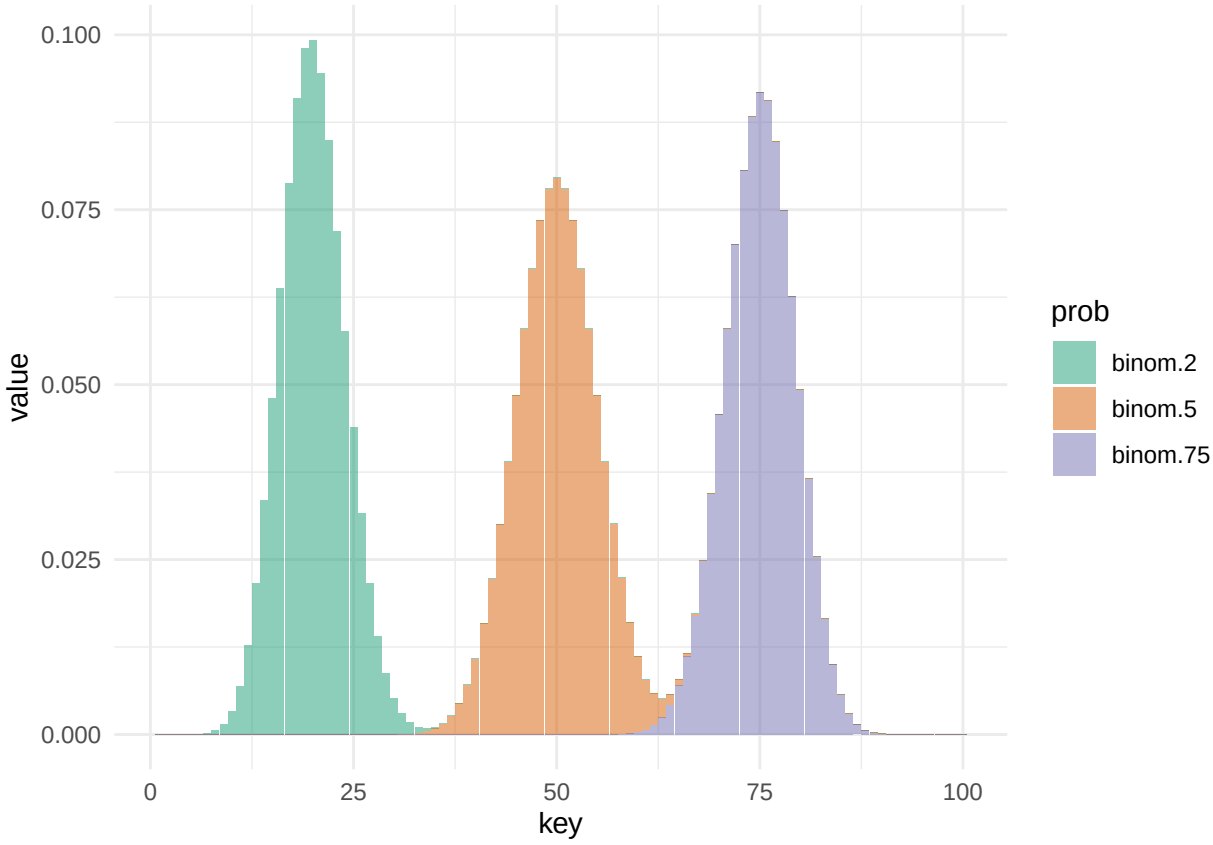
"We would want to see a p-value above .05 to suggest that we might be dealing with a normally-distributed variable."

"Clearly, the exponential distribution is significantly different."

"That is awesome!", Ali said.

6.1.2 Binomial Distribution

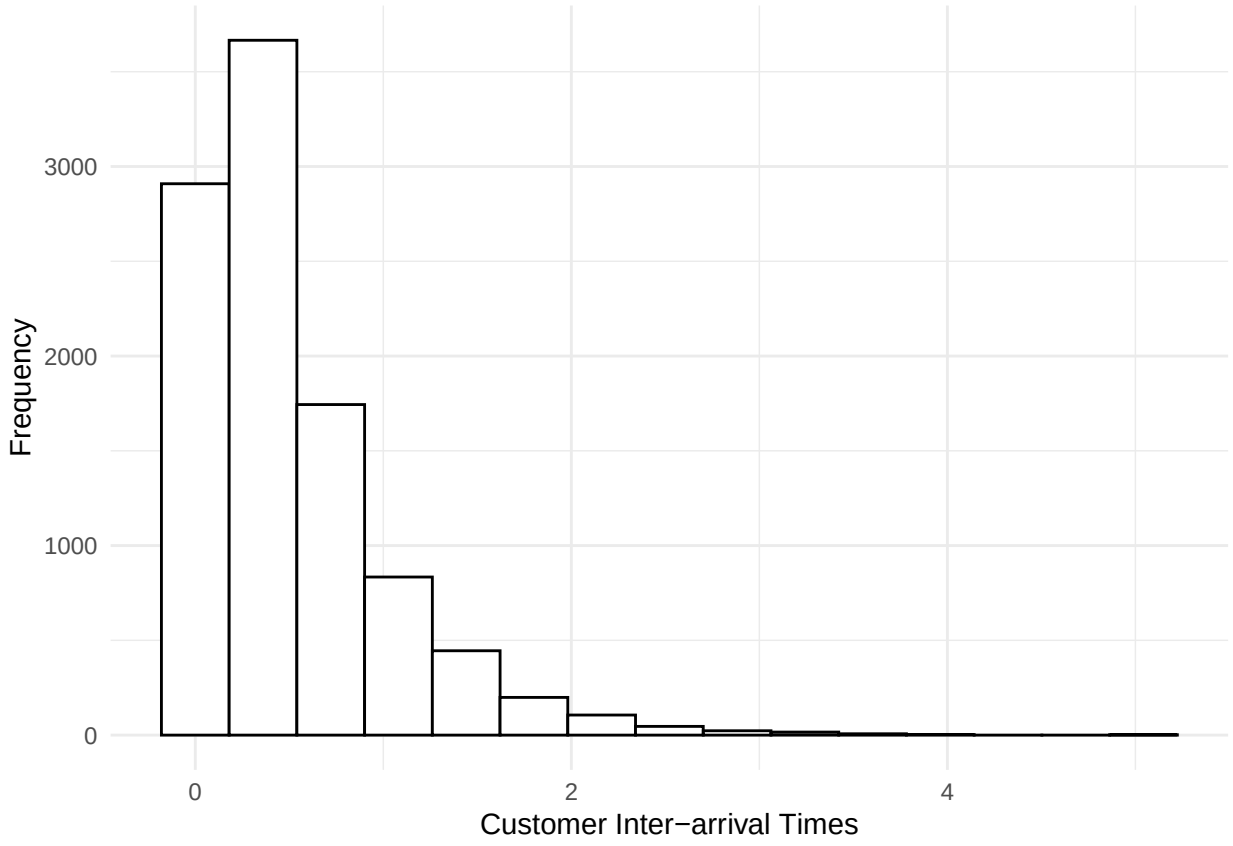
"Anything that has two outcomes – dead/alive, failure/success, missed/made – can be modeled with a binomial distribution."



6.1.3 Exponential Distribution

"We can only know one thing about the exponential distribution: μ (also expressed as a rate)."

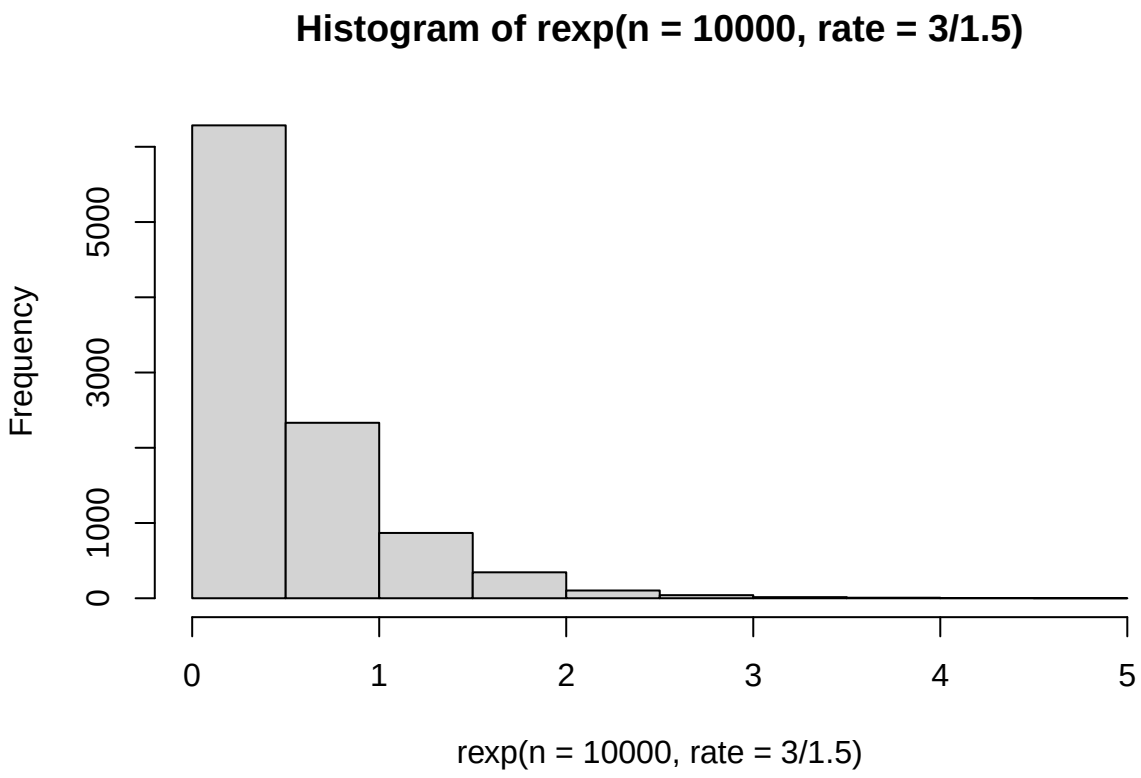
"Just about any arrival process can be approximated by an exponential distribution."



"That rate parameter is a bit confusing. . . the easiest way to express it is as a ratio."

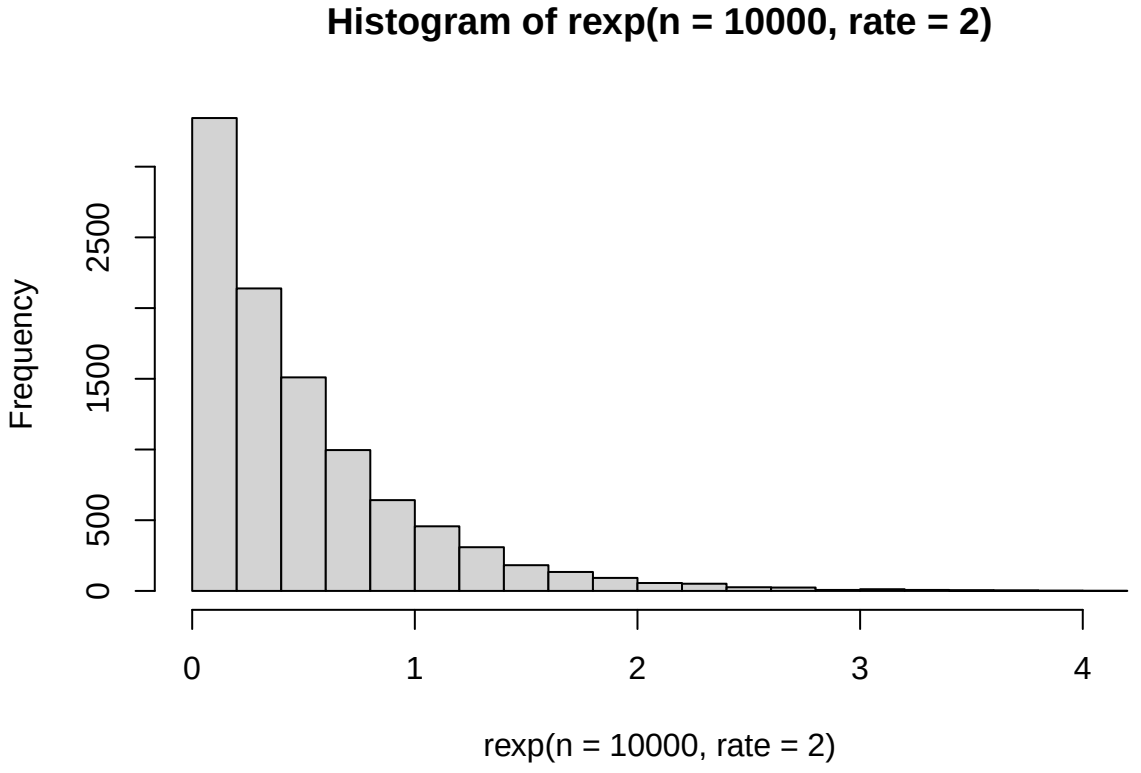
"If 3 people enter the store every minute and a half, than it would be a rate of 3/1.5".

```
hist(rexp(n = 10000, rate = 3 / 1.5))
```



"You could also just put the average rate in."

```
hist(rexp(n = 10000, rate = 2))
```

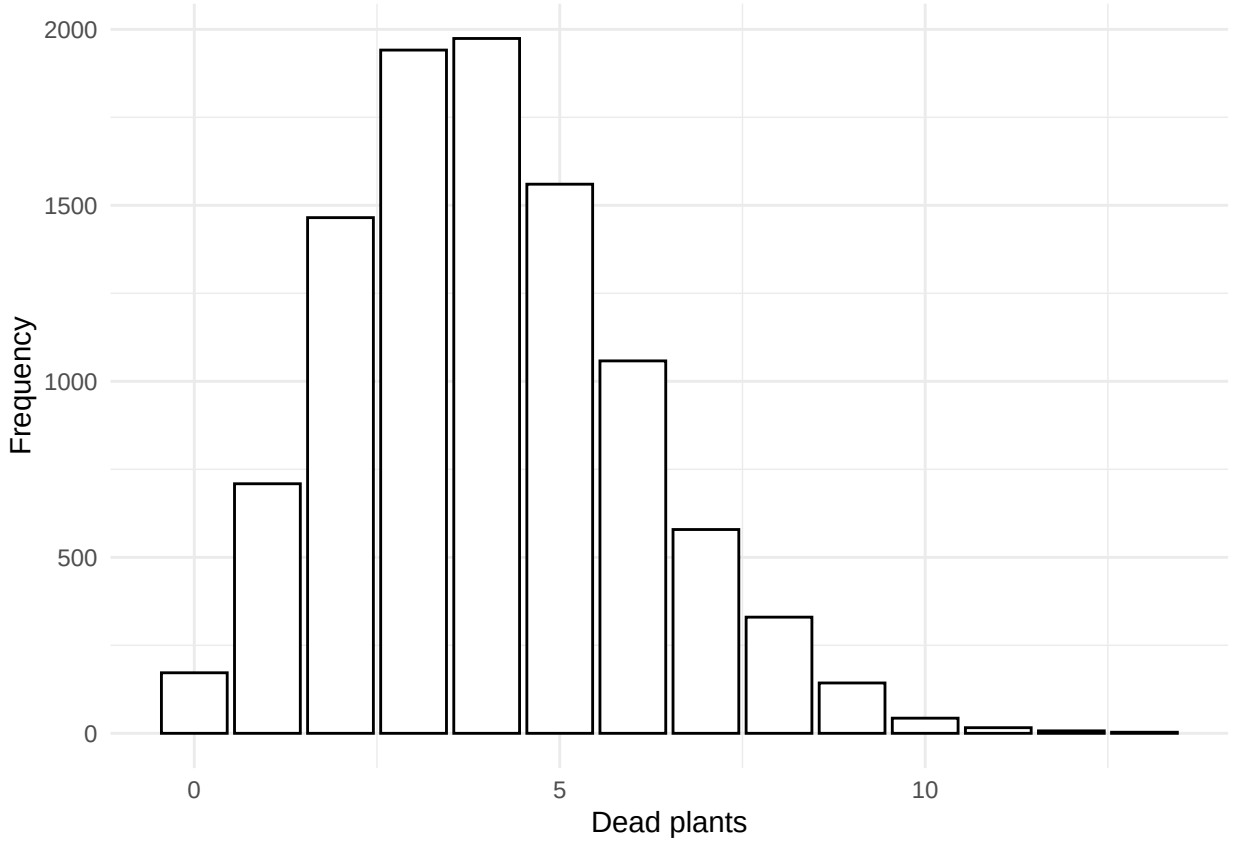


"They are really the same thing."

6.1.4 Poisson Distribution

"The poisson is an interesting distribution – it tends to deal with count-related variables. It tells us the probability of a count occurring. We know its λ."

"The λ is just a fancy way of saying the average number of events, or the incidence rate."



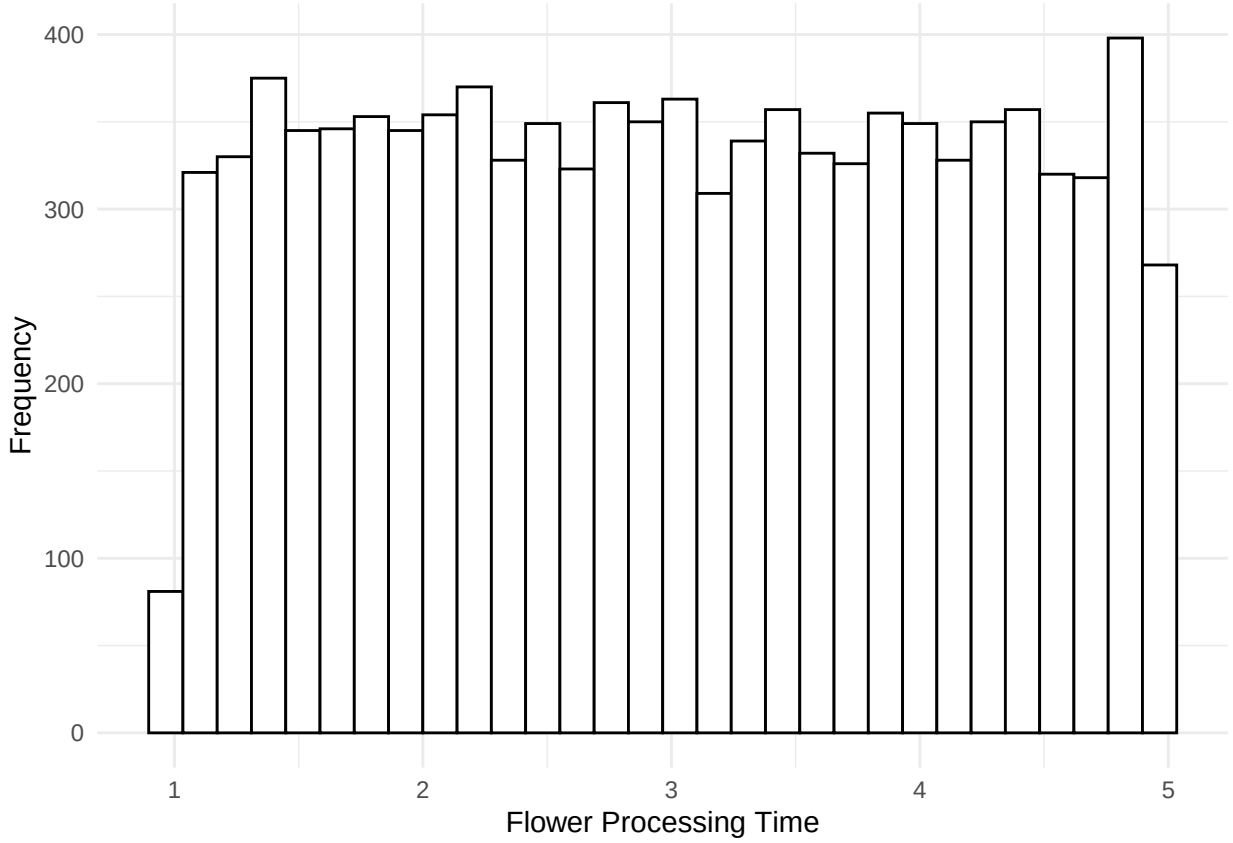
"Interestingly, the Poisson distribution has a relationship with the Exponential distribution: Poisson deals with the number of occurrences and the Exponential deals with the time between occurrences."

"Whoa. . . I've got a lot to learn here.", Ali mumbled.

"All with time!", Alex reassured.

6.1.5 Uniform Distribution

"While people tend to think about the Gaussian distribution as the most vanilla of all distributions, it really is not – I would say that distinction belongs to the uniform distribution. We don't even get any fancy Greek letters, just a minimum and a maximum. Why? Because knowing the min and max will tell us that there is an equal probability of drawing a value anywhere within that range."



"Sound cool?", Shashi asked.

"For sure!", Ali replied, and went off to find Alex.

6.2 An Important Distinction

As soon as Ali started talking to Alex, a tangent started. "First and foremost, I want to set you straight on something: simulation is not what-if analysis."

"What's the different?", Ali asked.

"The what-if analysis isn't really guided by distributions, but more along the lines of low, medium, and high values."

"I think I need an example.", Ali said.

"I thought you'd never ask!", Alex exclaimed.

"Let's look at a really simple process: the number of people who come into a retail store."

"Let's say that every person who buys something in the store, shops for 20 minutes on average, with a standard deviation of 2.5 minutes."

"We might just be interested to know how much total time those people are in the store."

```
what_if_times <- c(low_value = 50,
                   mid_value = 100,
                   high_value = 150)

what_if_sums <- sapply(what_if_times, function(x) {
  sum(rnorm(x, mean = 20, sd = 2.5))
```

```
})

what_if_sums
```

```
 low_value  mid_value high_value
  990.5809  1963.7761  2956.7066
```

```
mean(what_if_sums)
```

```
[1] 1970.355
```

"That seems simple enough.", remarked Ali. "Those answers really seem pretty obvious."

Alex nodded. "They definitely do, but we have a potential problem here."

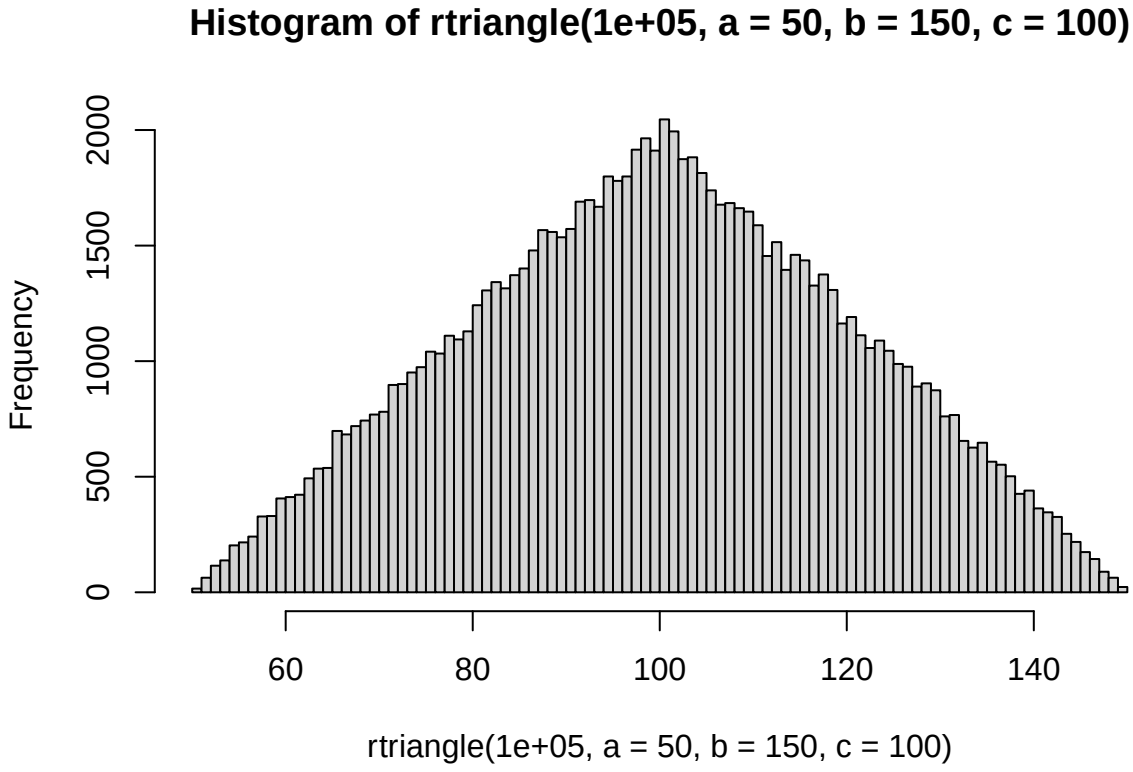
Ali had absolutely zero idea where Alex was going.

"Let me give you some more info; we are giving all of those an equal probability of occurring." Alex asked, "Does that seem reasonable?"

"Probably not.", Ali admitted.

"The most common number of people that come into the score is 100, but the highs and lows rarely happen."

"Check this distribution out:"



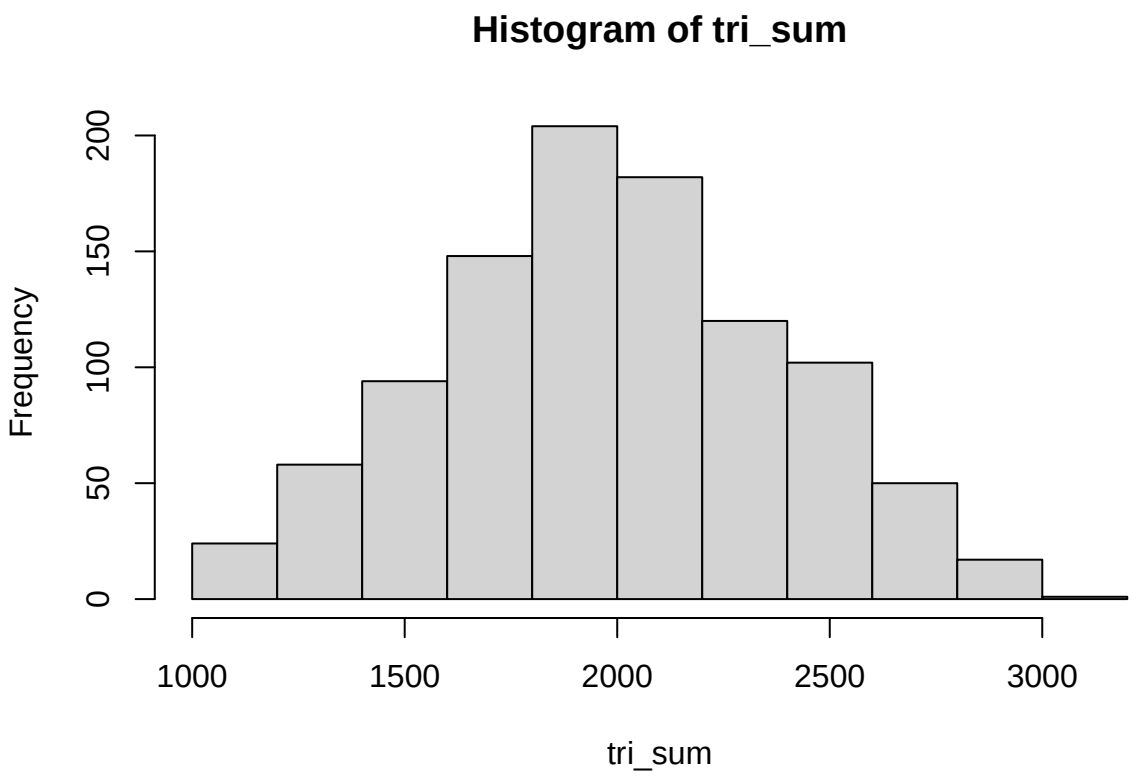
"In the triangular distribution, we have a few parameters: the lower limit, the upper limit, and the mode."

"That looks wild!", Ali smiled.

"Now, let's see how we can incorporate this into a really small simulation."

```
library(triangle)

triangle_draws <- rtriangle(n = 1000, a = 50, b = 150, c = 100)
```

```
tri_sum <- sapply(triangle_draws, function(x) {
  sum(rnorm(x, mean = 20, sd = 2.5))
})

hist(tri_sum)
```



"Is the average wildly different?", Alex asked.

"Nope!", Ali said, "but that is a much better representation of what we would expect!"

"Using distributions is always the way to go.", Alex said.

"Shashi said something like that.", Ali replied.

"Things become even more interesting as we start to incorporate more distributions into our problems.", Alex smiled as he began typing.

"I'm sure Shashi showed you the exponential distribution", Alex continued, "so let me show you what we can do with it."

"Let's say that our stores get raw product shipments at a rate of 1 every 3 days, on average."

```
interarrival_time <- rexp(100, rate = 1/3)
```

"Now, let's say that the average shipment follows a normal distribution, with a mean of 2 pounds and a standard deviation of .5 pounds."

```
shipment_pounds <- rnorm(100, mean = 2, sd = .5)
```

"I think I'm following you.", Ali nodded along.

Alex continued, "Let's just program something small."

"I'd be curious to know how many days it would take to get 100 shipments and how much total weight we would get."

"We will start by creating day 1 in the simulation:"

```
day <- 1
```

"And our starting pounds."

```
total_pounds_delivered <- 0
```

"Next we can specify how many shipments we would like to test this over."

```
n_shipments <- 100
```

"Finally, we need to use a `for` loop to iterate over those 100 shipments."

```
for(i in 1:n_shipments) {
  # For every shipment, we want to generate 1 draw from
  # the normal distribution.
  shipment_pounds <- rnorm(1, mean = 2, sd = .5)

  # We also want to generate the rate at which
  # the shipments will arrive.
  interarrival_time <- rexp(1, rate = 1/3)

  # Now we can add the shipped pounds to our total_pounds_delivered.
  # This will update total_pounds_delivered at every iteration.
  total_pounds_delivered <- total_pounds_delivered + shipment_pounds

  # And the same idea is used for keeping track of the days.
  day <- day + interarrival_time
}
```

"Let's see those results!"

```
day
```

```
[1] 306.9726
```

```
total_pounds_delivered
```

```
[1] 187.0088
```

"So it would take us about 307 days to get 100 shipments and we would get about 187.", Alex noted.

"That is absolutely awesome!", Ali gasped. "It is like we are programming the real world."

"That's right.", Alex nodded. "You are only limited by what you can program."

"I've got one more thing to show you... how to repeat the process."

"All we need to do is to take our complete code:"

```
 day <- 1

  total_pounds_delivered <- 0

  n_shipments <- 100

  for(i in 1:n_shipments) {
```

```
    shipment_pounds <- rnorm(1, mean = 2, sd = .5)

    interarrival_time <- rexp(1, rate = 1/3)

    total_pounds_delivered <- total_pounds_delivered + shipment_pounds

    day <- day + interarrival_time
  }
```

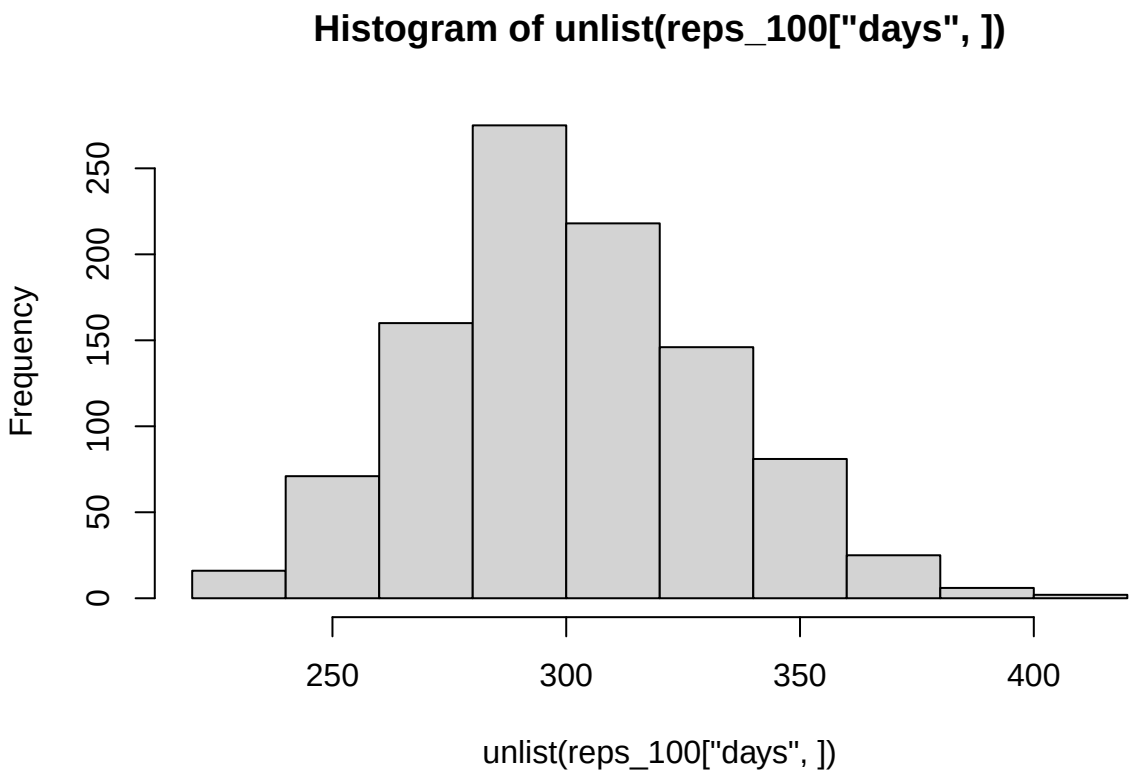
"And then throw that into the replicate function!"

```
reps_100 <- replicate(n = 1000, expr = {
  day <- 1

  total_pounds_delivered <- 0

  n_shipments <- 100

  for(i in 1:n_shipments) {

    shipment_pounds <- rnorm(1, mean = 2, sd = .5)

    interarrival_time <- rexp(1, rate = 1/3)

    total_pounds_delivered <- total_pounds_delivered + shipment_pounds

    day <- day + interarrival_time
  }

  # This is the only different part; I am just taking my results
  # and throwing them into a data frame. That way, I am always
  # returning a data frame!
  results <- data.frame(days = day,
                        total_pounds = total_pounds_delivered)

  results
}, simplify = TRUE)
```

"After that runs, `reps_100` will be a weird-looking object, but we can still use it for some histograms."

"We need to refer to the row names of `days` and `total_pounds` to extract the rows out and then `unlist` those rows into a vector:

```
hist(unlist(reps_100["days",]))
```




```
hist(unlist(reps_100["total_pounds",]))
```

Ali was in awe: “That is coolest thing I’ve seen.”

6.3 ClassOverflow

Let's spend some time together working through the email from earlier.