Project 1 Report

Clayton Engle

CSCI 3327 Probability and Applied Statistics

Stockton University

2 March 2023

Contents

I Introduction **3**
0.1 Overview 4
0.2 Git Workflow Explained 7

II Formulas **9**
0.3 Arithmetic Operations 10
0.4 Conditional Probability 12
0.5 Binomial Distribution 14
0.6 Geometric Distribution 15

III The Birthday Paradox **16**
0.7 Background 17
0.8 Mathematical Solve 17
0.9 Java Simulation 18

IV Monty Hall Problem **20**
0.10 Introduction 21
0.11 Mathematical Solve 21
0.12 Monte Carlo Simulation 22

V Sources **24**
0.13 Data Structures and Algorithms 25
0.14 Design Patterns 25

0.15 Math Formulas and Proofs 26

Part I

Introduction

Probability and statistics are fundamental concepts in data analysis across a wide range of fields, including engineering, finance, healthcare, and social sciences. Solving basic probability and statistics problems can prove to be arduous and counter intuitive, making it an onerous and error-prone task. Thus, having adequate software is an incredibly powerful tool for solving these types of problems in the real world. In this report, I will present an application that I have developed to solve common probability and statistics problems. The application is designed to be user-friendly and efficient, In this report, I will provide an in-depth explanation of the software's key features, capabilities, and its limitations. Overall, this report will demonstrate the usefulness and applicability of the software for anyone working with basic probability and statistics problems.

0.1 Overview

My program is organized into four sections which work together to provide a the user with am easy to use, snappy-feeling stats calculator. The first of these sections is the view, which contains a GUI for selecting functions and entering parameters. When the user presses the "ENTER" button, the information entered by the user is packaged up and sent to the controller section of the program. The controller functions as the nucleus of the program, as it manages where information is sent and decides which processes should be run and when. When the controller receives input from the view, it creates a task based on what needs to be done and passes the task off to the scheduler. Tasks usually involve running input through one or more functions, which are located in the model section of the program. The model has classes which hold the logic for simulations and stats operations. The final section of the program includes data structures and custom data types used throughout the program. In the following paragraphs, each of these sections will be explained in depth.

The view directory of the program has only 2 classes, both of which live directly in the "view" directory. Ubkuje the other three sections, which are mostly written from-scratch, the view section leans heavily of Java APIs, using the built-in Swing classes to create a simple user interface. I wanted to spend most of my effort on making efficient methods for stats operations and growing my knowledge on programming, so I made the GUI for the sake of completeness. For this reason, the view section of code runs completely

separately from the other parts of the program. Any user interface could be quickly wired to work with the rest of the program, as the view section sends Strings out, and takes Strings in, and that is it.

The model directory of the code is where all of the logic for performing the calculations exists. In the model, the Stats Library, Monty Hall, and Birthday Paradox classes perform the heavy lifting of the program. In the Stats Library class, every method is designed to work with doubles, or arrays of doubles. In class we learned that auto-boxing can be slow, so I made sure only to use wrapper classes when absolutely necessary. As such, the data structures I wrote for this class are all designed to work with the primitive double type. All functions here are pretty straightforward, and both the simulations are covered in detail later in this report.

The control directory of the program ties the user interface to the model section of the code. Here, input from the user interface is received as a String, and all information returned to the view is a String. However, the Stats Library class needs doubles, so a converter class handles this transition for each input and output. In these classes I needed to use the Java library,I did not have enough time to debug my hand written methods, which worked almost perfectly. Once the input strings have been converted into doubles, a new task is created. Tasks, in the case of my program, are a family of classes which are all "rumble", and contain instructions for completing a problem. To create these new objects on the fly, I implemented an abstract factory pattern. The concrete factory aspect of the design may seem unwarranted, since each factory merely instantiated a new task of a specified type, without adding any extra functionality to the object. My goal in choosing the abstract factory pattern was to have the factories alter certain aspects of the tasks, since the resources required for each task are very different. Additionally, while looking at the proofs for the equations in the Stats Library in the textbook, I had a few ideas on how certain operations could be multi-threaded to make processing large data sets faster, by using all the processors available to the JVM. Ideally, the factory could make a rough estimate of what its task needs, quickly check the state of the system and see what resources are available, and dynamically adjust how the task will be executed to optimize the workload for the current situation. For example, if a large amount of the memory is being used, an operation can reduce the size of the hash table it will use. Keeping track of the health and the load on the OS could help make small changes, and a lot of small changes add up to better performing code. However, I did not have the time to get this dialed in, so tasks are created

by the Task Factory and returned to the program control for scheduling.

The final component of the control is the thread scheduler, which is also hilariously beefy for what it actually does. In short, the scheduler receives tasks, and if a processor is available, runs the task until completion. At a deeper level, however, there is incredible potential in this beast for improving program performance. Now just by having a Swing GUI there were already 3 threads I needed to run frequently. The main thread, which is the program control needs to get some processing time as well, since it runs the whole application, and the scheduler itself. Then there will also be large, computationally intensive tasks where the actual data processing will be happening, hence the purpose of the application. To order and manage the list of tasks my program needs to execute, I began looking into different types of binary search trees, since they can add and remove ordered nodes in O log(n) time complexity, so even if my program expands in the future this scheduler will be serviceable. After watching some MIT I decided to use a Red-Black Tree, which is a self balancing binary search tree. The balancing constraints on a red black tree are looser than those on an AVL tree, so red black trees use less memory, but have a slightly worse performance due to higher constant factors. Red black trees also use colors and seem considerably easier to implement than an AVL tree. This data structure would allow urgent tasks to be added and reach the worker threads faster, by passing larger, less important operations. A static, or class-wide method in the Red-Black Node class could then be used to increment down the urgency of low priority functions to prevent starvation. Unlike most of the data structures used for this project I implemented generics in the red black tree because the algorithm was incredibly annoying to write, and now if I ever need another red black tree I have one on deck.

The tasks are executed on worker threads, which are threads created at the scheduler‘s inception, which are waiting on a synchronized queue to have a task for them. There is a thread-safe queue which has a maximum size of 2 elements at the foot of the red black tree. This queue will pull the node with the lowest key value (highest priority) from the tree and notify the worker threads it has a task. If a worker thread is free, it will take the task and execute it. The queue is small so that the tree can still sort elements by priority, but the queue only allows a single thread to access it at a time to prevent a race case where two treads are using the same resources at the same time. Ultimately the scheduler only is used for tasks created by the control. I had wanted to schedule very thread the program has through the

scheduler, and tune it to squeeze out extra performance gains, but for my own sanity I passed the GUI and main thread off to the JVM so when things break I have one less thing to check.

Finally, the main directory contains the main method, and a collection of all the data structures and data types that are used throughout the program. This directory is not quite as exciting as the others, however some of the data structures I wrote are pretty cool. As mentioned earlier, some of the implementations in this program are quite ridiculous in their current state. To turn this in I had to cut out

0.2 Git Workflow Explained

Git is a powerful version control system that offers a variety of tools to manage software development workflow. One of its key features is its ability to create branches, or separate copies of the code, where developers can work on different features or bug fixes without affecting the main codebase. This allows for easy collaboration between team members and reduces the risk of conflicts.

Git also has powerful merging capabilities that make it easy to bring changes from one branch into another. This can be particularly useful when working on complex projects where different features or components need to be integrated together.

Another important tool in Git is its commit system. Git allows developers to create detailed commit messages that describe the changes they've made to the code. This makes it easy to track changes over time and understand what changes were made and why. Additionally, Git's commit system makes it possible to revert changes or undo mistakes, which can be a lifesaver in case of errors or bugs.

Git also offers tools for code review, which can help teams to catch errors or bugs before they are merged into the main codebase. For example, pull requests can be created on web-based Git platforms like GitHub or GitLab, where developers can review changes and provide feedback. This allows for a collaborative approach to development that can lead to higher quality code.

Finally, Git has a number of features that make it well-suited for distributed development teams. For example, Git's ability to synchronize changes between different copies of the code makes it easy for developers to work on their own local copies of the code and then merge their changes with other

team members. This can be particularly useful when working with teams that are spread out across different locations or time zones.

Overall, Git's powerful tools for managing workflow make it an essential tool for modern software development. Its ability to handle complex projects and distributed teams, as well as its support for collaboration and code review, make it a popular choice for developers around the world.

Part II

Formulas

In this section, all the formulas we used in class are typed up. In some cases I typed out the proof of the formulas to use as a reference in later sections when discussing how I implemented the solver for each formula in my program.

0.3 Arithmetic Operations

Mean:

$$\overline{mean(x)} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

In this formula, $\overline{x}$ represents the arithmetic mean, n represents the number of values in the set, and x_i represents the i-th value in the set. The formula represents the sum of all values in the set divided by the number of values in the set, which gives the average value of the set.

Median:

$$\tilde{x} = x_{\frac{n+1}{2}} \; if \; n \; is \; odd, \; \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}) \; if \; n \; is \; even \tag{2}$$

In this formula, $\tilde{x}$ represents the arithmetic median, n represents the number of values in the set, and $x_{\frac{n+1}{2}}$ and $x_{\frac{n}{2}}$ represent the middle values in the set when n is odd or even, respectively.

Mode:

$$mode(x) = x_i argmax \sum_{i=1}^{n} \delta(x - x_i) \tag{3}$$

In this formula, $mode(x)$ represents the arithmetic mode, n represents the number of values in the set, x_i represents the i-th value in the set, and $\delta(x - x_i)$ is the Kronecker delta function, which equals 1 if $x = x_i$ and 0 otherwise. The mode is the value that appears most frequently in the set.

Variance:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 \tag{4}$$

In this formula, σ^2 represents the arithmetic variance, n represents the number of values in the set, x_i represents the i-th value in the set, and $\overline{x}$ represents

the arithmetic mean of the set. The variance measures the spread of the set around the mean.

Similar to the mean, the variance calculation can be manipulated so that multiple processes can be executed at once. Understanding the bones of the variance formula will be essential for explaining the details of how we can do this, which will be discussed in a later section of the report. So, starting with the variance formula shown above, here is the proof:

Expanding the square, we get:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2x_i\overline{x} + \overline{x}^2) \tag{5}$$

Using the linearity of summation, we can split this sum into three separate sums:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}x_i^2 - \frac{2}{n}\sum_{i=1}^{n}x_i\overline{x} + \frac{1}{n}\sum_{i=1}^{n}\overline{x}^2 \tag{6}$$

The first sum can be rewritten in terms of the square of the individual deviations from the mean:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 + \frac{1}{n}\sum_{i=1}^{n}\overline{x}^2 - \frac{2}{n}\sum_{i=1}^{n}x_i\overline{x} \tag{7}$$

Simplifying the second and third terms, we get:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 + \overline{x}^2 - 2\overline{x}^2 \tag{8}$$

Simplifying further, we get:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 - \overline{x}^2 \tag{9}$$

Finally, using the definition of the arithmetic mean, $\overline{x} = \frac{1}{n}\sum_{i=1}^{n}x_i$, we can substitute and simplify further:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \frac{1}{n}\sum_{j=1}^{n}x_j)^2 \tag{10}$$

This completes the proof of the arithmetic variance formula.

Standard Deviation:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{11}$$

In this formula, σ represents the arithmetic standard deviation, n represents the number of values in the set, x_i represents the i-th value in the set, and $\overline{x}$ represents the arithmetic mean of the set. The standard deviation is the square root of the variance and also measures the spread of the set around the mean, but in the same units as the original data.

0.4 Conditional Probability

Combinations:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Permutations:

$$P_n^k = \frac{n!}{(n-k)!}$$

In both formulas, n represents the total number of items, and k represents the number of items to be selected. The notation $\binom{n}{k}$ represents the number of combinations of n items taken k at a time, and P_n^k represents the number of permutations of n items taken k at a time.

Probability of event A given event B:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

In this formula, $P(A \mid B)$ represents the probability of event A given that event B has occurred. $P(A \cap B)$ represents the probability that both events A and B occur, and $P(B)$ represents the probability that event B occurs.

Bayes' Theorem:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

In this formula, $P(A \mid B)$ represents the probability of event A given event B, $P(B \mid A)$ represents the probability of event B given event A, $P(A)$ represents the prior probability of event A, and $P(B)$ represents the prior probability of event B. The denominator $P(B)$ is known as the marginal probability of event B, which can be calculated using the law of total probability:

$$P(B) = \sum_{i} P(B \mid A_i) \cdot P(A_i)$$

Where A_i represents all possible events that are mutually exclusive and exhaustive. Bayes' Theorem can be generalized for any number of events. Let's assume that we have n mutually exclusive and exhaustive events $A_1, A_2, \ldots, A_n$, and an event B that we are interested in. Then, Bayes' theorem can be generalized to:

$$P(A_i \mid B) = \frac{P(B \mid A_i) \cdot P(A_i)}{\sum_{j=1}^{n} P(B \mid A_j) \cdot P(A_j)}$$

for $i = 1, 2, \ldots, n$.
Starting from the definition of conditional probability:

$$P(A_i \mid B) = \frac{P(A_i \cap B)}{P(B)}$$

Using the law of total probability, we can express the joint probability $P(A_i \cap B)$ as:

$$P(A_i \cap B) = P(B \mid A_i) \cdot P(A_i)$$

Substituting this expression into the numerator of the previous equation, we get:

$$P(A_i \mid B) = \frac{P(B \mid A_i) \cdot P(A_i)}{P(B)}$$

Using the law of total probability again, we can express the marginal probability $P(B)$ as:

$$P(B) = \sum_{j=1}^{n} P(B \mid A_j) \cdot P(A_j)$$

Substituting this expression into the denominator of the previous equation, we get:

$$P(A_i \mid B) = \frac{P(B \mid A_i) \cdot P(A_i)}{\sum_{j=1}^{n} P(B \mid A_j) \cdot P(A_j)}$$

This is the generalized form of Bayes' theorem for n events. It allows us to update our beliefs about the probability of each event A_i given new evidence in the form of event B.

0.5 Binomial Distribution

Probability Mass Function:

$$P(X = k) = nkp^k(1-p)^{n-k} \tag{12}$$

where:

- nk is the binomial coefficient, representing the number of ways to choose k items out of n.
- p^k is p raised to the power of k, indicating the probability of k successes in n trials.
- $(1-p)^{n-k}$ is $(1-p)$ raised to the power of $(n-k)$, indicating the probability of $(n-k)$ failures in n trials.

Expected value of a binomial distribution:

$$E(X) = np \tag{13}$$

The variance of a binomial distribution is given by:

$$Var(X) = np(1-p) \tag{14}$$

Probability of a success at or before n trials:

$$P(X \leq n) = \sum_{k=0}^{n} nkp^k(1-p)^{n-k} \tag{15}$$

Probability of a success after n trials:

$$P(X > n) = \sum_{k=n+1}^{\infty} nkp^k(1-p)^{n-k} \tag{16}$$

0.6 Geometric Distribution

Probability mass function:

$$P(X = k) = (1-p)^{k-1}p$$

where: X is a random variable representing the number of trials until the first success p is the probability of success on each trial (and $q = 1 - p$ is the probability of failure) k is the number of trials (where $k = 1, 2, 3, \ldots$) The mean of a geometric distribution is:

$$E(X) = \frac{1}{p}$$

The variance of a geometric distribution is:

$$Var(X) = \frac{1-p}{p^2}$$

And the standard deviation of a geometric distribution is:

$$\sigma = \sqrt{Var(X)} = \sqrt{\frac{1-p}{p^2}}$$

Part III

The Birthday Paradox

0.7 Background

0.8 Mathematical Solve

Let there be n people in a room. We want to calculate the probability that at least two of them share a birthday.

The total number of possible birthdays is 365 (assuming no leap years). The probability that the first person has a unique birthday is $\frac{365}{365}$. The probability that the second person has a unique birthday is $\frac{364}{365}$, since they cannot have the same birthday as the first person. Similarly, the probability that the third person has a unique birthday is $\frac{363}{365}$, and so on.

Therefore, the probability that all n people have unique birthdays is:

$$\frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365-(n-1)}{365} = \frac{365!}{365^n(365-n)!}$$

Using the formula for combinations, we can simplify this expression:

$$\frac{365!}{365^n(365-n)!} = \frac{n!}{365^n(n-365)!} \times \frac{365!}{(365-n)!} = \frac{n!}{365^n(n-365)!} \times 365 \times 364 \times \cdots \times (366-n)$$

The probability that at least two people share a birthday is the complement of the probability that all n people have unique birthdays:

$$1 - \frac{n!}{365^n(n-365)!} \times 365 \times 364 \times \cdots \times (366-n)$$

Simplifying this expression, we get:

$$1 - \frac{365!}{365^n(365-n)!} \times \frac{(365-n)!}{365!} = 1 - \frac{n!}{365^n} \times \frac{(365-n)!}{(365-n)!} = 1 - \frac{n!}{365^n}$$

Therefore, the probability that at least two people share a birthday is:

$$1 - \frac{n!}{365^n}$$

This result shows that the probability of at least two people sharing a birthday increases rapidly as n increases. Using this formula we can find the probability that at least two people share a birthday among a group of 30 people:

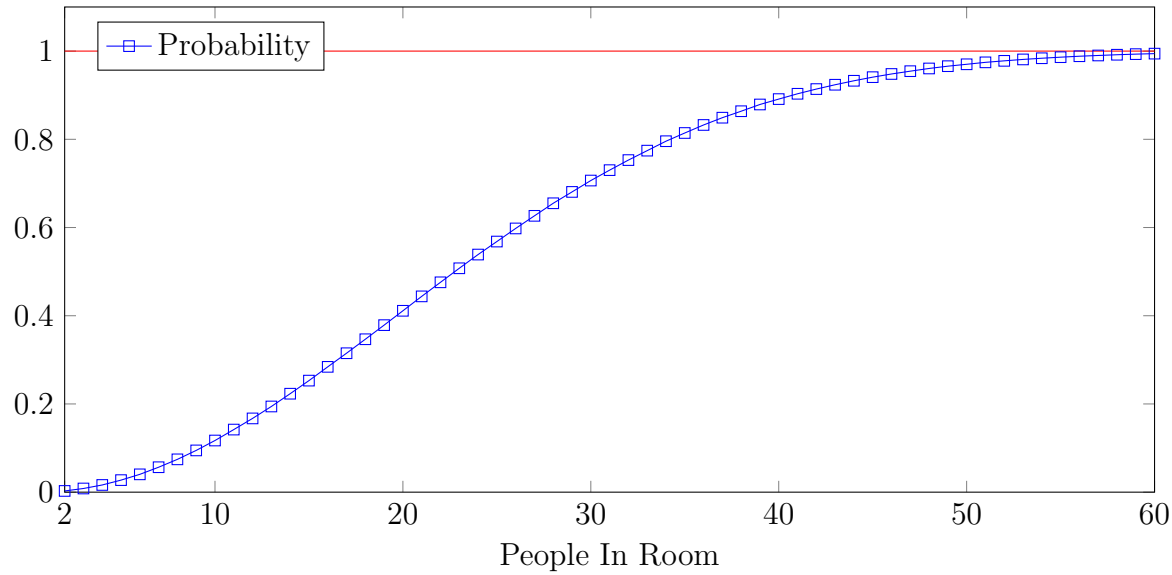
$$1 - \frac{n!}{365^n} = 1 - \frac{30!}{365^{30}} \approx 0.706$$

Therefore, the probability of at least two people sharing a birthday among a group of 30 people is approximately 0.706 or 70.6

0.9 Java Simulation

To help rationalize the probability of the birthday paradox, I developed a Java model to simulate the problem. My program takes as input the number of people in the problem, and the number of times to run the simulation. The graph below represents results from running the simulation two million times on a range of group sizes from 2 through 60. The x-axis here represents the number of people in a the simulated group, while the y-axis represents the number of times at least 2 members of the group had the same birthday divided by the total number of simulations run.

The table below shows the number of people in a group on the x axis, while the probability that at least 2 people in the group share a birthday. The blue line represents the results of 10 million trials run on each group size, from 2 to 60. The red line marks where y = 1, or the maximum probability of any event occurring. It is worth noting the chart is poorly labeled, as I tried to label it for hours, with no success.


Probability
1
0.8
0.6
0.4
0.2
0
2
10
20
30
40
50
60
People In Room

Part IV

Monty Hall Problem

0.10 Introduction

The Monty Hall problem is a classic probability puzzle that is based on a game show scenario. The problem is named after the host of the original game show, Monty Hall. In the game show, there are three doors, behind one of which is a valuable prize, such as a car. The other two doors conceal less valuable prizes, such as goats. The contestant is asked to select one of the doors as their choice for a chance to win the prize. After the contestant has made their initial selection, the host, Monty Hall, opens one of the other two doors to reveal a goat. He then offers the contestant a chance to switch their choice to the remaining unopened door, or to stick with their original choice.

The question is, should the contestant switch their choice or stick with their original choice to have the best chance of winning the prize? In the following sections I will show a mathematical breakdown of the solve, and then explain how a Java program I wrote uses a Monte Carlo simulation to find the solution.

0.11 Mathematical Solve

Suppose you have three doors, numbered 1, 2, and 3, and you choose door 1. Then, the probability of the prize being behind door 1 is $\frac{1}{3}$, the probability of the prize being behind door 2 is $\frac{1}{3}$, and the probability of the prize being behind door 3 is $\frac{1}{3}$.

Let's denote the event of the prize being behind door 1 as event A, the event of the prize being behind door 2 as event B, and the event of the prize being behind door 3 as event C.

After Monty Hall has revealed a goat behind one of the doors, the probability of the prize being behind door 1 given that the goat is not behind door 1 is 0, because Monty Hall will never reveal the prize. The probability of the prize being behind door 2 given that the goat is not behind door 2 is 1, because if the goat is not behind door 2, then the prize must be behind door 2. And the probability of the prize being behind door 3 given that the goat is not behind door 3 is 1, because if the goat is not behind door 3, then the prize must be behind door 3.

So, the probability of winning the prize if you switch doors is the sum of the probabilities of winning the prize given that the goat is not behind door

2 and the goat is not behind door 3:

$$P(A^C|B^C) = P(B|B^C) + P(C|B^C) = 1 + 1 = 2$$

And since the probability of the goat being behind door 2 or 3 is $\frac{2}{3}$, we have:

$$P(B^C) = P(B^C|A)P(A) + P(B^C|B)P(B) + P(B^C|C)P(C) = \frac{2}{3}$$

Therefore, the probability of winning the prize if you switch doors is:

$$P(A^C|B^C) = \frac{P(B|B^C)P(B^C)}{P(B^C)} = \frac{2}{2} = 1$$

And the probability of winning the prize if you stick with your original choice is:

$$P(A|B^C) = \frac{P(A)P(B^C)}{P(B^C)} = \frac{\frac{1}{3} * \frac{2}{3}}{\frac{2}{3}} = \frac{1}{3}$$

So, we have shown that the probability of winning the prize if you switch doors is $\frac{2}{3}$, while the probability of winning the prize if you stick with your original choice is $\frac{1}{3}$. Hence, it is better to switch doors.

This proof uses Bayes' theorem, which states that, for two events A and B and a sample space Ω, the following relationship holds:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $P(A|B)$ is the conditional probability of A given B, $P(B|A)$ is the conditional probability of B given A, and $P(A)$ and $P(B)$ are the marginal probabilities of A and B, respectively.

0.12 Monte Carlo Simulation

The program uses a Monte Carlo simulation to approximate the probabilities of winning the prize by sticking with the initial choice or switching. In each iteration of the simulation, the program randomly selects which door the prize is behind, which door the player initially chooses, and which door Monty

Hall reveals. It then simulates both strategies (sticking with the initial choice and switching) and counts the number of times the player wins with each strategy. Finally, it calculates the probabilities of winning with each strategy by dividing the number of wins by the total number of simulations.

Table 1 below shows the results of the simulation with a range of trials. As the number of simulations increases, the winning percentages become more and more accurate. At 10 million trials, the program won when it switched doors 66.6667% of the time, and recorded a win when switching doors 33.3333% of the time. This shows the simulation is accurate, and requires quite a bit of "brute force" to achieve an accurate result.

Table 1: Program Results

Trials	Win % Switching	Win % Sticking
10,000	0.6732	0.3268
100,000	0.66711	0.33289
1,000,000	0.666234	0.333766
10,000,000	0.666667	0.333333

Part V

Sources

0.13 Data Structures and Algorithms

Aggarwal, C. C. (2016). Hashing: Fundamentals and Applications. Springer International Publishing.

Compiler Explorer. (n.d.). Compiler Explorer. https://godbolt.org/

Cormen, T. (2011, September 8). Red-black trees [Video]. Introduction to Algorithms. Massachusetts Institute of Technology. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Google. (2023). FarmHash source code. In GitHub. https://github.com/google/farmhash/tree/

Tim Roughgarden. (2018, January 9). Hash Tables [Video]. Algorithms Specialization. Stanford University. https://www.coursera.org/lecture/algorithms-divide-conquer/hash-tables-C3qUt

Wikipedia. (2023, February 21). MurmurHash. In Wikipedia. https://en.wikipedia.org/wiki/M

0.14 Design Patterns

Freeman, E., Freeman, E. (2004). Head First Design Patterns. O'Reilly Media, Inc.

University of Waterloo. (n.d.). Software Design Patterns. Retrieved from https://www.youtube.com/playlist?list=PLizsthdRd0Yy6i5oMW2QZ6UvZ6O5twiMd

0.15 Math Formulas and Proofs

Wackerly, D. D., Mendenhall, W., Scheaffer, R. L. (2008). Mathematical statistics with applications (7th ed.). Brooks/Cole, Cengage Learning.