7 Discrete Probability

Introduction

In this chapter, we will see how to use Maple to perform computations in discrete probability and how to use Maple's capabilities to explore concepts of discrete probability. We will continue to make use of the **combinat** package described in the previous chapter. We will also introduce the **Statistics** package. While the **Statistics** package has the support for descriptive statistics and visualization of data that you would expect, it also provides functionality that will help us explore probability distributions and random variables. This includes the ability to create a random variable with a defined distribution and then perform computations with that variable. We load both packages now.

```
> with(combinat):
```

```
> with(Statistics):
```

In this chapter, we will make use of simulations to help explore concepts in discrete probability. In this context, a simulation refers to a computer program, that models a real physical system. For example, instead of flipping an actual coin 100 times and recording whether each flip resulted in "heads" or "tails," we could write a program that uses random numbers to generate a sequence of one hundred "heads" or "tails."

Simulations are useful in discrete probability from two different perspectives. First, they can help analyze and/or confirm probabilities for systems that are difficult to compute deductively. For example, Computations and Explorations 7 from the text asks you to simulate the odd-person-out procedure in order to confirm your deductive calculations. Second, simulations can be very helpful as a way to better understand a problem and how to arrive at a solution. For example, in the Computer Projects 10 exercise from the text, you are asked to build a simulation of the famous Monty Hall Three-Door problem. Building the simulation and analyzing the results can help improve your understanding of the reasons why the strategy described in the text is the best possible.

7.1 An Introduction to Discrete Probability

To find the probability of an event in a finite sample space, you calculate the number of times the event occurs and divide by the total number of possible outcomes (the size of the sample space).

As in Example 4 of Section 7.1, we calculate the probability of winning a lottery, where we need to choose 6 numbers correctly out of 40 possible numbers. The total number of ways to choose 6 numbers is:

```
> numbcomb(40, 6)
```

$$3\,838\,380 \qquad \textbf{(7.1)}$$

Since there is one winning combination, the probability is

```
> 1/(7.1)
```

$$\frac{1}{3\,838\,380} \tag{7.2}$$

We can find a real number approximation by using the **evalf** function—evaluate as a floating point number.

```
> evalf((7.2))
```

$$2.605265763\ 10^{-7} \tag{7.3}$$

We could also force a decimal approximation of the result by using **1.0** or simply **1.**, to show that we wish to work with decimals instead of the exact rational representation. For example, if we needed to choose from 50 numbers, the probability is

```
> 1.0/numbcomb(50,6)
```

$$6.292988981\ 10^{-8} \tag{7.4}$$

Continuing with this type of example, we define a functional operator that computes the probability of winning a lottery where 6 numbers must be matched out of n possible numbers.

```
> Lottery := n -> 1.0/numbcomb(n,6):
```

The probabilities above can then be computed with the function.

```
> Lottery(40), Lottery(50)
```

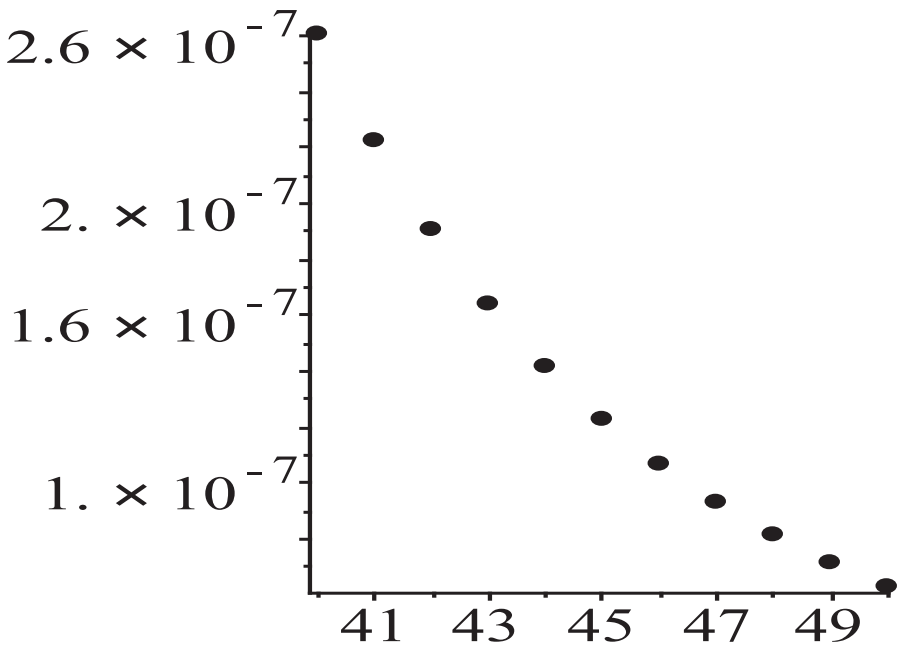
$$2.605265763\ 10^{-7},\ 6.292988981\ 10^{-8} \tag{7.5}$$

We can use the sequence function **seq** to look at a list of probabilities for a range of values of n and graph these values to visualize how the number of possible values in the lottery affects the probability of choosing the correct values.

```
> LotteryVals := [seq(Lottery(n), n = 40..50)]
```

$$\begin{aligned} LotteryVals := [&2.605265763\ 10^{-7}, 2.224007359\ 10^{-7},\\ &1.906292022\ 10^{-7}, 1.640297786\ 10^{-7}, 1.416620815\ 10^{-7},\\ &1.227738040\ 10^{-7}, 1.067598296\ 10^{-7}, 9.313091515\ 10^{-8},\\ &8.148955076\ 10^{-8}, 7.151123842\ 10^{-8}, 6.292988981\ 10^{-8}] \end{aligned} \tag{7.6}$$

```
> plot([$40..50], LotteryVals, style = point, symbol = solidcircle, symbolsize = 15)
```



Recall that the dollar operator with no left operand and a range as the right operand, as in **$40. 0.50**, produces the sequence described by the range. Refer to Section 2.3 of this manual for detailed information on the use of **plot**.

7.2 Probability Theory

We can use Maple to perform a variety of calculations of probabilities.

Random Variables

Consider Example 9 in Section 7.2, which asks us to calculate the probability that eight of the bits in a string of 10 bits are 0s if the probability of a 0 bit is 0.9, the probability of a 1 bit is 0.1, and the bits are generated independently. To perform this calculation, we can input the formula directly:

```
> numbcomb(10, 8) 0.9^8 0.1^2
                    0.1937102445                    (7.7)
```

We can think of this same question in terms of a random variable. Specifically, consider a random variable X that assigns to each string of 10 bits the number of the bits that are 0s. Then, the probability that eight of the bits were 0 is $P(X = 8)$.

To define a random variable in Maple, we use the **RandomVariable** command in the **Statistics** package. The **RandomVariable** command takes one parameter: a probability distribution which serves to specify the probabilities of the possible values of the random variable. The distribution can be one of Maple's built-in distributions or it can be a distribution you define.

Discrete Distributions

Maple provides many different probability distributions, including several discrete distributions. All of the commands described here are inert, that is, they do nothing on their own. Instead, they are used as the argument to the **RandomVariable** command, which is able to interpret them to create a random variable with the desired distribution.

Theorem 2 defines the binomial distribution, implemented in Maple as **Binomial**. The **Binomial** distribution takes two parameters: the number of independent Bernoulli trials and the probability of a success. For the bit string example that began this chapter, there are 10 trials, and we interpret success to be a 0 bit, so the probability of success is 0.9.

```
> RandomVariable(Binomial(10, 0.9))
```

$$\_R \tag{7.8}$$

Note that the output is a symbol consisting of an underscore followed by the letter R. After the first random variable, a number follows the R. This merely indicates that the result is a random variable and the number indicates how many random variables have been defined previously in this Maple session. In the future, we will suppress this output.

Related to the binomial distribution is the probability distribution of a single Bernoulli trial. The **Bernoulli** distribution takes only one parameter, the probability of success. The following creates the random variable associated to a single trial with probability of success 0.9.

```
> RandomVariable(Bernoulli(0.9)) :
```

Definition 1 in Section 7.1 defines the uniform distribution. Maple includes the distribution **DiscreteUniform**, which is different from the **Uniform** distribution (for continuous random variables). The **DiscreteUniform** distribution requires two arguments, the lower and upper bounds of the distribution. For example, the following produces a random variable distributed uniformly on $\{1, 2, 3, 4, 5\}$.

```
> RandomVariable(DiscreteUniform(1, 5)) :
```

Definition 2 in Section 7.4 defines the geometric distribution. The **Geometric** distribution in Maple requires one argument, the probability of success. The following produces the random variable associated to Example 10 from that section.

```
> RandomVariable(Geometric(0.5))
```

Computing Probabilities from a Random Variable

Once a random variable is defined, we use the **Probability** command to calculate probabilities. The probability command's required argument is an event, described as a relation involving a random variable.

Earlier, we described Example 9 in Section 7.2. The random variable associated to that example is defined by the following command.

```
> Ex9 := RandomVariable(Binomial(10, 0.9)):
```

We compute the probability that there are eight 1 bits, that is, $P(X = 8)$, by applying the **Probability** command to the equation that sets the name of the random variable, **Ex9**, equal to 8.

```
> Probability(Ex9 = 8)
```

$$0.1937102445 \qquad \textbf{(7.9)}$$

The probability of at least eight 1 bits is found with an inequality. Recall that in 2-D input mode, the $\geq$ symbol is obtained by pressing the greater than key followed by the equals key.

```
> Probability(Ex9 ≥ 8)
```

$$0.9298091736 \qquad \textbf{(7.10)}$$

The probability of an intersection of events is found by passing a set of relations to **Probability**. For example, the probability that the number of 1s is between 5 and 8 can be thought of as the intersection $\{X \geq 5\} \cap \{X \leq 8\}$. This is computed in Maple as follows:

```
> Probability({Ex9 ≥ 5, Ex9 ≤ 8})
```

$$0.2637541683 \qquad \textbf{(7.11)}$$

To find probabilities of unions of events, you must add the results of the **Probability** command applied to each event separately. $P(\{X \leq 5\} \cup \{8 \leq X\})$.

```
> Probability(Ex9 ≤ 5) + Probability(Ex9 ≥ 8)
```

$$0.9314441110 \qquad \textbf{(7.12)}$$

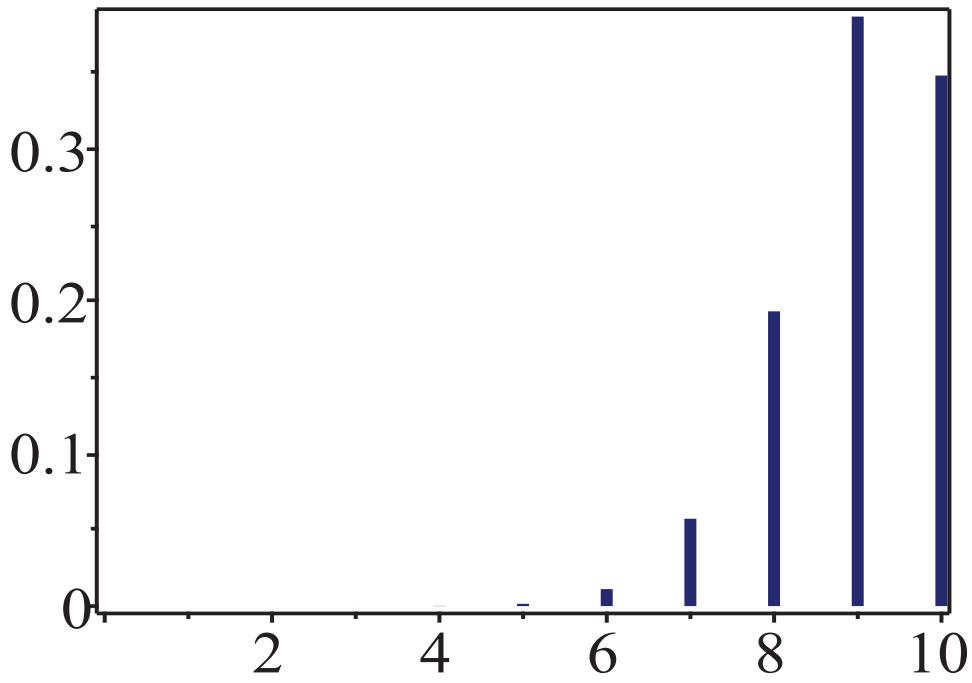
Of course, you must be careful that the events are disjoint in order to add the probabilities.

Graphing Probabilities

It is often useful to graph the probabilities associated to the values of a random variable. To do this, Maple provides the **DenisityPlot** command.

The required argument is the name of a random variable. You can also specify the range of values of the random variable to be displayed, as illustrated below. Omitting the range will cause Maple to determine the size of the plot for itself.

```
> DensityPlot(Ex9, range = 0..10)
```



The **DensityPlot** command also accepts most of the options that are available for the **plot** command.

Defining Random Variables from Your Own Data

Maple provides two convenient ways for you to define probability distributions, and thereby random variables, of your own.

First, to define specific probabilities associated to the integers 1 through n, you can use the **ProbabilityTable** command. The argument to **ProbabilityTable** is a list of real numbers between 0 and 1 that sum to 1. For example, to create a random variable with probability of 1 as $1/2$, probability of 2 equal to $1/8$ and probability of 3 is $3/8$, you enter the following statement.

```
> ProbT := RandomVariable(ProbabilityTable([1/2, 1/8, 3/8])) :
```

The resulting random variable can be used as any other.

```
> Probability(ProbT = 2)
                              1/8                                  (7.13)
```

If, instead of a list of probabilities, you have a list representing the results of experiments, you can use the **EmpiricalDistribution**. This requires one argument, which can be a list or Array.

For example, suppose you manually roll a die 20 times and obtain the following results:

```
> dieRolls := [3, 2, 1, 1, 5, 2, 3, 6, 5, 1, 2, 5, 6, 4, 4, 3, 1, 3, 1, 1]
    dieRolls := [3, 2, 1, 1, 5, 2, 3, 6, 5, 1, 2, 5, 6, 4, 4, 3, 1, 3, 1, 1]   (7.14)
```

Form a random variable by applying **EmpiricalDistribution** to the list.

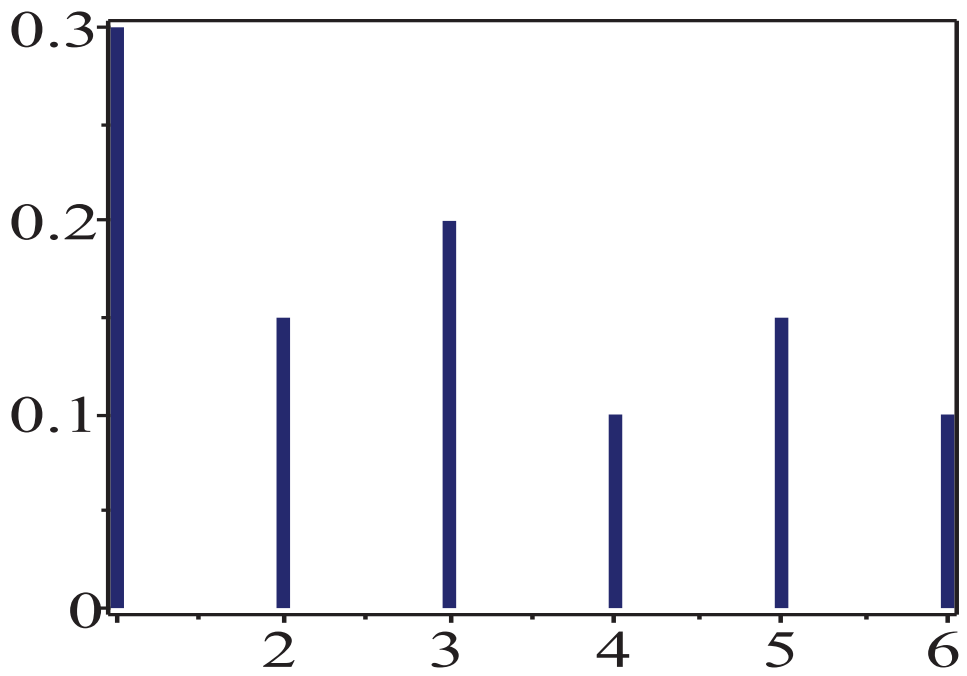
```
> dieRV := RandomVariable(EmpiricalDistribution(dieRolls)):
```

Now, you can compute probabilities and graph the distribution.

```
> Probability(dieRV ≥ 4)
```

$$\frac{7}{20} \tag{7.15}$$

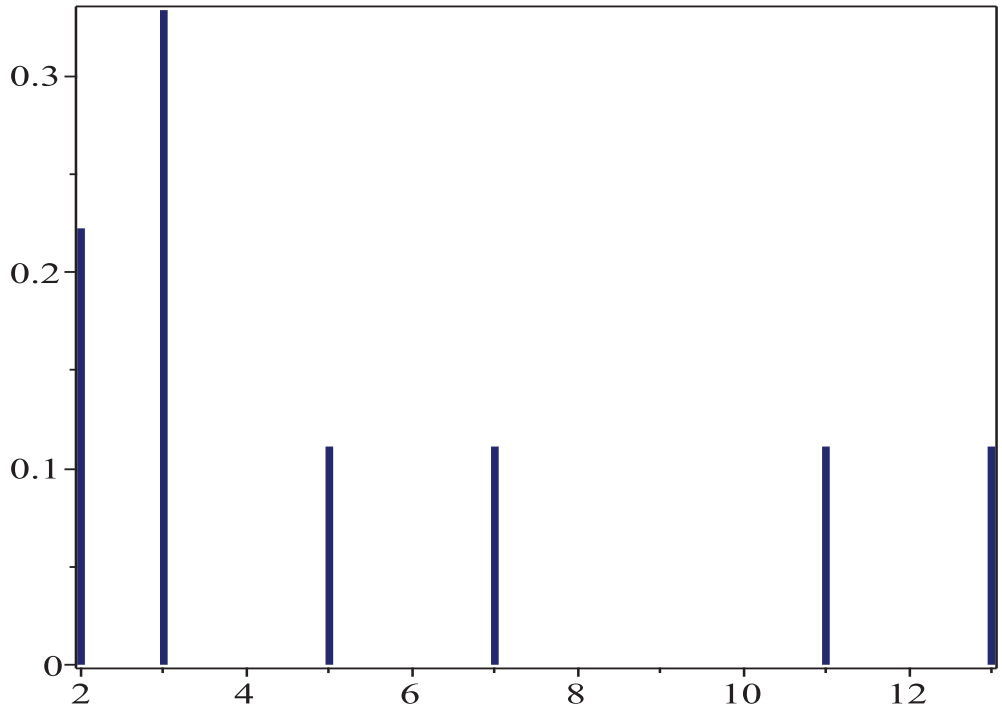
```
> DensityPlot(dieRV)
```



The **EmpiricalDistribution** command also allows you to specify the probabilities of each outcome by using the **probabilities** option. This is like **ProbabilityTable**, but more flexible in that **ProbabilityTable** insists that the outcomes be the positive integers. With **EmpiricalDistribution**, you can provide a list of values for the outcomes as the first argument and a list of probabilities as the value associated to the **probabilities** option. For example, the following creates a random variable whose values are the first six prime numbers.

```
> primeRV := RandomVariable(EmpiricalDistribution([2, 3, 5, 7, 11, 13],
      probabilities = [2/9, 1/3, 1/9, 1/9, 1/9, 1/9])):
```

```
> DensityPlot(primeRV)
```



Combining Random Variables

Most interesting questions in probability arise from combining random variables. For example, consider two loaded dice. One is weighted so that the probability that a 1 appears is $2/7$, and the probabilities of all other values are $1/7$. The other is weighted so that the probability of a 4 appearing is $3/8$, and the probabilities of all other values are $1/8$. What is the probability that the sum is 7 when the two dice are rolled?

To answer this question, we first define two random variables. Since the values of each die is 1 through 6, we can use the **ProbabilityTable** function to define the distribution.

```
> die1 := RandomVariable(ProbabilityTable([2/7, 1/7, 1/7, 1/7, 1/7, 1/7])):
```

```
> die2 := RandomVariable(ProbabilityTable([1/8, 1/8, 1/8, 3/8, 1/8, 1/8])):
```

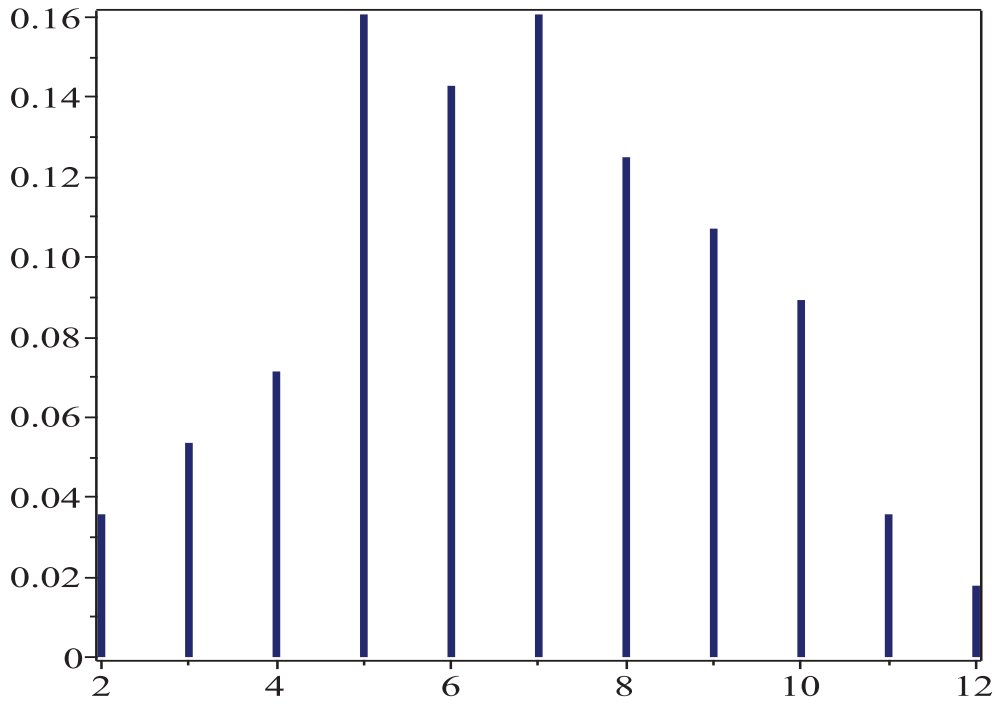
The question is about the sum of the values on the dice, so we apply **Probability** to the sum of the random variables.

```
> Probability(die1 + die2 = 7)
```

$$\frac{9}{56} \qquad \textbf{(7.16)}$$

We can also use the sum of the random variables as the argument to **DenisityPlot**.

```
> DensityPlot(die1 + die2)
```



Sampling

Once you have defined a random variable, you may wish to use it to conduct experiments or simulations. To do this, you use the **Sample** command.

Sample requires two arguments: a random variable or a distribution as the first argument, and a positive integer indicating the sample size as the second argument. It produces a row vector containing results obtained by randomly choosing values in accordance with the random variable.

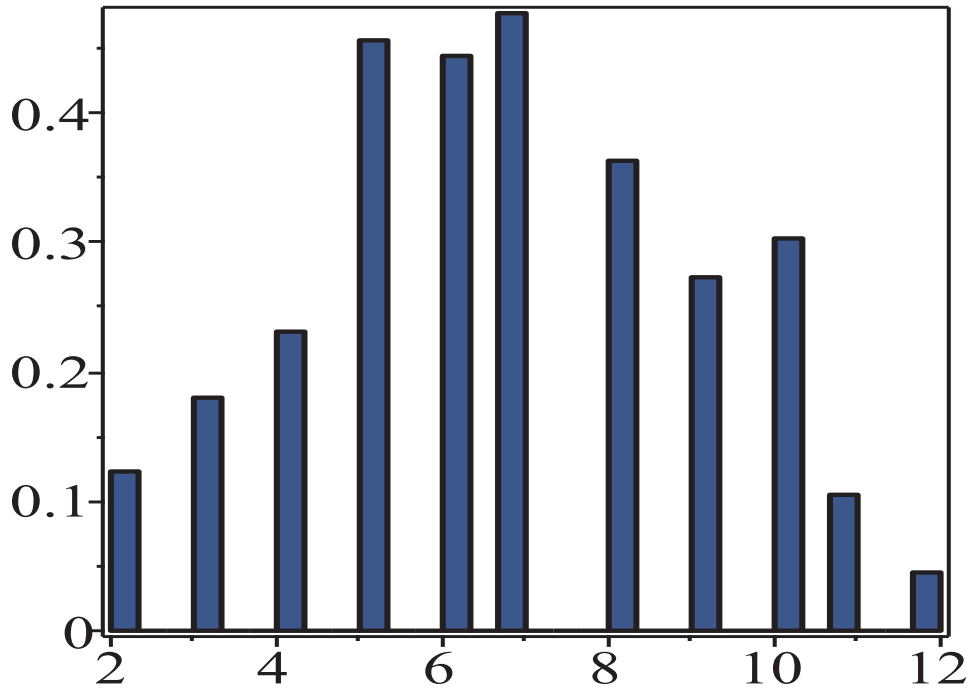
The following simulates rolling the pair of weighted dice **die1** and **die2** 1000 times.

```
> dieSample := Sample(die1 + die2, 1000);
```

$$dieSample := \begin{bmatrix} 1..1000\ Vector_{row} \\ Data\ Type: float_8 \\ Storage: rectangular \\ Order: Fortran\_order \end{bmatrix} \quad \textbf{(7.17)}$$

We can use the **Histogram** command to draw a histogram of the data. You see that the data produced has approximately the same distribution as the density plot of the sum of the variables.

```
> Histogram(dieSample)
```



Monte Carlo Methods

We can also implement Monte Carlo algorithms using Maple. Miller's test for base b is described in the preamble to Exercise 44 in Section 4.4 of the textbook. In that description, it is mentioned that a composite integer n passes Miller's test for base b for fewer than $n/4$ bases less than n, and Exercise 44 asked you to show that primes pass Miller's test for all bases that they do not divide. In other words, Miller's test is a probabilistic primality test that fails less than one-fourth of the time. In this section, we use Miller's test to create a Monte Carlo primality testing algorithm.

Miller's Test

First, we must implement Miller's test for base b. Recall the description preceding Exercise 44 in Section 4.4. Let n and b be positive integers. Assume s is a nonnegative integer and t is an odd positive integer such that $n-1 = 2^s t$. If $b^t \equiv 1 \pmod{n}$ or if there is a j with $0 \leq j \leq s-1$ such that $b^{2^j t} \equiv -1 \pmod{n}$, then n is said to pass Miller's test for base b.

To implement Miller's test, we first must calculate s and t. Initialize s to 0 and set t equal to $n-1$. If t is even, we add 1 to s and divide t by 2. When t is no longer even, then s and t are the correct values.

Once s and t have been calculated, we check the congruence $b^t \equiv 1 \pmod{n}$. If that congruence is satisfied, then n passes Miller's test and we return true. Otherwise, we begin testing the congruences $b^{2^j t} \equiv -1 \pmod{n}$. A for loop assigns j to each integer from 0 to $s-1$ and inside the for loop, the congruence is tested. If any congruence holds, the procedure returns true. (Recall from Section 4.1 of this manual that **modp** returns the smallest positive integer congruent to its first argument modulo its second argument. Thus we test for congruence to -1 modulo n by comparing the result to $n-1$.) If the procedure completes without having returned true, then it returns false.

```
1 Miller := proc(n::posint,b::posint)
2    local s, t, j;
```

```
    s := 0;
    t := n-1;
    while modp(t,2) = 0 do
       t := t/2;
       s := s + 1;
    end do;
    if modp(b^t,n) = 1 then
       return true;
    end if;
    for j from 0 to s-1 do
       if modp(b^(2^j*t),n) = n-1 then
          return true;
       end if;
    end do;
    return false;
end proc:
```

Monte Carlo Primality Test

Now, we use Miller's test to implement a Monte Carlo primality testing algorithm, as described in Example 16 in Section 7.2 of the text. The question the Monte Carlo algorithm is going to answer is "Is n composite?" for an integer n. For each iteration, the algorithm will select a random base b with $1 < b < n$ and check to see if n passes Miller's test for base b. If Miller's test returns false, then we know that n is composite and the Monte Carlo algorithm will return true, indicating that yes, n is composite. If Miller's test returns true, then the iteration results in "unknown" and the next iteration is started. After 30 iterations, if Miller's test has only resulted in true, then the algorithm will return false, indicating that it is very likely that the number is prime. Since Miller's test falsely identifies a composite as prime less than one-fourth of the time, the probability that the Monte Carlo algorithm will incorrectly identify a composite number as prime is

```
> (1./4)^30
```

$$8.673617380\ 10^{-19} \tag{7.18}$$

Here is the Miller Monte Carlo test:

```
MillerMC := proc(n::integer)::string;
    local gen, b;
    gen := rand(2..n-1);
    from 1 to 30 do
       b := gen();
       if (not Miller(n,b)) then return "composite" end if;
    end do;
    return "prime";
end proc:
```

Note the use of the **rand** command. This command, when passed a range like **2..n-1**, does *not* produce a random number between 2 and $n - 1$. Rather, when used this way, the **rand** command

returns a procedure that generates a random number in the specified range. In our algorithm, we assign the variable **gen** to the random number generator created by **rand(2..n-1)**, and the assignment **b := gen()** produces a random number between 2 and $n - 1$ and stores that value in **b**.

We use **MillerMC** to test a random integer to see if it is prime. We can use Maple's **ithprime** function to find the 40 000th prime and then check that our algorithm confirms that it is prime.

```
> ithprime (40 000)
                          479 909                                    (7.19)

> MillerMC (479 909)
                          "prime"                                    (7.20)
```

7.3 Bayes' Theorem

Section 7.3 focuses on applications of Bayes' theorem, which asserts that for events E and F from a sample space S with $p(E) \neq 0$ and $p(F) \neq 0$, one has

$$p(F \mid E) = \frac{p(E \mid F)\, p(F)}{p(E \mid F)\, p(F) + p\left(E \mid \overline{F}\right) p\left(\overline{F}\right)}.$$

The text describes how to use this theorem to create a Bayesian spam filter. We will use Maple to implement such a filter.

Recall the notation from the text. A message is received containing the word w. The event S will be the event that the message is spam and the event E is the event that the message contains the word w. If we assume that a message is as likely to be spam as not, so that $p(S) = p\left(\overline{S}\right) = 1/2$, then Bayes' theorem tells us that the probability that the incoming message is spam given that it contains the word w is:

$$p(S \mid E) = \frac{p(E \mid S)}{p(E \mid S) + p\left(E \mid \overline{S}\right)}.$$

By estimating the conditional probabilities with empirical data, we can compute an estimate that the given message is spam.

Before building the spam filter, we will first need messages to serve as spam and nonspam. For the spam messages, we will use the sonnets of William Shakespeare, and for the nonspam messages, we will use sonnets written by Shakespeare's contemporaries, Michael Drayton, Bartholomew Griffin, and William Smith, published in the book *Elizabethan Sonnet Cycles*.

It may seem strange to consider Shakespeare's sonnets to be spam, but consider the goals and methods of a Bayesian spam filter. The goal of a spam filter is to filter out the "junk mail." In the case of the Bayesian filter described by the text, these filters work by comparing the specific words used by authors of spam in contrast to authors of nonspam messages. Think about email messages you receive from your classmates versus messages your professors may send you. Chances are good

that you and your peers use more slang and generally less formal English when writing to each other than you and your professor use when communicating. This applies to kinds of message writers like peers versus professors, but it also can apply to individual message writers, like a mathematics professor versus a literature professor. A literature professor, for example, is not likely to use words like "Bayes' theorem" in an email to you. A Bayesian spam filter can pick up on these differences in word choice and effectively filter messages based on the assumption that different authors generally use different words. We will see, by comparing Shakespeare with other Elizabethan sonnet writers, that a Bayesian filter can even distinguish one author from others writing at the same time, for the same audience, and in a very similar style.

Obtaining Data

On the website for this manual, you will find these three files: "ShakespeareData.txt", "ElizabethanData.txt" and "testMessages.txt". The first two contain the sonnets of Shakespeare and the other authors, respectively. Five of Shakespeare's poems and five of the other authors' poems were randomly selected and moved to the "testMessages.txt" file. We will use our "spam" filter on the poems in this file to determine which of them were written by Shakespeare and which were not.

Begin by downloading the three files and storing them in the same directory as this Maple Worksheet. Then, load the three files and store the text in variables using the **ReadFile** command. Note that the commands below are set up to obtain the directory in which this Worksheet is saved.

```
> shakespeare := FileTools[Text][ReadFile](
    FileTools[JoinPath](["ShakespeareData.txt"], base = worksheetdir)) :

> elizabethan := FileTools[Text][ReadFile](
    FileTools[JoinPath](["ElizabethanData.txt"], base = worksheetdir)) :

> test := FileTools[Text][ReadFile](
    FileTools[JoinPath](["testMessages.txt"], base = worksheetdir)) :
```

If your worksheet is not in the same location as the text files, you may need to replace the argument of **ReadFile** with the full path. The command below will certainly not work on your computer, but illustrates the command.

```
> test := FileTools[Text][ReadFile](
    "/Users/danieljordan/DiscreteMath/testMessages.txt")
```

The following illustrates how to have Maple produce a file dialog window in order to load the file. Note that this input has been made nonexecutable, so that it will not create the dialog if you execute the entire worksheet.

```
> test := FileTools[Text][ReadFile](Maplets[Utilities][GetFile](
    'title' = "Open Text File", 'directory' = currentdir(homedir),
    'filefilter' = "txt")) :
```

If you inspect the files in a text editor, you will see that the sonnets are separated by three ampersands ("&&&"). The three commands above store each of the files as a single string. It would be more useful to store them as lists of strings, with each sonnet being one element of a list. To separate the files into lists, we use **RegSplit** from the **StringTools** package.

```
> SPoems := [StringTools[RegSplit]("&&&", shakespeare)] :

> EPoems := [StringTools[RegSplit]("&&&", elizabethan)] :

> testPoems := [StringTools[RegSplit]("&&&", test)] :
```

The **RegSplit** command splits the string in the second argument based on the pattern given in the first argument. In this case, the pattern is the string "&&&" so the **RegSplit** command uses that pattern as a delimiter in the **shakespeare**, **elizabethan**, and **test** strings to separate them into lists. The pattern can be a string, as we use it here, or it can be aregular expression, hence the name **RegSplit**. Now that the "messages" are prepared, we begin building the filter.

Estimating the Probabilities

The spam filter relies on two computations: first, the probability that a message contains a word given that it is spam, and second, the probability that a message contains the word given that it is not spam. That is, we will need empirical estimates for $p(E \mid S)$ and $p\left(E \mid \overline{S}\right)$.

Following the notation of the textbook, for a word w, let $p(w)$ be the estimate of $p(E \mid S)$, the probability that a message contains w given that it is spam. Therefore, $p(w)$ is the number of spam messages containing the word w divided by the number of spam messages. Likewise, let $q(w)$ be the estimate for $p\left(E \mid \overline{S}\right)$, the probability that a message contains w given that it is not spam. This is computed as the number of nonspam messages containing w divided by the number of nonspam messages.

Counting the number of messages (i.e., poems) in each list can be done with **nops**.

```
> nops(SPoems)
    149                                                    (7.21)
```

(This is five less than the 154 sonnets that Shakespeare published, because five of them were moved to the "testMessages.txt" file as "unknown" messages.)

To count the number of messages that contain a particular word, we make use of two functions. First, we use the **Words** command in the **StringTools** package to separate a string into its component words and remove punctuation. For example:

```
> exampleWords := StringTools[Words]("To count the number of
    messages that contain a particular word we'll make use of two functions")
    exampleWords := ["To", "count", "the", "number", "of", "messages",
      "that", "contain", "a", "particular", "word", "we'll", "make", "use",
      "of", "two", "functions"]                            (7.22)
```

Second, we use the **ListTools Search** function to determine if a message contains a particular word. **Search(element, L)** returns the index of the first occurrence of **element** in the list **L**. If the element is not in the list, then it returns 0. For example:

```
> ListTools[Search]("of", exampleWords)
    5                                                      (7.23)
```

```
> ListTools[Search]("elephant", exampleWords)
```

0 **(7.24)**

Putting these together, we can create a procedure for counting the number of times a word appears in a list of messages as follows. For each message in the list, we use the **Words** function to separate the message into words. Then, we use the **Search** function to see if the word we are looking for is in the sonnet. If the **Search** function returns a value greater than 0, then we know the word is in the message and we increment a counter. Here is the procedure:

```
countMessages := proc(w::string, L::list)
   local count, m, P;
   count := 0;
   for m in L do
      P := StringTools[Words](m);
      if (ListTools[Search](w,P) > 0) then
         count := count + 1;
      end if;
   end do;
   return count;
end proc:
```

For instance, we can see in how many sonnets Shakespeare uses the word "fairest":

```
> countMessages("fairest", SPoems)
```

4 **(7.25)**

The empirical probability that a sonnet contains the word "fairest" given that it was written by Shakespeare is:

```
> countMessages("fairest", SPoems) / nops(SPoems)
```

$$\frac{4}{149} \qquad \textbf{(7.26)}$$

The probability that a sonnet contains the word "fairest" given that it was written by one of our other authors is:

```
> countMessages("fairest", EPoems) / nops(EPoems)
```

$$\frac{10}{173} \qquad \textbf{(7.27)}$$

Applying Bayes' theorem, we can compute the probability that a sonnet was written by Shakespeare given that it contains the word "fairest":

```
> evalf( (7.26) / ((7.26) + (7.27)) )
```

0.3171402383 **(7.28)**

The above computation illustrates how to write a procedure to compute the probability that a sonnet is spam (i.e., "written by Shakespeare") given that it contains a specific word:

```
PShakespeareGivenWord := proc(w::string)
    local SCount,ECount,PWordGivenS,PWordGivenNotS;
    global SPoems, EPoems;
    SCount := nops(SPoems);
    ECount := nops(EPoems);
    PWordGivenS := countMessages(w,SPoems)/SCount;
    PWordGivenNotS := countMessages(w,EPoems)/ECount;
    return evalf(PWordGivenS/(PWordGivenS + PWordGivenNotS));
end proc:
```

For example, the probability that a sonnet is Shakespearean given that it contains the word "beauty" is:

> *PShakespeareGivenWord*("beauty")

0.5601371298 **(7.29)**

Using Multiple Words

We can improve the filter by using multiple words, rather than just one. Using the notation of the text, let $p(w_i)$ and $q(w_i)$ be the probability that a message contains word w_i given that it is spam and that it is not spam, respectively. Then, the probability that a message is spam given that it contains all of the words $w_1, w_2, \ldots, w_k$ is:

$$r(w_1, w_2, .., w_k) = \frac{\prod_{i=1}^{k} p(w_i)}{\prod_{i=1}^{k} p(w_i) + \prod_{i=1}^{k} q(w_i)}.$$

The **mul** command is useful here. Recall that we compute $\prod_{i \in S} i^2$ for $S = \{1, 3, 5, 7, 9\}$, with:

> $S := [1, 3, 5, 7, 9]$:

> $mul\left(i^2, i \textbf{ in } S\right)$

893 025 **(7.30)**

For instance, to compute the probability that a message contains the words "from", "fairest", and "creatures",

> $S := [\text{"from"}, \text{"fairest"}, \text{"creatures"}]$:

> $mul\left(\frac{countMessages(w, SPoems)}{nops(SPoems)}, w \textbf{ in } S\right)$

$$\frac{416}{3\,307\,949} \quad \textbf{(7.31)}$$

We can modify our **PShakespeareGivenWord** procedure to work on lists of words instead of single words by putting the probability computations inside of **mul** commands. It is also a good idea to protect against division by zero errors, so we will put the division inside of an if statement. This is needed in case one or more of the selected words appears in none of the sonnets by either Shakespeare or the other authors. In this case, we default to a probability of 0.5.

```
PShakespeareGivenList := proc(L::list)
   local SCount, ECount, PGivenS, w, PGivenNotS;
   global SPoems, EPoems;
   SCount := nops(SPoems);
   ECount := nops(EPoems);
   PGivenS := mul(countMessages(w,SPoems)/SCount,w in L);
   PGivenNotS := mul(countMessages(w,EPoems)/ECount,w in L);
   if (PGivenS + PGivenNotS <> 0) then
      return evalf(PGivenS/(PGivenS + PGivenNotS));
   else
      return 0.5;
   end if;
end proc:
```

Therefore, the probability that a sonnet is by Shakespeare given that it contains the words "from", "fairest", and "creatures" is:

> *PShakespeareGivenList*(["from", "fairest", "creatures"])

0.5559873068 **(7.32)**

Selecting Test Words Randomly

Finally, we can use the **randcomb** command to randomly select words from a test message, and then use those randomly selected words to compute the probability that the message was written by Shakespeare. Here is the first test message in "testMessages.txt":

> *testPoems*[1]

"When to the sessions of sweet silent thought
I summon up remembrance of things past,
I sigh the lack of many a thing I sought,
And with old woes new wail my dear time's waste:
Then can I drown an eye, unused to flow,
For precious friends hid in death's dateless night,
And weep afresh love's long since cancell'd woe,
And moan the expense of many a vanish'd sight:
Then can I grieve at grievances foregone,
And heavily from woe to woe tell o'er
The sad account of fore-bemoaned moan,
Which I new pay as if not paid before.
But if the while I think on thee, dear friend,
All losses are restor'd and sorrows end.
" **(7.33)**

We use the **Words** command to separate the poem into individual words:

```
> exampleTestWords := StringTools[Words](testPoems[1])
```

exampleTestWords := ["When", "to", "the", "sessions", "of", "sweet",
"silent", "thought", "I", "summon", "up", "remembrance", "of",
"things", "past", "I", "sigh", "the", "lack", "of", "many", "a",
"thing", "I", "sought", "And", "with", "old", "woes", "new", "wail",
"my", "dear", "time's", "waste", "Then", "can", "I", "drown",
"an", "eye", "unused", "to", "flow", "For", "precious", "friends", "hid",
"in", "death's", "dateless", "night", "And", "weep", "afresh", "love's",
"long", "since", "cancell'd", "woe", "And", "moan", "the", "expense",
"of", "many", "a", "vanish'd", "sight", "Then", "can", "I", "grieve",
"at", "grievances", "foregone", "And", "heavily", "from", "woe", "to",
"woe", "tell", "o'er", "The", "sad", "account", "of", "fore",
"bemoaned", "moan", "Which", "I", "new", "pay", "as", "if", "not",
"paid", "before", "But", "if", "the", "while", "I", "think", "on",
"thee", "dear", "friend", "All", "losses", "are", "restor'd", "and",
"sorrows", "end"] **(7.34)**

Randomly select four of those words:

```
> exampleTestList := combinat[randcomb](exampleTestWords, 4)
```

exampleTestList := ["lack", "love's", "think", "All"] **(7.35)**

Now, use our procedure to find the probability that a message with these four words was written by Shakespeare:

```
> PShakespeareGivenList(exampleTestList)
```

0.7329839295 **(7.36)**

Putting this all together:

```
PShakespeare := proc(testMessage::string,testSize::integer)
   local testWordList;
   testWordList :=
      combinat[randcomb](StringTools[Words](testMessage),
      testSize);
   return PShakespeareGivenList(testWordList);
end proc:
```

As an example, we run the filter on the second test message with a test size of 3.

```
> PShakespeare(testPoems[2], 3)
```

0.3805889954 **(7.37)**

7.4 Expected Value and Variance

In Section 7.2 of this manual, we introduced Maple's commands for using random variables. In this section, we explore Maples's **Statistics** package more closely and use random variables to explore the concepts of expected value and variance.

As mentioned earlier, the **Statistics** package provides the distribution **Geometric**, which takes one parameter, the probability of a "success."

```
> X := RandomVariable(Geometric(1/4))
```

$$X := \_R9 \tag{7.38}$$

We use the **Probability** command to compute probabilities of events. For example, the probability $p(X = 5)$ is computed by:

```
> Probability(X = 5)
```

$$\frac{243}{4096} \tag{7.39}$$

Note that Maple's definition of a geometric random variable differs slightly from the textbook's. The textbook defines the value of the geometric random variable, in terms of coin flips, to be the number of flips it takes to get a tails, where the parameter is the probability of tails. Maple's definition is that the value of the random variable is the number of heads that appear before tails comes up. Thus, the probability $p(X = k)$ is

```
> Probability(X = k)
```

$$\begin{cases} 0 & k < 0 \\ \frac{\left(\frac{3}{4}\right)^k}{4} & otherwise \end{cases} \tag{7.40}$$

Contrast this with the formula given in the text.

The **Statistics** package also includes commands for computing the expected value, variance, and standard deviation of a random variable:

```
> ExpectedValue(X)
```

$$3 \tag{7.41}$$

```
> Variance(X)
```

$$12 \tag{7.42}$$

Maple can also compute these symbolically, if we use an unassigned name for the argument to **Geometric**.

```
> Y := RandomVariable(Geometric(p)):
```

```
> ExpectedValue(Y)
```

$$\frac{1-p}{p} \tag{7.43}$$

> *Variance* (Y)

$$\frac{1-p}{p^2} \tag{7.44}$$

> *StandardDeviation* (Y)

$$\frac{\sqrt{1-p}}{p} \tag{7.45}$$

Notice that the expected value differs from the expected value of a geometric distribution given in the text. This is because of the difference in definitions mentioned previously. Since the difference between Maple's definition and the textbook's definition is that Maple's geometric random variables have values one less than the textbooks, we can create a random variable Z that has the geometric distribution defined by the textbook by adding one to a random variable created by Maple:

> $Z := RandomVariable(Geometric(p)) + 1$:

We check that Z agrees with the formula given by the text, namely $p(Z = k) = (1-p)^{k-1}$ for $k = 1, 2, 3, \ldots$

> *Probability* $(Z = k)$

$$\begin{cases} 0 & k < 1 \\ p(1-p)^{k-1} & otherwise \end{cases} \tag{7.46}$$

We also confirm that the expected value agrees with the text:

> *ExpectedValue* (Z)

$$-\frac{-1+p}{p} + 1 \tag{7.47}$$

Finally, you can also use these functions with combinations of random variables, as illustrated below.

> $Rvariable1 := RandomVariable\left(Geometric\left(\frac{1}{4}\right)\right)$:

> $Rvariable2 := RandomVariable(Binomial(20, 0.3))$:

> *ExpectedValue* $(Rvariable1 + 2 \cdot Rvariable2)$

$$15.00000000 \tag{7.48}$$

> *Variance* $(Rvariable1 + 2 \cdot Rvariable2)$

$$28.80000000 \tag{7.49}$$

Solutions to Computer Projects and Computations and Explorations

Computer Projects 7

Given a positive integer m, simulate the collection of cards that come with the purchase of products to find the number of products that must be purchased to obtain a full set of m different collector cards. (See Supplementary Exercise 33.)

Solution: We will define a procedure called **CardSimulate** that will simulate the process of choosing random collectible cards until all the possible cards have been obtained. This procedure needs to do three things: (1) keep track of which cards have been obtained; (2) keep selecting random cards until the complete set is obtained; and (3) keep track of how many cards have been purchased.

Think of the cards as numbered 1 through *m*. To keep track of which cards have been obtained and which have not, we will use a list that we call **currCollection**, for current collection. The entries in this list will be 0s and 1s, with a 0 representing the fact that the card corresponding to that position is not owned and 1 that it is. To initialize **currCollection**, we use the **$** operator.

```
> [0$10]
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]                                    (7.50)
```

Recall that when the dollar operator is given a positive integer as its right operand produces that number of copies of its left operand.

Second, random selection of cards can be accomplished by the **rand** command. The optional argument to **rand** specifies the range of values that it will return. Since there are *m* possible collectible cards, we use **rand(1..m)**. With no argument, **rand()** returns a number, but with an argument, it creates a procedure. Therefore, we assign a name to the procedure that the **rand** command creates and then use that name to make random cards. For example:

```
> exampleRand := rand(1..100)
    exampleRand := proc()
      proc() option builtin = RandNumberInterface; end proc(6, 100, 7) + 1
    end proc                                                          (7.51)

> exampleRand()
    6                                                                 (7.52)
```

Our procedure will generate a random card and set the entry in **currCollection** at that card's position equal to 1. This needs to keep happening until all the cards are owned. Therefore, we need to know when all of the entries of the list are 1s. We can do this by adding up the entries in the list. Since the entries are always 0 or 1, when the list is all 1s, the sum will be equal to *m* and that is the only way the sum can be *m*. To add the entries in the list, we can use the **add** command as follows:

```
> exampleList := [1, 0, 0, 1, 1, 1]:

> add(exampleList)
    4                                                                 (7.53)
```

Third, we keep track of how many cards have been purchased with a counter that we increment each time a random card is generated.

Putting all of these pieces together, here is the procedure:

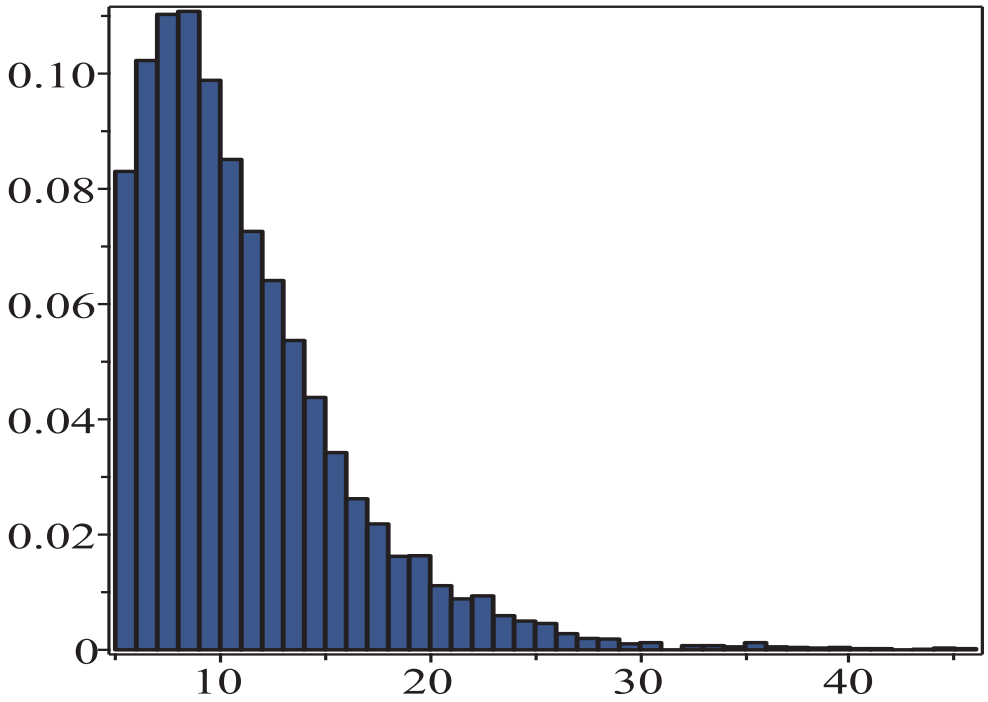
```
1 CardSimulate := proc(m::integer)
2   local currCollection, i, count, tempCard, cardGen;
3   currCollection := [0$m];
4   count := 0;
```

```
    cardGen := rand(1..m);
    while add(currCollection) < m do
      tempCard := cardGen();
      count := count + 1;
      currCollection[tempCard] := 1;
    end do;
    return count;
end proc:
```

Run the simulation 10 000 times for $m = 5$ and draw a histogram of the resulting data:

```
> simulations := [seq(CardSimulate(5), i = 1..10 000)]:
```

```
> Statistics[Histogram](simulations, binwidth = 1)
```



Computer Projects 9

Given a positive integer n, find the probability of selecting the six integers from the set $\{1, 2, n, ..\}$ that were mechanically selected in a lottery.

Solution: We will follow Example 4 in Section 7.1 of the main text. The total number of ways of choosing 6 numbers from n numbers is $C(n, 6)$, which is found with the procedure **numbcomb** in the **combinat** package. This gives us the total number of possibilities, only one of which will win.

```
lottery := proc(n::posint)
  local total;
  total := combinat[numbcomb](n,6);
  1.0 / total;
end proc:
```

> *lottery*(49)

$$7.151123842\ 10^{-8} \tag{7.54}$$

If the rules of the lottery change, so that the number of numbers chosen in something other than 6, then we must modify the procedure above. We can easily modify our program to allow us to specify how many numbers we want to choose, by adding another parameter.

```
lottery2 := proc(n::posint, k::posint)
   local total;
   total := combinat[numbcomb](n,k);
   1.0 / total;
end proc:
```

> *lottery2*(49, 6)

$$7.151123842\ 10^{-8} \tag{7.55}$$

> *lottery2*(30, 3)

$$0.0002463054187 \tag{7.56}$$

Computations and Explorations 3

> Estimate the probability that two integers selected at random are relatively prime by testing a large number of randomly selected pairs of integers. Look up the theorem that gives this probability and compare your results with the correct probability.

Solution: To solve this problem, three things must be done:
1. Devise a method for generating pairs of random integers.
2. Produce a large number of these pairs, test whether they are relatively prime, and note the probability estimate based on this sample.
3. Look up the theorem mentioned in the question.

Naturally, part 3 is left to the reader.

A simple approach is to use the Maple procedure **rand** to generate a list of random integers. Then, having generated such a list we can test whether the pairs of its members are coprime using the Maple procedure **igcd** in a second loop. We implement these two loops in a new Maple procedure called **RandPairs**:

```
RandPairs := proc(numberPairs::integer)
   local listSize, i, tmp, randnums, count;
   listSize := 2 * numberPairs;
   randnums := [];
   # Generate list of random integers
   for i from 1 to listSize do
      tmp := rand();
      randnums := [op(randnums),tmp];
   end do;
   # Count the number of pairs that are coprime
   count := 0;
   for i from 1 to listSize-1 by 2 do
```

```
        if igcd(randnums[i],randnums[i+1]) = 1 then
          count := count + 1;
        end if;
    end do;
    evalf(count / numberPairs);
end proc:
```

We can now execute this procedure on 100 pairs of integers, as follows:

```
> RandPairs(100)
    0.6000000000                                                    (7.57)
```

Note that repeating the computation may very well lead to a somewhat different result since the list of integers we used was generated randomly. You should try this with a much larger sample size, say 10 000 pairs of integers.

Computations and Explorations 4

> Determine the number of people needed to ensure that the probability at least two of them have the same day of the year as their birthday is at least 70%, at least 80%, at least 90%, at least 95%, at least 98%, and at least 99%.

Solution: Given that we know the formula for the probability of two people having the same birthday, we can use Maple to loop over a range of possible numbers of people until we reach a probability greater than the desired probability. Example 13 in Section 7.2 of the text shows that the probability that n people in a room have different birthdays is

$$p_n = \frac{365}{366}\frac{364}{366}\frac{363}{366}\cdots\frac{367-n}{366} = \frac{P(366,n)}{366^n}.$$

Our task is to find n such that $1 - p_n$ is greater than the values specified in the problem. We can do this using the Maple procedure below.

```
Birthdays := proc(percentage::float)
    local numPeople, curProb;
    #  Initialize
    curProb := 0;
    numPeople := 0;
    # loop  until  there  are  enough people
    while curProb < percentage do
        numPeople := numPeople + 1;
        curProb := 1 -
            (combinat[numbperm](366,numPeople)/366^numPeople);
    end do;
    return numPeople;
end proc:
```

This procedure returns the number of people required to attain the given probability that two have the same birthday. We now execute our procedure for probabilities of 0.70 and 0.95.

```
> Birthdays(0.70)
                    30                                              (7.58)

> Birthdays(0.95)
                    47                                              (7.59)
```

Exercises

Exercise 1. Use Maple to determine the integer k such that the chances of picking six numbers correctly in a lottery from the first k positive integers is less than

a) 1 in 100 million (10^{-8}),
b) 1 in a billion (10^{-9}),
c) 1 in 10 billion (10^{-10}),
d) 1 in 100 billion (10^{-11}), and
e) 1 in a trillion (10^{-12}).

Exercise 2. Implement a Monte Carlo algorithm that determines whether a permutation of the integers 1 through n has already been sorted or is a random permutation. (See Exercise 40 in Section 7.2 of the textbook.)

Exercise 3. Modify the implementation of the collector card simulator given in the solution to Computer Projects 7 to model the situation in which the cards do not appear with equal probabilities. For instance, there could be five possible cards all of which appear with probability 2/9 except for card number 5 which appears with probability 1/9.

Exercise 4. Modify the implementation of the collector card simulator given in the solution to Computer Projects 7 to model the situation in which cards are purchased in packs. For example, there could be 10 possible cards and they are purchased three to a pack. Assume the cards in a pack are always different from each other. The procedure should return the number of packs necessary to collect all of the cards.

Exercise 5. Compute the average of the probabilities returned by running the Bayesian filter **PShakespeare** 100 times with the **testMessage** argument equal to **testPoems[10]** and a **testSize** of 1, that is, on the tenth of the test poems and using one word. Repeat this with a **testSize** of 2, 3, ..., 10. Graph the average probabilities for the different numbers of test words. Is there a trend in the average probabilities as the number of words increases? Explain why.

Exercise 6. The textbook describes how a Bayesian filter can be improved by considering pairs of words. Implement a Bayesian spam filter that uses this idea. Using the Shakespearean and Elizabethan sonnets as messages, compare the performance of your filter with **PShakespeare**.

Exercise 7. As described in the textbook, spam filters are most effective when the words being used as the basis of comparison are not chosen randomly, as they are in the implementation of **PShakespeare** above, but instead are chosen more carefully. Specifically, choosing words which have very high or very low probability of appearing in spam messages can improve the performance of the filter. Implement a Bayesian spam filter that uses this idea. Using the Shakespearean and Elizabethan sonnets as messages, compare the performance of your filter with **PShakespeare**.