Chapter 4

Probability simulations

4.1 Declaring data types for program arguments

Maple classifies data into the following types (ask Maple: ?type)

```
!         .          And       NONNEGATIVE Not
Or        PLOT       PLOT3D    Point       Range
RootOf    TEXT       `**`      `*`         `+`
`^`       algebraic  algext    algfun      algnum
algnumext anyfunc    anything  arctrig     array
atomic    boolean    complex   complexcons constant
cubic     dependent  disjcyc   equation    even
evenfunc  expanded   exprseq   facint      float
fraction  freeof     function  hfarray     identical
indexed   indexedfun infinity  integer     intersect
laurent   linear     list      listlist    literal
logical   mathfunc   matrix    minus       monomial
name      negative   negint    nonneg      nonnegint
nonposint nothing    numeric   odd         oddfunc
operator  point      polynom   posint      positive
prime     procedure  protected quadratic   quartic
radalgfun radalgnum  radext    radfun      radfunext
radical   radnum     radnumext range       rational
ratpoly   realcons   relation  rgf_seq     scalar
```

Among them, the following types are frequently used in this book:

```
array, numeric, integer, odd, even, positive, negative
```

When writing a program, usually we want each argument to be a certain type data, and we would like the machine to check if the input item fits the data type. For example, in the program of finding quartiles in the array members

$$x_m, x_{m+1}, \cdots, x_n$$

we can use three arguments m, n, x as integer, integer, and array respectively. Then we can start the program definition like

```
quartiles:=proc(m::integer, n::integer, x::array)
```

When executing the program, Maple will automatically check each input item to see if it fits the corresponding type and, if not, output an error message.

Example: The following program calculates the square root of the sum of three real numbers.

```
sqrt3:=proc(
            a::numeric,   # input: real number
            b::numeric,   # same as above
            c::numeric,   # same as above
          ans::evaln      # output: must be declared "evaln"
            );

   ans:=sqrt(a+b+c);

end;
```

```
> read(`a:/txt/3sqrt.txt`):

> sqrt3(3,5,8,t);
```

$$4$$

```
> sqrt3(s,5,7,ans);

Error, sqrt3 expects its 1st argument, a, to be of type numeric, but
received s
```

As shown in this example, if an argument is designated as an output item, it must be declared "evaln".

4.2 Probability experiment

4.2.1 Introduction

In probability theory, an **experiment** is an activity or occurrence with an observable result, such as drawing a card from a deck. Each repetition of an experiment is called a **trial**. For example, there are three white balls and five red balls in a box. Drawing a ball from the box, one can observe its color. So drawing a ball from a box is an experiment. If one replace the ball back to the box after it is drawn, this experiment is called *drawing with replacement*. If that's the case, there can be infinitely many trials for the experiment.

If we ask a question: what is the probability of drawing, with replacement, three consecutive red balls? We may answer the question theoretically: $p = (\frac{5}{8})^3$, or we may set a program and ask Maple to simulate the experiment and calculate the probability approximately.

4.2.2 Random number generators

4.2.3 Random real number generator

The function rand() generates a 12 digit random positive integer. The following program generates a random real number in the interval $[a, b]$:

```
#
# function that output a random real number in [a,b]
#
real_ran:=proc(
               a::numeric,
               b::numeric
              )
   local x;
   x:=rand();
   evalf( a+ (b-a)*x/999999999999 );

end;
```

```
> read(`a:/txt/realran.txt`):
```

```
> s:=real_ran(-3,2);
```

$$s := 1.603124737$$

4.2.4 Random integer generator

The following function generates an integer between m and n, inclusive

```
#
# program generates a random integer between m and n, inclusive
#
int_ran:=proc(
              m::integer,
              n::integer
             )

   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))

end;
```

```
> read(`a:/txt/intran.txt`):
> k:=int_ran(1,52);
```

$$k := 50$$

```
> m:=int_ran(1,52);
```

$$m := 8$$

4.2.5 Example: Drawing a ball with replacement

Suppose in a box, there are 3 white balls and 5 red balls. We can consider balls numbered from 1 to 3 are white and those numbered from 4 to 8 are red. Randomly drawing a ball, mathematically, is equivalent to drawing a number from 1 to 8, and we'll know the color according to the number. Therefore, the experiment of "drawing a ball" can be simulated by the program

```
one_ball:=proc()
   local k;
   k:=int_ran(1,8);
   if k<=3 then
      "white";
   else
      "red";
   fi;
end;

int_ran:=proc(
              m::integer,
              n::integer
             )

   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))

end;
```

Program note

- The program one_ball does not require any input.
- The program int_ran is used by one_ball as a "subprogram". Therefore it should be included in the same text file. When the file is read into Maple, both main program and subprogram will be loaded.

Test run:

```
> read(`a:oneball.txt`):
> one_ball();
```

"*white*"

```
> one_ball();
```

"*red*"

4.2.6 Example: Drawing, say 3, balls without replacement

Suppose there are 8 balls in a box, 3 white and 5 red. Generally, if we draw k balls from the box *without replacement*, the experiment can be simulated in the following way:

(1) number balls from 1 to 8. The first three numbers correponds to while, remaining for red.

(2) Draw k balls one by one:

- The first draw is from ball 1 to 8. Then swap the drawn ball to number 8 (=8-1+1)
- The 2nd draw is from ball 1 to 7. Then swap the drawn ball to number 7 (=8-2+1)
- The 3rd draw is from ball 1 to 6. Then swap the drawn ball to number 6 (=8-3+1)
- Generally, the j-th drawn is from ball 1 to $n = 8 - j + 1$, then swap the drawn ball to number n so it won't be redrawn.

(4) Translate numbers to colors.

```
#
#  Program simulating drawing k balls in a box
#  containing 3 white balls and 5 red balls
#
draw_k_ball:=proc(
```

```
              k::integer, # number of balls to draw
           draw::evaln    # output: colors drawn
              )
   local ball, i, j, n, m, temp;

   ball:=array(1..8);      # making a box of balls
   for i to 3 do
      ball[i]:="white";
   od;
   for i from 4 to 8 do
      ball[i]:="red";
   od;

   draw:=array(1..k);      # open space for balls to be drawn

   n:=8;                   # n is the number of balls remaining
                           #    there are 8 balls initially
   for j to k do           # loop to draw balls one by one

      m:=int_ran(1,n);     # draw a number from 1 to n

      draw[j]:=ball[m];    # get the color

      if m <> n and ball[m]<>ball[n] then
         temp:=ball[m];    # move the drawn ball to end of pool
         ball[m]:=ball[n]; # so it won't be redrawn
         ball[n]:=temp;
      fi;

      n:=n-1;              # update n

   od;

end;
```

```
> read(`a:drawball.txt`):
> draw_k_ball(3,ball):
> eval(ball);
```

$$["red", "white", "white"]$$

4.2.7 Example: Approximate a probability

If we want to calculate the probability of drawing 3 balls without replacement and getting all red, we can simulate the experiment n times, count the successful drawings and use

$$\frac{number\_of\_successful\_drawings}{n}$$

as approximate value of the probability.

```
#
# program that calculate the approximate probability of
```

```
# drawing k balls, without replacement, out a box containing 3 red and 5
# white and  getting all red
#
# input:  k --- the number of balls to be drawn
#         n --- the number of trials to repeat
#
draw_test:=proc(  k::integer,
                  n::integer
               )
   local i, j, count, ball, flag;

   ball:=array(1..k);     # open the space for balls
   count:=0;              # initialize counting
   #
   # repete the experiment n times
   #
   for i to n do
      draw_k_ball(k,ball);   # draw k balls w/o replacement
      flag:=0;               # initialize flag=0, assuming successful
      for j to k while flag=0 do   # check balls if they are red
         if ball[j]="white" then
            flag:=1;               # raise the flag: a white ball is found
         fi;
      od;
      #
      # count a successful drawing if all red (flag=0)
      #
      if flag=0 then
         count:=count+1;
      fi;
   od;

   evalf(count/n);    # approximate probability

end;
#
#  Program simulating drawing k balls in a box
#  containing 3 white balls and 5 red balls
#
draw_k_ball:=proc(
                  k::integer, # number of balls to draw
               draw::evaln    # output: colors drawn
                 )
   local ball, i, j, n, m, temp;

   ball:=array(1..8);      # making a box of balls
   for i to 3 do
      ball[i]:="white";
   od;
   for i from 4 to 8 do
      ball[i]:="red";
   od;

   draw:=array(1..k);      # open space for balls to be drawn

   n:=8;                   # n is the number of balls remaining
                           #    there are 8 balls initially
```

```
   for i to k do           # loop to draw balls one by one

      m:=int_ran(1,n);     # draw a number from 1 to n

      draw[i]:=ball[m];    # get the color

      if m <> n and ball[m]<>ball[n] then
         temp:=ball[m];    # move the drawn ball to end of pool
         ball[m]:=ball[n]; # so it won't be redrawn
         ball[n]:=temp;
      fi;

      n:=n-1;              # update n

   od;

end;
#
# function that generates a random integer from m to n, inclusive
#
int_ran:=proc(
              m::integer,
              n::integer
             )
   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))
end;
```

```
> read(`a:drawtest.txt`):
> draw_test(3,300);
```

$$.1700000000$$

The exact probability is

```
> evalf((5!)^2/((2!)*(8!)));
```

$$.1785714286$$

Now we increase the number of trials to 10000

```
> draw_test(3,10000);
```

$$.1758000000$$

We got much better approximation. When the number of trials are bigger, it is *more likely*, though no guarantee, to get a more accurate approximation of the probability.

4.2.8 Example: Two points in a square having distance less than one

Given a square of 1 ft by 1 ft, what is the probability of throwing two stones in to the box having distance less than one? It is not easy to calculated the probability exactly. So let's write a program to approximate it.

A 1x1 square can be seen as set of points { (x, y) — $0 \leq x \leq 1$, and $0 \leq y \leq 1$,} on the xy-coordinate system. A randomly thrown stone can be represented as a point (x, y) in that set. The simulation of the experiment is to generate two sets of random numbers x and y in the interval [0,1].

```
#
# program that approximates the probability of throwing two points
# in a 1x1 square having distance less than 1
#
#  input:   num_of_trials --- number of trials
#  output:  the approximate probability
#
two_pts:=proc(num_of_trials::integer)
   local count, i, x1, y1, x2, y2, distance;

   if num_of_trials > 0 then
      count:=0;
      for i to num_of_trials do
         #
         # simulate the experiment
         #

         x1:=real_ran(0,1);  # generate the first point
         y1:=real_ran(0,1);

         x2:=real_ran(0,1);  # generate the second point
         y2:=real_ran(0,1);

         #
         # checking
         #
         distance:=evalf(sqrt( (x2-x1)^2+(y2-y1)^2 ));
         if distance < 1.0 then
            count:=count+1;
         fi;
      od;
      evalf(count/num_of_trials);
   else
      print(`invalid input`);
   fi;
end;
#
# function that output a random real number in [a,b]
#
real_ran:=proc(
               a::numeric,
               b::numeric
              )
   local x;
   x:=rand();
   evalf( a+ (b-a)*x/999999999999 );
end;
```

```
> read(`a:twopoint.txt`):
> two_pts(500);
```

.9800000000

```
>  two_pts(2000);
```

.9670000000

Combined estimate:

```
>  (0.98*500+0.967*2000)/2500;
```

.9696000000

We can make an estimate of 0.97. Further testing

```
>  two_pts(10000);
```

.9769000000

To get even better approximation, the number of trials many should be in millions.

4.3 Card games

4.3.1 Program that draws a random k-card hand

The following is the program that draws a random k-card hand. It actually consists of three programs. One program is considered the main program that does the following process:

< 1 > Get k distinct random numbers from 1 to 52 by calling "int_ran" as subprogram

< 2 > For each number obtained, translate it into the corresponding card name by calling "card" as subprogram.

```
#
################## main program ########################################
#
# program that drawing k cards, without replacement
# out a standard deck
#
#  input:  k    --- the number of balls to be drawn
#  output: hand --- array that contains the names of cards drawn
#
draw_k_card:=proc(
                  k::integer, # number of cards to draw
               hand::evaln    # output: the drawn hand
                  )
   local deck, i, m, n, temp;
```

```
   if k < 1 or k > 52 then   # quick exit if k < 1 or k > 52
      print(`1st argument must be between 1 and 52`);
      RETURN();
   fi;

   hand:=array(1..k);       # open the space for output
   deck:=array(1..52,       # making a pool of numbers to draw
         [seq(i,i=1..52)]);
   n:=52;                   # the number of cards remaining

   for i from 1 to k do     # loop to draw card 1 by 1

      m:=int_ran(1,n);      # draw a number from 1 to n
      hand[i]:=deck[m];

      if m<>n then          # move the number to the end
         temp:=deck[m];     # of the pool to avoid redrawn
         deck[m]:=deck[n];
         deck[n]:=temp;
      fi;

      n:=n-1;               # update n

   od;

   #
   # get the names of the cards
   #
   for i from 1 to k do
      m:=hand[i];         # get the number
      hand[i]:=card(m);   # call subprogram that identifies the card
   od;

end;
################# end of the main program ############################

###################### subprogram ###################################
#
#
# function that generates a random integer between m and n, inclusive
#
int_ran:=proc(m::integer,n::integer)
   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))
end;
############# end of subprogram int_ran ##############################

###################### subprogram ###################################
#
# function that identify the card name given its number
# input: m --- a number from 1 to 52
# output: an array of two entries that identifies the card
#
card:=proc(m::integer)
   local rank, suit;
   if m>0 and m<= 52 then  # quick exit if m is beyond 1-52
```

```
      rank:= m mod 13;     # get numerical rank (1-13)
      suit:=ceil(m/13);    # get numerical suit (1-4)
      #
      # identifies the actual suit as a string
      #
      if suit=1 then
         suit:="heart";
      elif suit=2 then
         suit:="spade";
      elif suit=3 then
         suit:="diamond";
      else
         suit:="club";
      fi;
      #
      # identifies the actual rank
      #
      if rank=1 then
         rank:="ace";
      elif rank=11 then
         rank:="jack"
      elif rank=12 then
         rank:="queen"
      elif rank=0 then
         rank:="king"
      fi;                # remaining cases are rank=2,3,4,...,10,
                         # already defined
      #
      # output the card name as an array
      #
      array(1..2,[rank,suit])
   fi;
end;
############# end of subprogram card ################################
```

4.3.2 Simulation of card games

We can use Maple programs to simulate many probability experiment. Those programs share a similar structure:

count:=0; # initialize the counter

for i from 1 to n do

block 1: simulation
simulate the experiment

block 2: checking
count 1 if the experiment was successful

od;
probability:=evalf(count/n); # calculate the probability

We usually need "int_ran", "real_ran" and/or other subprograms.

4.3.3 Example: Drawing a k-card hand having at least one ace

The following is the program

```
#
# program that compute the approximate probability of
# drawing a k-card hand having at least one ace
#
# input:  k --- number of cards in a hand
#         n --- number of trials for the experiment
#
# This program require draw_k_card package as subprogram
#
k_card_get_ace:=proc(k::integer,n::integer)
   local count, i, j, hand, flag;

   count:=0;   # initialize counting
   for i from 1 to n do
      #
      # simulate the experiment
      #
      draw_k_card(k,hand);  # draw a k card hand
      #
      # checking
      #
      flag:=0;              # initialize the flag=0, meaning no ace
      for j from 1 to k while flag=0 do
         if hand[j][1] = "ace" then
            flag:=1;        # raise the flag, an ace is in the hand
         fi;
      od;
      count:=count+flag;    # count
   od;

   evalf(count/n);          # calculate the probability
end;

#
################# subprogram ################################
#
# program that drawing k cards, without replacement
# out a standard deck
#
```

```
#  input:  k    --- the number of balls to be drawn
#  output: hand --- array that contains the names of cards drawn
#
draw_k_card:=proc(
                  k::integer, # number of cards to draw
               hand::evaln    # output: the drawn hand
                  )
   local deck, i, m, n, temp;

   if k < 1 or k > 52 then   # quick exit if k < 1 or k > 52
      print(`1st argument must be between 1 and 52`);
      RETURN();
   fi;

   hand:=array(1..k);       # open the space for output
   deck:=array(1..52,       # making a pool of numbers to draw
         [seq(i,i=1..52)]);
   n:=52;                   # the number of cards remaining

   for i from 1 to k do     # loop to draw card 1 by 1

      m:=int_ran(1,n);      # draw a number from 1 to n
      hand[i]:=deck[m];

      if m<>n then          # move the number to the end
         temp:=deck[m];     # of the pool to avoid redrawn
         deck[m]:=deck[n];
         deck[n]:=temp;
      fi;

      n:=n-1;               # update n

   od;

   #
   # get the names of the cards
   #
   for i from 1 to k do
      m:=hand[i];         # get the number
      hand[i]:=card(m);   # call subprogram that identifies the card
   od;

end;
################# end of the main program #############################

##################### subprogram ####################################
#
#
# function that generates a random integer between m and n, inclusive
#
int_ran:=proc(m::integer,n::integer)
   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))
end;
############# end of subprogram int_ran ###############################


##################### subprogram ####################################
```

```
#
# function that identify the card name given its number
# input: m --- a number from 1 to 52
# output: an array of two entries that identifies the card
#
card:=proc(m::integer)
   local rank, suit;
   if m>0 and m<= 52 then  # quick exit if m is beyond 1-52

      rank:= m mod 13;     # get numerical rank (1-13)
      suit:=ceil(m/13);    # get numerical suit (1-4)
      #
      # identifies the actual suit as a string
      #
      if suit=1 then
         suit:="heart";
      elif suit=2 then
         suit:="spade";
      elif suit=3 then
         suit:="diamond";
      else
         suit:="club";
      fi;
      #
      # identifies the actual rank
      #
      if rank=1 then
         rank:="ace";
      elif rank=11 then
         rank:="jack"
      elif rank=12 then
         rank:="queen"
      elif rank=0 then
         rank:="king"
      fi;              # remaining cases are rank=2,3,4,...,10,
                       # already defined
      #
      # output the card name as an array
      #
      array(1..2,[rank,suit])
   fi;
end;
############# end of subprogram card ################################
```

```
> read(`a:k-ace.txt`):
```

Let's see what's the probability of drawing a 4-card hand having at least one ace by repeat the trial 400 times.

```
> k_card_get_ace(4,400);
```

$$.3125000000$$

The exact probability is $\frac{15225}{52129} = .2812632745$

Find out the probability of drawing a 5-card hand having at least one ace through 200 trials.

```
> k_card_get_ace(5,200);
```

$$.3100000000$$

The exact probability is $1 - \frac{\frac{48!}{5!\,43!}}{\frac{52!}{5!\,47!}} = .3411580017$

```
> k_card_get_ace(5,500);
```

$$.3480000000$$

A little better approximation. If we combine the 700 trials, the total number of successful trials is

0.31*200+0.348*500=236, so the probability is $\frac{236}{700} = .3371428572$, very good approximation.

```
> k_card_get_ace(5,1000);
```

$$.3460000000$$

4.3.4 Example: Drawing a k-card hand having exactly one ace

The main program is shown below with subprograms omitted. The program file "kcard1ac.txt" should include the package "draw_k_card"

```
#
# program that compute the approximate probability of
# drawing a k-card hand having exactly one ace
#
# input:  k --- number of cards in a hand
#         n --- number of trials for the experiment
#
# This program require draw_k_card package as subprogram
#
k_card_get_1_ace:=proc(k::integer,n::integer)
   local count, i, j, hand, num_of_ace;

   count:=0;   # initialize counting
   for i from 1 to n do
      #
      # simulate the experiment
      #
      draw_k_card(k,hand);  # draw a k card hand
      #
      # checking
      #
      num_of_ace:=0;        # initialize the flag=0, meaning no ace
      for j from 1 to k do
         if hand[j][1] = "ace" then
            num_of_ace:=num_of_ace+1;   # count an ace
         fi;
```

```
      od;
      if num_of_ace=1 then
         count:=count+1;    # count a successful trial
      fi;
   od;

   evalf(count/n);          # calculate the probability
end;
```

(... draw_k_card package omitted in printing)

```
> read(`a:kcard1ac.txt`):
```

Let's investigate 4-card hands, try 300 and 1000 times.

```
> k_card_get_1_ace(4,300);
```

$$.2766666667$$

```
> k_card_get_1_ace(4,1000);
```

$$.2480000000$$

The exact probability is $\frac{69184}{270725} = .2555$. If we combine the 1300 trials:

```
> (0.2766666667*300+0.248*1000)/1300;
```

$$.2546153846$$

Much better.

4.4 More on probability simulations

4.4.1 How accurate is HIV test?

Suppose an HIV test is 99% accurate. The meaning of the accuracy is understood as

- if a person carries HIV virus, he/she has a 0.99 probability of being tested positive and 0.01 probability of being tested negative;
- if a person is does not carry HIV virus he/she has a 0.99 probability of being tested negative and 0.01 probability of being tested positive.

It is estimated that 0.5% of the population carry HIV virus. If there is someone is tested HIV positive, what is the probability that he/she actually carry HIV virus?

You ccan console this person with the fact that he/she has only about $\frac{1}{3}$ of the chance to be an actual HIV carrier. And you can use a Maple program to prove it to him/her.

The program is to simulate the situation that someone in a certain population is tested positive, and find out if this person is an HIV carrier.

To simulate the situation that someone is tested positive:

1. generate a population that 0.5% of them are HIV carrier and the rest are healthy
2. pick persons one by one to take the test:
   (a) if the person is a carrier, 99% chance positive, 1% chance negative
   (b) if the person is not a carrier, 99% chance negative, 1% chance negative until an HIV positive is found

```
#
#  In a population, 1 out of 200 people carries HIV virus.
#  Suppose that an HIV test is 99\% accurate, meaning for a HIV carrier,
#  he has the probability of 0.99 of testing positive, while a healthy
#  person has 0.99 chance of being tested negative. Estimate the
#  probability of the person who is tested  positive is actually a HIV
#  carrier.
#
#       Input: n --- the number of times to repeat the experiment
#       Output: --- the approximate probability
#
true_hiv:=proc(n)
   local count, i, j, k, u, s, result;

   count:=0;                # initialize the counting of success
   #
   # construct the population with 1/200 sick
   #
   u:=array(1..200,["sick",seq("healthy",i=2..200)]);

   for i to n do            # loop for repeating the experiment

      result:="negative";   # loop of finding a person tested positive
      while result="negative" do
         #
         # pick a person out out the population
         #
         k:=int_ran(1,200);
         #
         # perform the test, which is simulated as a lottery
         #
         if u[k]="sick" then
            s:=int_ran(1,100);      # sick person's lottery:
            if s = 1 then
```

```
            result:="negative"   # 1 out 100 chance of being negative
         else
            result:="positive"
         fi;
      else
         s:=int_ran(1,100);      # healthy person's lottery:
         if s = 1 then
            result:="positive"   # 1 out 100 chance of being positive
         else
            result:="negative"
         fi;
      fi;
   od;
   #
   # check if the positive testee is actually sick
   #
   if u[k]="sick" then
      count:=count+1
   fi;

od;

evalf(count/n);

end;
#
#
# function that generates a random integer between m and n, inclusive
#
int_ran:=proc(m::integer,n::integer)
   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))
end;
```

```
> read(`a:true_hiv.txt`):
> true_hiv(500);
```

When someone is tested positive, the probability that he carries HIV is

$$.2740000000$$

The exact probability is $\frac{99}{298} = .33221476$

4.4.2 Are you lonesome?

In a group of k people, every one is to pick two others as friends. If a person was not picked by anyone, this person is lonesome. Find the probability that someone is lonesome.

```
# In a group of k people, every one wants to make friend with two
# other people in the group. What is the probability that someone
# is lonesome, meaning he/she is not picked by anyone
```

```
#
#  input:   k  --- the number of people in the group
#           n  --- the number of trials of the experiment
#
lonesome:=proc(  k::integer,
                 n::integer
               )
   local  u, i, j, r, s, flag, count;

   u:=array(1..k);      # open the space for status of each person

   count:=0;            # initialize counting

   for i to n do        # repeat the socializing experiment n times

      for j to k do  # at beginning, everyone is lonesome
         u[k]:="lonesome"
      od;

      for j to k do     # loop for each one to make friends
         #
         # pick the first friend: the person r
         #
         r:=j;
         while r=j do          # no one make friend with himself
            r:=int_ran(1,k);   # person j pick a friend
         od;
         u[r]:="happy";        # person r is picked so no longer lonesome
         #
         # pick the second friend: the person s}
         #
         s:=j;
         while s=j or s=r do   # keep picking until getting a new person
            s:=int_ran(1,k)
         od;
         u[s]:="happy";        # now the person s is no longer lonesome
      od;
      #
      # check if anyone is lonesome after
      #
      flag:=0;
      for j to k while flag=0 do
         if u[j]="lonesome" then
            flag:=1
         fi;
      od;

      if flag=1 then
         count:=count+1
      fi;

   od;

   evalf(count/n);

end;
#
```

```
#
# function that generates a random integer between m and n, inclusive
#
int_ran:=proc(m::integer,n::integer)
   round(evalf(m-0.5+(n-m+1)*rand()/999999999999))
end;
```

```
> read(`a:/txt/lonesome.txt`):
```

Let try a 5 person group via 200 trials

```
> lonesome(5,200);
```

$$.06000000000$$

Quite low probability

```
> lonesome(30,500);
```

$$.1180000000$$

In a 30-person group, it seems the probability that someone is lonesome is a little higher.

4.4.3 The Monty Hall dillema

————————————————————————————————-

To Switch or Not to Switch
by Donald Granberg, University of Missouri

In the September 9, 1990, issue of *Parade*, Craig F. Whitaker of Columbia Maryland, poses this query to Marilyn vos Savant (in the "Ask Marilyn" column):

Suppose you are on a game show and you are given the choice of three doors. Behind one door is a car; behind others, goats. You pick a door, say number 1, and the host, who knows what's behind the doors, opens another door, say number 3, which has a goat. He then says to you, "Do you want to switch to door number 2?" Is it to your advantage to switch your choice?

Marilyn's answer was direct and unambiguous,

Yes, you should switch. The first door has a one-third chance of winning, but the second door has a two-thirds chance. Here is a good way to visualize what happend. Suppose there are a million doors, and you pick door number 1. The the host, who knows what's behind the doors and will always avoid the one with the prize, opens them all except door number 777,777. You'd switch to that door pretty fast, wouldn't you?

Despite her explanation, she received a large volume of mail, much of which was from irate and incredulous readers who took issue with her answer. ... Letters came from great variety of people and places, from people with lofty titles and affiliations and from others of more humble circumstances. For example, Andrew Bremner of the Department of Pure Mathematics at Cambridge University in England, wrote with a touch of noblesse oblige.

Dear Marilyn,

... your answer that you should switch to door number 2 ... is incorrect. Each of doors number 1 and number 2 has 1/2 chance of winning. ... Your correspondents seem rather rude; I wager your womanhood is a factor!

Yours sincerely,

Andrew Bremne

> ... no other statistical puzzle comes so close to fooling all all the people all the time.... The phenomenon is paticularly interesting precisely because of its specificity, its reproducibility, and its immunity to higher education.... Think about it. Ask your brightest friends. Do not tell them though (or at least not yet), that even Nobel physicists systematically give the wrong answer, and that they insist on it, and are ready to berate in print those who propose the right answer.... By Massimo Piattelli-Palmarini in Bostonia (Jul/Aug, 91)

Project: Write a maple program to simulate the game and to approximate the probability of winning by switching. Your simulation should include the following components:

1. randomly assign a car and two goats to door[1], door[2], and door[3].
2. randomly pick a door number k among 1, 2, 3.
3. the host reveals a losing door number m from the two doors other than the door you chose.
4. switch to the remaining door and check if it is the winning door.

Play the game n times and calculate the approximate probability of winning by switching.

4.5 Exercises

1. **Estimating π** A circle of radius 1 is drawn inside a 2×2 square. If one throws an object into the square, the probability of landing inside the circle is $\frac{\pi}{4}$. If one repeats this experiment n times, he can estimate this probability and the estimated probability times 4 should be close to π. This is an easy (but inefficient) algorithm of estimating π. Write a program, using this algorithm, to estimate π for input n, which is the number of trials of the experiment.

2. **Quality control** When shipping diesel engines abroad, it is common to pack 12 engines in one container that is then loaded on a rail car and sent to a port. Suppose that a company has received complaints from its customers that many of the engines arrive in nonworking condition. The company thereby makes a spot check of containers after loading. The company will test three engines from a container randomly. If any of the three are nonworking, the container will not be shipped. Suppose a given container has 2 nonworking engines. Use a Maple program to estimate the probability that the container will not be shipped.

3. **A k-card hand flush**

   Write a program to estimate the probability of drawing a k-card hand having the same suit (called a *flush*. The exact probability is $4\,\frac{13!\,(52-k)!}{(13-k)!\,52!}$ The exact probabilities for $k=2$ and $k=3$ are $\frac{4}{17}$ and $\frac{22}{425}$ respectively

4. **Problem 4: Drawing balls**

   A box contains 4 white balls and 8 red balls. Write a program that estimates the probability of drawing k balls without replacement and getting all red.

5. **How lucky are you if you are tested HIV negative?**

   Use the HIV example above, find the probability that, if a person is tested HIV negative, the person is indeed healthy.

6. **Are you lonesome again?**

   There are a 15 boys and 10 girls. Every boy will date a girl. However, there might be several boys who want to date a same girl. Find the probability that at least one girl is lonesome.

7. **Friend match**

   There are k people in a group and you are one of them, say the person #1. Every one is to make friends with two other people. If you pick a friend, who also pick you as a friend, then you two matches. Find the probability that you can find your match.

8. **Two boys** In an issue of *Parade*, a question was debated in the column *Ask Marilyn*: **If a family ahs exactly two kids and at least one of them is a boy, what is the probability that both kids are boys?** Marilyn vos Savant, the columnist, gave the answer 1/3 but many readers didn't agree. One of the readers thought the answer should have been 1/2 and challenged Marilyn with $ 1000 bet. Write a program to simulate the experiment and settle the bet.

   The simulation part of the program should consist of the following components:

   (a) generate two "kids", in an array with entries either "boy" or "girl", each kid has a 50/50 chance of being a boy or a girl.

   (b) repeat step (a) until at least one of the two kids is a boy.

   To assign a gender to each kid, you can use the random integer generator from 1 to 2 and designate them as boy and girl respectively.