Eric Kim and Zhaoliang (Leon) Zhou

2022-08-29

```r
library(numbers)
library(Zseq)
library(ggplot2)
library(dplyr)
set.seed(2022)
```

```r
# a function that finds the divisors/factors given an integer X
# function returns divisors of X including,
# but except the number X itself
div_func <- function(X){
  div_vec = vector(mode="numeric")
  for (i in 1:X){
    if ((X %% i)==0){
      div_vec = append(div_vec, i)
    }
  }
  div_vec = div_vec[-length(div_vec)]
  return(div_vec)
}
# test
div_func(100)
```

```
## [1]  1  2  4  5 10 20 25 50
```

```r
# write a function that get the digits for a given number X
get_digits <- function(X)
   {
      as.integer(unlist(strsplit(as.character(X), "")))
}
# test
get_digits(1452)
```

```
## [1] 1 4 5 2
```

```r
# write a function that finds all the prime less than a given number N
find_prime <- function(N){
  prime_vec <- numeric()
  for (i in 1:N){
    divs = div_func(i)
    if (length(divs) == 1){
      prime_vec = append(prime_vec,i)
    }
  }
return(prime_vec)
}
```

```
find_prime(100)
```

```
##  [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

```
#write a function that test a given number is prime or not
is_prime = function(N){
  divs = div_func(N)
  if (length(divs) ==1)
    return(TRUE)
  else
    return(FALSE)
}
is_prime(13)
```

```
## [1] TRUE
```

```
# write a function that reverse a number
Reverse_number <- function(x){

  n <- trunc(log10(x)) # now many powers of 10 are we dealing with
  x.rem <- x # the remaining numbers to be reversed
  x.out <- 0 # stores the output
  for(i in n:0){
  x.out <- x.out + (x.rem %/% 10^i)*10^(n-i) # multiply and add
  x.rem <- x.rem - (x.rem %/% 10^i)*10^i # multiply and subtract

  }
return(x.out)
}
Reverse_number(45)
```

```
## [1] 54
```

1.

Find the golden number, or the perfect number

```
perf_numbers = vector(mode="numeric") # create an empty vector to store values
N = 1e4
for (i in 1:N){
  # find all the divisors except for the number itself
  Div = div_func(i)
  # take the sum of divisors
  Sum = sum(Div)
  # if the sum is the number itself,
  # then store into the vector
  if (Sum == i){
    perf_numbers <- append(perf_numbers, i)
  }
}

perf_numbers
```

```
## [1]    6   28  496 8128
```

From the results above, we can see all the golden numbers up to 10,000 are: 6, 28, 496, and 8128

2.

Find the happy numbers up to a specific integer N

```r
# First write a function that test if a given number X is a happy number
is.happy <- function(X)
{
   stopifnot(is.numeric(X) && length(n)==1)

   digits <- get_digits(X)
   previous <- c()
   repeat
   {
      sumsq <- sum(digits^2, na.rm=TRUE)
      if(sumsq==1L)
      {
         happy <- TRUE
         break
      } else if(sumsq %in% previous)
      {
         happy <- FALSE
         attr(happy, "cycle") <- previous
         break
      } else
      {
         previous <- c(previous, sumsq)
         digits <- get_digits(sumsq)
      }
   }
   happy
}
```

```r
# test
is.happy(4)
```

```
## [1] FALSE
## attr(,"cycle")
## [1]  16  37  58  89 145  42  20   4
```

Next, write a function that find all the happy number up to a specific number N

```r
find_happy <- function(N){

  happy.list = c(which(apply(rbind(1:N), 2, is.happy)) )
  return(happy.list)
}
# find all the happy numbers up to 500
find_happy(500)
```

```
##  [1]   1   7  10  13  19  23  28  31  32  44  49  68  70  79  82  86  91  94  97
## [20] 100 103 109 129 130 133 139 167 176 188 190 192 193 203 208 219 226 230 236
## [39] 239 262 263 280 291 293 301 302 310 313 319 320 326 329 331 338 356 362 365
## [58] 367 368 376 379 383 386 391 392 397 404 409 440 446 464 469 478 487 490 496
```

Above are all the happy numbers up to a given range 500.

3.

Approximate sqrt(N). The method described in the homework sheet is actually the Newton's method for approximation square root.

```
sqrt_apprx = function(N){
  G = 2  # initial guess
  tol = 1e-4 # stop threshold, we want to stop when achieve 0.1% accuracy
  i = 0 # keep track of number of steps needed
  repeat{
  # next guess is 0.5(FG + N/FG)
    G = 0.5 * (N / G + G)
  # stop if accuracy becomes smaller than threshold
    i=i+1
    if (abs(G * G - N) < tol) break
  }
return(c(G,i))
}
```

```
# test function
sqrt_apprx(90)
```

```
## [1] 9.486833 6.000000
```

Above is the approximation for sqrt(90). From the output above, we see that it took 6 iterations to achieve the threshold.

Then, we compare the result using sqrt() function in R

```
c(sqrt_apprx(3), sqrt(3))
```

```
## [1] 1.732051 3.000000 1.732051
```

```
c(sqrt_apprx(10), sqrt(10))
```

```
## [1] 3.162278 4.000000 3.162278
```

The 3 and 4 from above are the number of steps it took to estimated the square root. From above, for $\sqrt{3}$, we approximated it to be 1.732051 which is the same as using the R build-in function. The result is the same for $\sqrt{10}$. From the code above, we could see our function successfully approximate the sqrt(N), and the results are the same as using R build in function

4.

From the website https://en.wikipedia.org/wiki/Pythagorean_triple, we can learn that we could use the Euclide's formula to generate pythagorean triples: a = m^2 - n^2 b = 2mn c = m^2 + n^2

where m and n satisfies the condition: GCD(m,n)=1 and m-n being odd, and m>n>1

Next, we write a function that generate Pythagorean triples x,y,z where $x^2 + y^2 = z^2$ and z<=100:

```
zl=c()
for (z in 1:100){
  for (x in 1:z){
    y=sqrt(z^2-x^2)
    if(y%%1==0 && y !=0 && y>=x){
      zl=c(zl,paste0(x,"-",y,"-",z))
    }
  }
}
print(zl)
```

```
##  [1] "3-4-5"     "6-8-10"    "5-12-13"   "9-12-15"   "8-15-17"   "12-16-20"
##  [7] "7-24-25"   "15-20-25"  "10-24-26"  "20-21-29"  "18-24-30"  "16-30-34"
## [13] "21-28-35"  "12-35-37"  "15-36-39"  "24-32-40"  "9-40-41"   "27-36-45"
## [19] "14-48-50"  "30-40-50"  "24-45-51"  "20-48-52"  "28-45-53"  "33-44-55"
## [25] "40-42-58"  "36-48-60"  "11-60-61"  "16-63-65"  "25-60-65"  "33-56-65"
## [31] "39-52-65"  "32-60-68"  "42-56-70"  "48-55-73"  "24-70-74"  "21-72-75"
## [37] "45-60-75"  "30-72-78"  "48-64-80"  "18-80-82"  "13-84-85"  "36-77-85"
## [43] "40-75-85"  "51-68-85"  "60-63-87"  "39-80-89"  "54-72-90"  "35-84-91"
## [49] "57-76-95"  "65-72-97"  "28-96-100" "60-80-100"
```

Above are all the Pythagorean triplets in the order of x,y,z with z<=100.

5.

```
# start with an example
num <- 5126
num_list <- get_digits(num)
max_num = as.character( sort(num_list, decreasing = T) )
max_num = as.numeric(paste(max_num, collapse=""))

min_num = as.character( sort(num_list, decreasing = F) )
min_num = as.numeric(paste(min_num, collapse=""))

max_num
```

```
## [1] 6521
```

```
min_num
```

```
## [1] 1256
```

```
diff <- max_num - min_num
diff
```

```
## [1] 5265
```

```
# build a function called find_magic
# to find the magic number
# argument: any interger
# return: the magic number, the number of iterations to find the magic number
find_magic <- function(num){
  i=0
  repeat{
    i=i+1
    previous = num
    # get the digits
    num_list <- get_digits(num)
    # reorder the digits to get the max value
    max_num = as.character( sort(num_list, decreasing = T) )
    max_num = as.numeric(paste(max_num, collapse=""))

    #reorder the digits to form the smallest number
    min_num = as.character( sort(num_list, decreasing = F) )
    min_num = as.numeric(paste(min_num, collapse=""))

    # calculate the difference max-min
    diff <- max_num - min_num

    # check for converge
    # if the current number is the same as the previous number - converged, and stop
    # OW, continue to update
    if (diff==previous){break}
    num = diff
    print(diff)
}
return(c(diff,i))
}
```

```
# test the function
find_magic(2986)
```

```
## [1] 7173
## [1] 6354
## [1] 3087
## [1] 8352
## [1] 6174

## [1] 6174    6
```

From the result above, we can see that the magic number is 6174, which is refereed to as the Kaprekar's constant.

6.

Find all the pairs of twin primes that are less than 100 we can write a function to do this

```
find_twin_primes <- function(N){
  # first find all the prime number <=N
  prime_vec <- find_prime(N)

  # record the locations where difference is 2
  pairs_loc1 <- which(diff(prime_vec)==2)
  # order location from small to large
  pairs_loc2 <- sort(c(pairs_loc1, pairs_loc1+1))
  pairs_loc2
  twin_mat <- matrix(prime_vec[pairs_loc2], ncol=2, byrow=TRUE)
  return(twin_mat)
}
```

```
# test the function
find_twin_primes(100)
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    5    7
## [3,]   11   13
## [4,]   17   19
## [5,]   29   31
## [6,]   41   43
## [7,]   59   61
## [8,]   71   73
```

Therefore, from the above code, we can find all the twin primes less than a give number N. All the twin primes that are less than 100 are: (3,5), (5,7), (11,13), (17,19), (29,31), (41, 43), (59, 61), (71,73) as printed above.

7.

Find all palindromic primes up to 500

First, write a function to test if a number is palindromic or not

```
check_palindromic <- function(N){
  N = as.integer(N)
  revs = 0
  num = N
  while(N>0){
    r = N%%10
    revs = revs*10+r
    N = N%/%10
}
  if (revs == num)
    return(TRUE)
  else{return(FALSE)}
}
```

```
check_palindromic(100)
```

```
## [1] FALSE
```

Next, write a function that find all the palindromic primes up to N

```
find_palin_primes <- function(Num){
  prime_vec <- find_prime(Num)
  for (i in 1:length(prime_vec)){
    a_prime = prime_vec[i]
    if (check_palindromic(a_prime) == TRUE){
      print(a_prime)
  }
}
}
```

```
find_palin_primes(500)
```

```
## [1] 2
## [1] 3
## [1] 5
## [1] 7
## [1] 11
## [1] 101
## [1] 131
## [1] 151
## [1] 181
## [1] 191
## [1] 313
## [1] 353
## [1] 373
## [1] 383
```

Above are all the palindromic primes up to 500.

8.

Find the smallest 6-element cunningham chain. If p is a prime, the the 6 element: p 2*p+1 4*p+3 8*p+7 16*p+15 32*p+31 are all primes

```r
prime_vec <- find_prime(100)
for (i in 1:length(prime_vec)){
  p = prime_vec[i]
  if ( is_prime(2*p+1) == TRUE &
       is_prime(4*p+3) == TRUE &
       is_prime(8*p+7) == TRUE &
       is_prime(16*p+15) == TRUE &
       is_prime(32*p+31) == TRUE){
    print(c(p, 2*p+1, 4*p+3, 8*p+7, 16*p+15, 32*p+31) )
  }

}
```

```
## [1]   89  179  359  719 1439 2879
```

From the result above, the smallest 6-element cunningham chain are: (89,179,359,719,1439,2879)

9.

Find all cyclic primes up to 1000

First, write a function that finds the cyclic permutation of the digits of a number

```r
find_cyclic <- function(N){
  num = N
  vec = numeric()
  n = length(get_digits(N))
      while (1){
        #print(num)
        vec = append(vec, num)
        # Following three lines
        # generates a circular
        # permutation of a number.
        rem = num %% 10
        div = floor(num / 10)
        num = ( (10^(n-1)) *rem + div)
        # If all the permutations
        # are checked and we obtain
        # original number exit from loop.
        if (num == N)
            break
      }
  return(vec)
}
```

```
#test
find_cyclic(123)
```

```
## [1] 123 312 231
```

Next, write a function that test if a number and all its cyclic permutation of digits are prime

```
if_cyclic_prime = function(X){
  a.c <- find_cyclic(X)
  vec1 <- numeric()
  vec2 <- numeric()
  for (i in 1:length(a.c)){
    if(is_prime(a.c[i]) ==TRUE)
      vec1 = append(vec1, "prime")
    else
      vec2 = append(vec2, "noprime")
}

  if(length(vec1)==length(a.c)) return(TRUE)
  else return(FALSE)
}
# test the function
if_cyclic_prime(132)
```

```
## [1] FALSE
```

Then, write a function that finds all the cyclic prime up to a given number

```
find_cyclic_prime = function(Num){
  vec = numeric()
  prime_vec <- find_prime(Num)
  for (i in 1:length(prime_vec)){
    a_prime = prime_vec[i]
  if (if_cyclic_prime(a_prime) == TRUE){
      vec = append(vec, a_prime)
    }

  }
  return(vec)
}
```

```
find_cyclic_prime(1000)
```

```
##  [1]   2   3   5   7  11  13  17  31  37  71  73  79  97 113 131 197 199 311 337
## [20] 373 719 733 919 971 991
```

Above are all the cyclic prime numbers up to 1000.

10.

First, write a function that reverse a given number

```
n=123
rev_number=function(n){
  if(n>0)
  return(as.integer(paste0(rev(unlist(strsplit(as.character(n), ""))), collapse = "")))
  else{
    return(-as.integer(paste0(rev(unlist(strsplit(as.character(abs(n)), ""))), collapse = "")))
  }
}
rev_number(n)
```

```
## [1] 321
```

Then, write a function that find all reversible primes up to N (N=1000)

```
find_reversible_prime = function(Num){
  vec = numeric()
  prime_vec <- find_prime(Num)
  for (i in 1:length(prime_vec)){
    a_prime = prime_vec[i]
    prime_rev = rev_number(a_prime)

  if (is_prime(prime_rev) == TRUE){
      vec = append(vec, a_prime)
    }
  }
  return(vec)
}
```

```
find_reversible_prime(1000)
```

```
##  [1]   2   3   5   7  11  13  17  31  37  71  73  79  97 101 107 113 131 149 151
## [20] 157 167 179 181 191 199 311 313 337 347 353 359 373 383 389 701 709 727 733
## [39] 739 743 751 757 761 769 787 797 907 919 929 937 941 953 967 971 983 991
```

Above are the list of reversible primes up to 1000.

11.

First case, at least one 6 in four die rolls.

Second case, at least one double 6 in 24 rolls of two dice

First case

For first case, we can caculate the probability as follows: Let X be the even a 6 shows up P(X >=1) = 1-P(X=0) = 1 - (4 choose 0)x(1/6)^0 x (1 - 1/6)^4 = 52%

Here, we perform a simulation study to estimate the probability:

```
set.seed(2022)
nthrows = 1e4
result_vec <- numeric()
for (i in 1:nthrows){
  die1 = sample(1:6, 1, replace = T, prob = rep(1/6,6))
  die2 = sample(1:6, 1, replace = T, prob = rep(1/6,6))
  die3 = sample(1:6, 1, replace = T, prob = rep(1/6,6))
  die4 = sample(1:6, 1, replace = T, prob = rep(1/6,6))
  if (die1 ==6 | die2 ==6 | die3 ==6 | die4 ==6){
    result_vec[i]=1
  }
  else {
    result_vec[i]=0
  }
}
mean(result_vec)
```

```
## [1] 0.5201
```

From above, we performed a simulation study with 10,000 replications to estimate the probability of at least one 6 in four die rolls. From the result, we can see the probability is about 5201% which is greater than 50%, and this is the reason he was winning money.

Second case

Probability at least one double 6 in 24 rolls of two dice. Mathematically, this probability is 1-(35/36)^24 = 49%. Here, we use simulation to estimate this probability.

We can first write a function that returns 1 if a double 6 occurs in 24 rolls of two dice

```
roll_dice_func <- function(){
  roll_vec <- numeric()
  for (i in 1:24){
    dice = sample(1:6, 2, replace = T, prob = rep(1/6,6))
    dice1 = dice[1]
    dice2 = dice[2]
    if (dice1 == 6 & dice2 == 6){
      roll_vec[i] =1
    }
    else {roll_vec[i] =0}
  }
  avg = mean(roll_vec)
  if (avg>0){return(1)}
  else {return(0)}
}
```

```
# test function
roll_dice_func()
```

```
## [1] 1
```

Next, we perform the simulation with 1e4 numbers of replication to estimate this probability:

```
set.seed(2022)
reps = 1e4
vec2 <- numeric()
for (i in 1:reps){
  vec2[i] = roll_dice_func()
}
mean(vec2)
```

```
## [1] 0.4921
```

From the result above, the estimated probability of at least one double 6 in 24 rolls of two dice is 49.21%. Since this probability is less than 50%, he is lossing money.

12.

Suppose that we toss a fair coin twenty times. What is the probability that we get a sequence of exactly four heads in a row at some point?

Write a function: input: number of tosses, length of sequence (ex. 4 heads in a row) output: probability

Here, we use simulation study to estimate the probability. We use reps times of replication, for each replication, we toss a fair coin N times and record if there is a consecutive K heads exactly.

```
coin_func <- function(N,k){
  reps = 1e5
  vec_heads = numeric(reps)
  for (i in 1:reps){
  coin = sample(c(0,1), size = N, replace = T,prob = c(0.5, 0.5))

  if(TRUE %in% c(rle(coin)$lengths[which(rle(coin)$value==1)]==k)){vec_heads[i]=1}
  else {vec_heads[i]=0}
  }
mean(vec_heads)
}
```

Next, we test with 20 tosses with 4 consecutive heads exactly

```
set.seed(2022)
coin_func(20,4)
```

```
## [1] 0.27137
```

From the result above, we see the probability 20 tosses with 4 consecutive heads exactly is around 0.27. Next, test with 50 tosses with 6 consecutive heads exactly:

```
coin_func(50,6)
```

```
## [1] 0.16944
```

Thus, the probability 50 tosses with 6 consecutive heads exactly is around 0.17. The above function can calculate the probability given any number of tosses and sequence of heads(or tails) give the coin is fair.

13

150 mailbox

```
mailbox1 = numeric(150)
mailbox_mat = data.frame(mailbox1)

for (i in 2:150) {
  vect = numeric(150)
  vect[seq(0, length(vect), i)]=1
  mailbox_mat[,i]= vect

}
```

```
mailbox_mat[,151]= numeric(150)
for (j in 1:150){
  count_open = sum(mailbox_mat[j,]==1)
  if(count_open %% 2 == 0){mailbox_mat[j,151] = "close"}
  else {mailbox_mat[j,151]= "open"}
}
```

```
mailbox_mat[,151]
```

```
##   [1] "close" "open"  "open"  "close" "open"  "open"  "open"  "open"  "close"
##  [10] "open"  "open"  "open"  "open"  "open"  "open"  "close" "open"  "open"
##  [19] "open"  "open"  "open"  "open"  "open"  "open"  "close" "open"  "open"
##  [28] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "close"
##  [37] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
##  [46] "open"  "open"  "open"  "close" "open"  "open"  "open"  "open"  "open"
##  [55] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
##  [64] "close" "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
##  [73] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "close"
##  [82] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
##  [91] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
## [100] "close" "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
## [109] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
## [118] "open"  "open"  "open"  "close" "open"  "open"  "open"  "open"  "open"
## [127] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"
## [136] "open"  "open"  "open"  "open"  "open"  "open"  "open"  "open"  "close"
## [145] "open"  "open"  "open"  "open"  "open"  "open"
```

From the result above, we see that open mailboxes are at positions 1, 4, 9, 16… which are perfect squares.

14.

Buffon needle problem. The idea for coding is from Wikipedia: https://en.wikipedia.org/wiki/Buffon%27s_needle_problem

```
# function has two parameters:
# l: needle length
# t line spacing
buffon <- function(l, t) {
# Sample the location of the needle's centre.
   #
   x <- runif(1, min = 0, max = t / 2)
   #
   # Sample angle of needle with respect to lines.
   #
   theta = runif(1, 0, pi / 2)
   #
   # Does the needle cross a line?
   #
   x <= l / 2 * sin(theta)
   # return T/F
}
buffon(1,2)
```

```
## [1] FALSE
```

Next, we use 1e4 number of replications to estimate the probability that the needle crossed the line. Then, we estimate the pi

```
set.seed(2022)
 l = 1
```

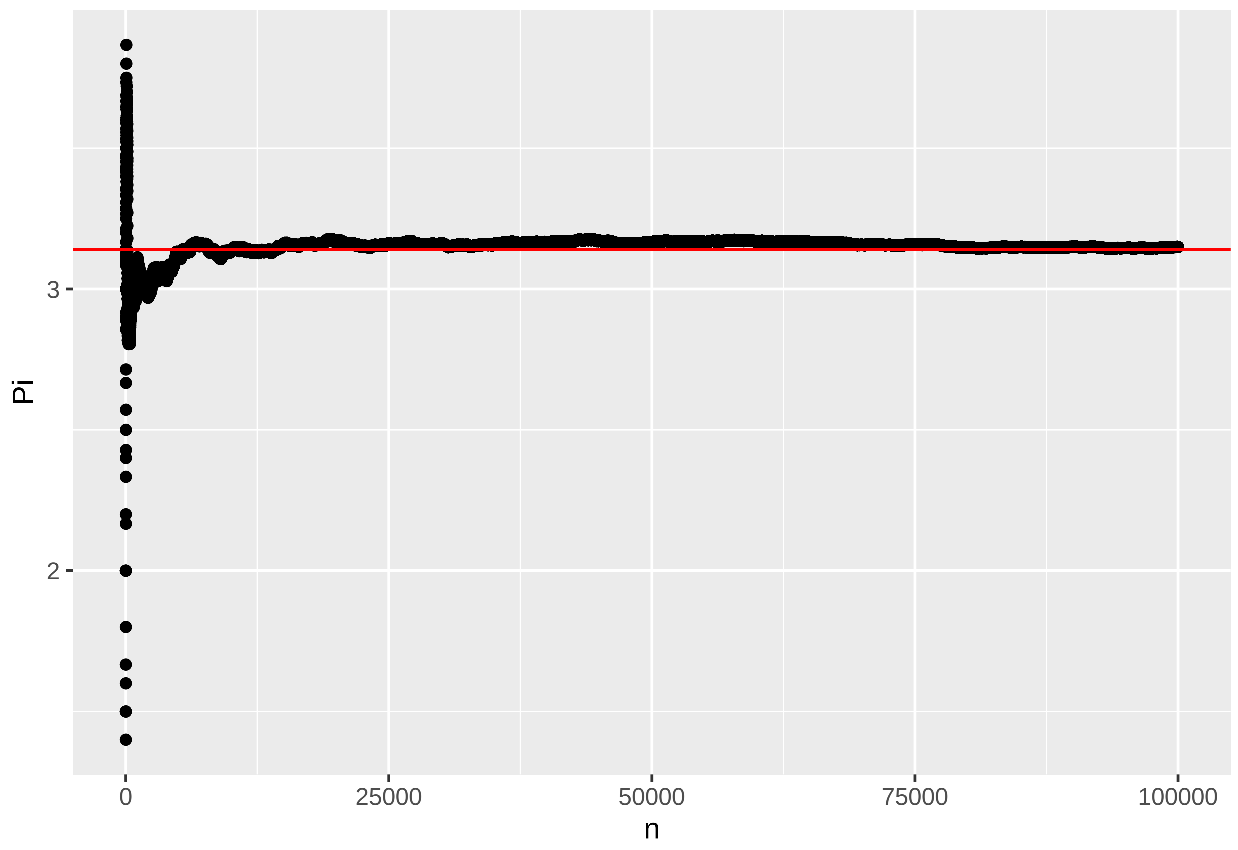
```
t = 2
#
N = 1e5
#
cross = replicate(N, buffon(l, t))
sum(cross==T)/N
```

```
## [1] 0.31753
```

```
library(dplyr)
#
estimates = data.frame(
  n = 1:N,
  Pi = 2 * l / t / cumsum(cross) * (1:N)
) %>% subset(is.finite(Pi))
```

Here, we have estimated probability around 3.175.

```
ggplot(estimates, aes(x = n, y = Pi)) +
  geom_point() +
  geom_hline(yintercept=3.14, color = "red")
```



From the plot above, we could see the estimates conveges to the value of Pi as n increases. The red line is the reference line for Pi, and the black dot line is our estimates.

15

Write a program that construct a magic square

```r
magic_square <- function(n) {
  if (n %% 2 == 1) {
    p = (n + 1) %/% 2 - 2
    ii = seq(n)
    outer(ii, ii, function(i, j) n * ((i + j + p) %% n) + (i + 2 * (j - 1)) %% n + 1)
  } else if (n %% 4 == 0) {
    p = n * (n + 1) + 1
    ii = seq(n)
    outer(ii, ii, function(i, j) ifelse((i %/% 2 - j %/% 2) %% 2 == 0, p - n * i - j, n * (i - 1) + j))
  } else {
    p = n %/% 2
    q = p * p
    k = (n - 2) %/% 4 + 1
    a = Recall(p)
    a = rbind(cbind(a, a + 2 * q), cbind(a + 3 * q, a + q))
    ii = seq(p)
    jj = c(seq(k - 1), seq(length.out=k - 2, to=n))
    m = a[ii, jj]; a[ii, jj] <- a[ii + p, jj]; a[ii + p, jj] <- m
    jj = c(1, k)
    m = a[k, jj]; a[k, jj] <- a[k + p, jj]; a[k + p, jj] <- m
    a
  }
}
```

```r
magic_square(3)
```

```
##      [,1] [,2] [,3]
## [1,]    8    1    6
## [2,]    3    5    7
## [3,]    4    9    2
```

```r
magic_square(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   17   24    1    8   15
## [2,]   23    5    7   14   16
## [3,]    4    6   13   20   22
## [4,]   10   12   19   21    3
## [5,]   11   18   25    2    9
```

```r
magic_square(7)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]   30   39   48    1   10   19   28
## [2,]   38   47    7    9   18   27   29
## [3,]   46    6    8   17   26   35   37
## [4,]    5   14   16   25   34   36   45
## [5,]   13   15   24   33   42   44    4
## [6,]   21   23   32   41   43    3   12
## [7,]   22   31   40   49    2   11   20
```

Using the steps provided, the function could find the magic squares for odd number of n. Couple examples are given above for n = 3, 5,7.

16.

Write a program that finds the GCD and LCM of two integers A and B.

First, find the GCD

```
gcd_func <- function(A,B) {
  r <- A%%B;
  return(ifelse(r, gcd_func(B, r), B))
}
```

```
# test GCD
gcd_func(12,14)
```

```
## [1] 2
```

Above is the function that finds the GCD of given integers 12 and 14

Next, write a function that finds the LCM

```
lcm_func <- function(A, B) {
# choose the greater number
if(A > B) {
greater = A
} else {
greater = B
}
while(TRUE) {
if((greater %% A == 0) && (greater %% B == 0)) {
lcm = greater
break
}
greater = greater + 1
}
return(lcm)
}
```

```
lcm_func(12,14)
```

```
## [1] 84
```

The LCM for 12 and 14 is 84

Next, we verify using the formula GCDxLCM = AxB

```
A = 12
B = 14
GCD1 = gcd_func(A,B)
LCM1 = lcm_func(A,B)
GCD1*LCM1 == A*B
```

```
## [1] TRUE
```

Since the result is true, we have verified that 2 is the GCD of 12 and 14, and 84 is the LCM for 12 and 14. We have verified our functions work.

17.

The birthday problem. find the probability that at least two people in the room have the same birthday, given n people in the room.

```
birthday_prob = function(n){
  p <- numeric(n)  # create numeric vector to store probabilities
  for (i in 1:n)      {
            q <- 1 - (0:(i - 1))/365  # 1 - prob(no birthday matches)
            p[i] <- 1 - prod(q)  } # take cumulative product
  prob <- p[n]

return(prob)
}
```

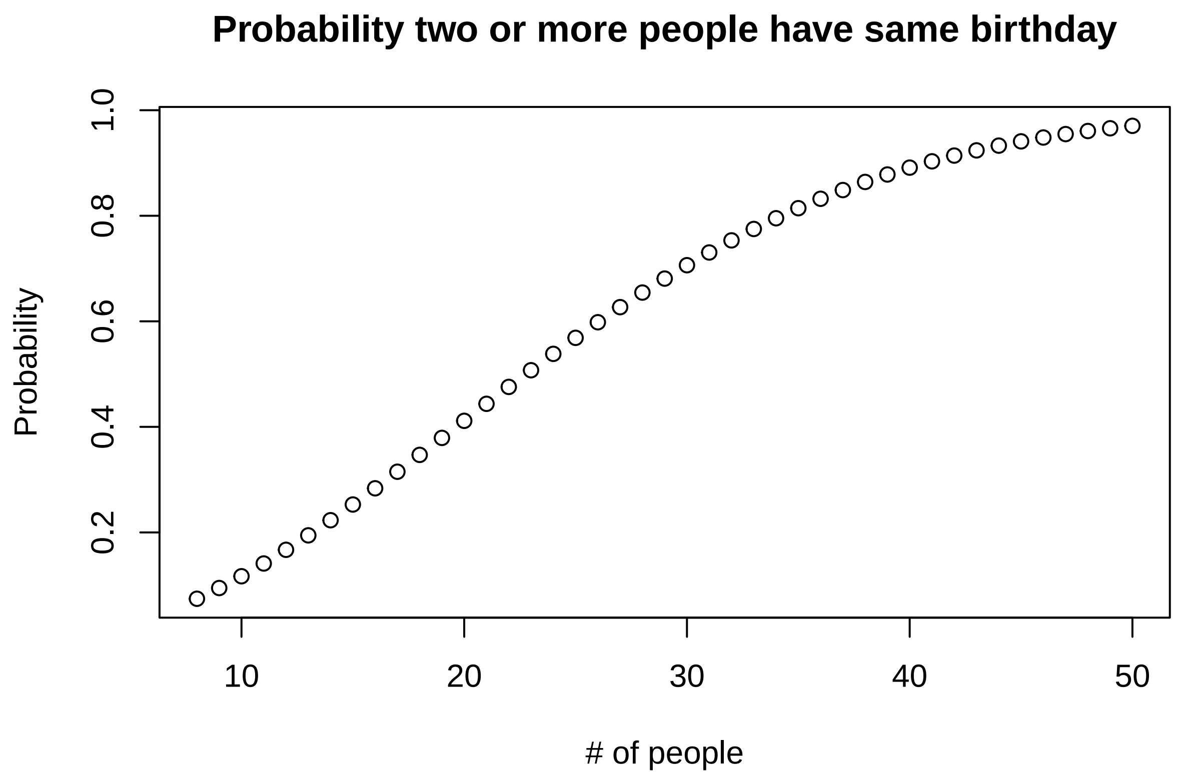
```
birthday_prob(23)
```

```
## [1] 0.5072972
```

```
birthday_prob_vecs <- numeric()
nrange  <- 8:50
for (i in nrange){
  birthday_prob_vecs = append(birthday_prob_vecs, birthday_prob(i))
}
birthday_prob_vecs
```

```
##  [1] 0.07433529 0.09462383 0.11694818 0.14114138 0.16702479 0.19441028
##  [7] 0.22310251 0.25290132 0.28360401 0.31500767 0.34691142 0.37911853
## [13] 0.41143838 0.44368834 0.47569531 0.50729723 0.53834426 0.56869970
## [19] 0.59824082 0.62685928 0.65446147 0.68096854 0.70631624 0.73045463
## [25] 0.75334753 0.77497185 0.79531686 0.81438324 0.83218211 0.84873401
## [31] 0.86406782 0.87821966 0.89123181 0.90315161 0.91403047 0.92392286
## [37] 0.93288537 0.94097590 0.94825284 0.95477440 0.96059797 0.96577961
## [43] 0.97037358
```

```
# make a plot
plot(nrange, birthday_prob_vecs,
     main = "Probability two or more people have same birthday",
     xlab = "# of people",
     ylab = "Probability")
```



The above plot is the required plot for # of people against the probabilities. We can see, as the number of people in the room n increases, the probability of two or more people having same birthday also increases.

18

Write a program and simulation study regarding the Monty Hall problem

First, write a function that tells us if switch would win, or stay would win

```
monty_hall <- function() {
  # Assign the prize
  prize <- sample(1:3,1)
  # Pick a door
  choice <- sample(1:3,1)
  # Monty picks a door
  monty <- sample(c(1:3)[-c(choice,prize)], 1)
  return(ifelse(prize!=choice, "switch", "stay"))
}
```

Next, in order to see the probabilities of switch and win or stay and win, we run simulation study by repeating the above function several times and record the outcome

```
set.seed(2022)
N <- 2^(1:15)
choice <- data.frame(N=numeric(), switch=numeric())
for (trials in N) {
  run <- table(replicate(trials, monty_hall()))
  choice  <- choice  %>%  add_row(N=trials, switch=(sum(run["switch"]))/trials)
```

```r
}
# remove NA rows
choice[is.na(choice)]<-0
choice
```

```
##        N    switch
## 1      2 1.0000000
## 2      4 1.0000000
## 3      8 0.8750000
## 4     16 0.8750000
## 5     32 0.6875000
## 6     64 0.6093750
## 7    128 0.6250000
## 8    256 0.6757812
## 9    512 0.6621094
## 10  1024 0.6552734
## 11  2048 0.6450195
## 12  4096 0.6594238
## 13  8192 0.6683350
## 14 16384 0.6652222
## 15 32768 0.6668091
```

From the above table output, we see that as the number of replications increases, the switch-win probability increases (prob=0.667 when N=32768), and converge to around 2/3. Therefore, by simulation study, we can say switch would have a higher probability of winning.

19

Find 7 sexy prime quadruplets less than 1000

```r
prime_vec <- find_prime(1000)
for (i in 1:length(prime_vec)){
  num1 = prime_vec[i]
  num2 = num1 + 6
  num3 = num1 + 12
  num4 = num1 + 18
  if (is_prime(num2)==T & is_prime(num3)==T & is_prime(num4)){
    print(c(num1, num2, num3, num4))
  }
}
```

```
## [1]  5 11 17 23
## [1] 11 17 23 29
## [1] 41 47 53 59
## [1] 61 67 73 79
## [1] 251 257 263 269
## [1] 601 607 613 619
## [1] 641 647 653 659
```

The result above shows the 7 sexy prime quadruplets.

20

Use the code provided by Prof. Demirtas with some modification

```r
y=252
while (y <288){
 z=0
 while (z<315){
   # store the difference
 a=(4-y/72-z/63)*56-(5-y/56-z/63)*72
 # make sure its positive
 b=(4-y/72-z/63)*56
 # pick a small threshold
 if(abs(a)<=0.001 & b>=0)
   cat("From X to Y, uphill:",b,"downhill:",y,"level:",z,"distance:",b+y+z,"\n")
 z=z+0.01 }
 y=y+2 }
```

```
## From X to Y, uphill: 2 downhill: 254 level: 27.5 distance: 283.5
## From X to Y, uphill: 4 downhill: 256 level: 23.5 distance: 283.5
## From X to Y, uphill: 6 downhill: 258 level: 19.5 distance: 283.5
## From X to Y, uphill: 8 downhill: 260 level: 15.5 distance: 283.5
## From X to Y, uphill: 10 downhill: 262 level: 11.5 distance: 283.5
## From X to Y, uphill: 12 downhill: 264 level: 7.5 distance: 283.5
## From X to Y, uphill: 14 downhill: 266 level: 3.5 distance: 283.5
```

From the above result, we can see the total distance between X and Y is 283.5. Then we verify by hand.

Assume initial trip from X to Y, distance uphill=a, distance gound=b, distance downhill = c. We then have the following equations:

$a/56 + B/63 + C/72 = 4$ and $a/72 + b/63 + c/56 = 5$

If we multiple the LCM of 56,63,and 72 to both sides, we get the following: $9a + 8b + 7c = 2016$ and $7a + 8b + 9c = 2520$

Thus, if we add those above equations, we would get: $16a + 16b + 16c = 4536$ $a + b + c = 283.5$

```r
4536/16
```

```
## [1] 283.5
```

Therefore, the total distance between X and Y is 283.5.

21

Find all sphenic numbers that are less than 1000.

```r
prime_factors = function(x, i=2, factors = NULL){
  if(x<i) factors
  else if(! x %% i) prime_factors(x/i, i, c(factors, i))
  else  prime_factors(x, i+1, factors)
}

sphenic=c()
for(i in 2:1000){
  p_factor=prime_factors(i)
  if(length(p_factor)==3){
    for (j in 1:1){
      if(p_factor[1] %in% p_factor[-1] || p_factor[2] %in% p_factor[-2] || p_factor[3] %in% p_factor[-3
        break
      }else{
        sphenic=c(sphenic,i)
      }
    }
  }
}
sphenic
```

```
##   [1]  30  42  66  70  78 102 105 110 114 130 138 154 165 170 174 182 186 190
##  [19] 195 222 230 231 238 246 255 258 266 273 282 285 286 290 310 318 322 345
##  [37] 354 357 366 370 374 385 399 402 406 410 418 426 429 430 434 435 438 442
##  [55] 455 465 470 474 483 494 498 506 518 530 534 555 561 574 582 590 595 598
##  [73] 602 606 609 610 615 618 627 638 642 645 646 651 654 658 663 665 670 678
##  [91] 682 705 710 715 730 741 742 754 759 762 777 782 786 790 795 805 806 814
## [109] 822 826 830 834 854 861 874 885 890 894 897 902 903 906 915 935 938 942
## [127] 946 957 962 969 970 978 986 987 994
```

Above are all the sphenic numbers that are less than 1000.

22

```
taxicap2=function(n)
{
  k=1
  count=0
  taxi=c()
  final=0
  while(count<n){
    int_count=0
    for (i in 1:(ceiling(k^(1/3))+1)){
      for (j in (i+1):(ceiling(k^(1/3))+1)){
        if((i*i*i)+(j*j*j)==k){
          int_count= int_count+1
          taxi=c(taxi,paste0(i," & ",j))
        }
      }
    }
    if(int_count==2){
      count=count+1
    }
    final=paste0(count,": ",k)
    k=k+1
  }
  print(paste0("Taxicap Number: ", final))
  print(paste0("Numbers: ",taxi[(length(taxi)-1):length(taxi)]))

}
taxicap2(1)
```

```
## [1] "Taxicap Number: 1: 1729"
## [1] "Numbers: 1 & 12" "Numbers: 9 & 10"
```

```
#[1] "Taxicap Number: 1: 1729"
#[1] "Numbers: 1 & 12" "Numbers: 9 & 10"
```

```r
taxicap2(2)
```

```
## [1] "Taxicap Number: 2: 4104"
## [1] "Numbers: 2 & 16" "Numbers: 9 & 15"
```

```r
#[1] "Taxicap Number: 2: 4104"
#[1] "Numbers: 2 & 16" "Numbers: 9 & 15"
```

```r
taxicap3=function(n)
{
  #Starting k from 87520000 since this takes too long
  k=87520000
  count=0
  taxi=c()
  final=0
  while(count<n){
    int_count=0
    for (i in 1:(ceiling(k^(1/3))+1)){
      for (j in (i+1):(ceiling(k^(1/3))+1)){
        if((i*i*i)+(j*j*j)==k){
          int_count= int_count+1
          taxi=c(taxi,paste0(i," & ",j))
#          print(paste0(int_count," ",k))
        }
      }
    }
    if(int_count==3){
      count=count+1
    }
    final=paste0(count,": ",k)
    k=k+1
  }
  print(paste0("Taxicap Number: ", final))
  print(paste0("Numbers: ",taxi[(length(taxi)-2):length(taxi)]))

}
taxicap3(1)
```

```
## [1] "Taxicap Number: 1: 87539319"
## [1] "Numbers: 167 & 436" "Numbers: 228 & 423" "Numbers: 255 & 414"
```

```r
#[1] "Taxicap Number: 1: 87539319"
#[1] "Numbers: 167 & 436" "Numbers: 228 & 423" "Numbers: 255 & 414"
```

Taxicap Number: 1: 87539319 Numbers: 167 & 436, ,Numbers: 228 & 423, Numbers: 255 & 414.

Find all second order taxicab numbers that are less than 50,000.

```r
numbers <- 1:100
cubes <- numbers^3

# The possible pairs of numbers
pairs <- combn(numbers, 2)

# sum the cubes of the combinations
# This takes every couple and sums the values of the cubes with index
```

```r
sums <- apply(pairs, 2, function(x){sum(cubes[x])})
couples <- which(table(sums) == 2)

taxi.numb<- as.integer(names(couples))
taxi.numb[1:8]
```

```
## [1]  1729  4104 13832 20683 32832 39312 40033 46683
```

Above numbers are 2nd order taxicab number that are less than 50,000.

23

how many children were there? First, we assume the total amount of money is D. Since each children get the same amount, we can find out the amout that the first two chilren receive. First chilren: 1000 + 0.1(D-1000) = (1000 + 0.1D - 100) = (0.1D + 900); Second children: 2000 + 0.1[(0.9D-900)-2000] = 2000 + 0.1(0.9D-2900) = (2000 + 0.09D - 290) = (0.09D + 1710).

Next, our task is to find D which satisfies this equation

```r
num_kids <- function(){
  int_D = 1000
  repeat{
    D = int_D
    first_kid = 1000  + 0.1*(D - 1000)
    second_kid = (.09*D + 1710)
    if(first_kid == second_kid){break}
    else{int_D = int_D+100}

  }
  return(D)
}
```

```r
num_kids()/((0.1*num_kids() + 900))
```

```
## [1] 9
```

From the result above, we found D = 81000 which is the total amount of money. Then, the first children receives (0.1D + 900) = 9000. Thus, 81000/9000 = 9. Thus, from our result above, we can conclude there are 9 children.

24

the 9-digits puzzle. Arrange digits 1-9 into a 9 digit number, each digit only used once. For each N from 1-9, the first N digits of the number divisible by N.

We can first represent the 9-digit number as ABCDEFGHI. Thus, A is divisible by 1; AB is divisible by 2; ABC is divisible by 3 etc.

```r
digit9 = function(){

    all_solutions = numeric()
    for (a in 1:9){
        for (b in 1:9){
            for (c in 1:9){
                for (d in 1:9){
                    #for (e in 1:9){
                        for (f in 1:9){
                            for (g in 1:9){
                                for (h in 1:9){
                                  for (i in 1:9){
                                    if (length(unique(c(a,b,c,d,5,f,g,h,i)))==9){
                                        num9digit = a*1e8 + b*1e7 + c*1e6 + d*1e5 + 5*1e4 + f*1e3 + g*1

                                        if ((a*10+b) %% 2 == 0 &
                                            (a*1e2 + b*10 + c) %% 3 == 0 &
                                            (a*1e3 + b*1e2 + c*10 + d) %% 4 ==0 &
                                            (a*1e5 + b*1e4 + c*1e3 + d*1e2 + 5*10 + f) %% 6 ==0 &
                                            (a*1e6 + b*1e5 + c*1e4 + d*1e3 + 5*1e2 + f*10 + g) %% 7 ==0
                                            (a*1e7 + b*1e6 + c*1e5 + d*1e4 + 5*1e3 + f*1e2 + g*10 + h)
                                        all_solutions = append(all_solutions, num9digit)}
                                        }

                                  }}}}}}}}
      #}
    return(all_solutions)
}
```

```
digit9_vec <- digit9()
digit9_vec
```

```
## [1] 381654729
```

From the result above, we can see that the solution to this question is unique, and the answer is 381654729.

25

Find the least possible number of books in Leon's collection

```
find_books = function(){
  int = 10
  repeat{
    # numer of books eric gets
    eric = 2+(int-2)/5
    # number of books anubha gets
    anu = 6+(int-eric-6)/5

    # erice books > anubha books
    if((int-2)%%5==0 & (int-eric-6)%%5==0 & eric>anu){break}
    else {int = int+1}
  }


  return(int)
}
```

```
find_books()
```

```
## [1] 97
```

Thus, we can see from the result above, the least number of books in Leon collection is 97.

26

Find the magic number

We can write a function to do so

```r
find_magic_num <- function(N){
  int_num = N
  i = 0
  repeat{
    i = i+1
    pre_num = int_num
    digits = get_digits(int_num)
    next_num = sum(digits^3)
    if(next_num == pre_num){break}
    else{int_num = next_num}
  }

  return(next_num )
}
```

```r
find_magic_num(33)
```

```
## [1] 153
```

Next, we test with the multiples of 3 that are less than 100

```r
x = 1:33
mul3 = 3*x
for (i in mul3){
  print(find_magic_num(i))
}
```

```
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
## [1] 153
```

Therefore, from the above result, we can see the magic number is 153.

27

In the beginning apples:oranges:pears=7:11:9. Then, Anubhav ate 21 fruits, then

apples:oranges:pears=2:3:3

First, assume the total number of fruits = B. B should be a multiple of 27, so let B = 27b. Let A be what's left. A should be a multiple of 8, so let A = 8a.

Therefore, 27b-21=8a, b = (8a+21)/27. Thus, our task now is to find a such that (8a+21)/27 is an integer.

```
for (a in 1:50){
  if((8*a+21) %% 27 ==0 ){
    print(a)
  }
}
```

```
## [1] 21
## [1] 48
```

From the output above, we see that a=21. Therefore, there were 8a=168 fruits left and apple=42, oranges=63, pears=63. then, we can find the original amount of apples: $(168 + 21) * (7/27) = 49$

```
(168+21)*(7/27)
```

```
## [1] 49
```

49-42=7. Thus, he ate 7 apples.

In conclusion, there were 168 fruits left, and he ate 7 apples.

28

Show that $(a^2 + b^2)/(a * b + 1)$ is a square when it is an integer

```
int_vec <- numeric()
maxnum <- 1000
for (a in 1:maxnum){
  for (b in 1:maxnum){
    numer = a^2+b^2
    denom = a*b+1
    if (numer %% denom == 0){
    r = (a^2+b^2)/(a*b+1)
    #print(c(a,b))
    int_vec = append(int_vec, r)
    }
  }
}
int_vec
```

```
##  [1]   1   4   9  16  25  36  49   4   4  64  81 100   9   9   4   4  16   4   4
## [20]  25  36   9  49   4  64  81 100
```

From the result above, we found a and b such that (a^{2+b}2)/(a*b+1) is an integer. The above out put is the results for $(a^2 + b^2)/(ab + 1)$ for different values of a and b. From the results, we can see all of them are squares i.e. the sqrt of the ratio is also an interger.

```
sqrt(int_vec)
```

```
##  [1]  1  2  3  4  5  6  7  2  2  8  9 10  3  3  2  2  4  2  2  5  6  3  7  2  8
## [26]  9 10
```

29

```
# Total=n
# 1st point: if gorilla has more than 2000 -> 5 bananas / mile to transport (Total 5 trips)
# 2nd point: if gorilla has more than 1000 -> 3 bananas / mile to transport (Total 3 trips)
# last point: if gorilla has less than or equal to 1000 -> 1 bananas / mile to transport (Total 1 trips

# x=distance to first point, y=distance to second point, z= distance to last point= 1000-x-y
# first point (until 2000 bananas remains) : 3000-5x = 2000  -> x=200
# second point (until 1000 bananas remains) : 2000 - 3y = 1000
#  -> y=333.33=333 where distance cannot be fraction because it is related to the number of bananas
# which means, we do have 1001 bananas and have to throw 1 banana away. Thus 1000 banana remaining.
# last point (to the super market): z=1000-200-333 =467
#  -> bananas to the market = 1000-467=533 bananas
b=3000
x=y=z=0
for (d in 0:1000){
  if(b>2004){
    b=b-(5)
    x=x+1
  }else if(b>2000){   #Throw remaining bananas away. Since it is not worth it
    b=2000
  }else if (b>1002){
    b=b-(3)
    y=y+1
  }else if(b>1000){ #Throw remaining bananas away. Since it is not worth it
    b=1000
  }else if (b<=1000){
    b=b-(1)
    z=z+1
  }
}
b
```

```
## [1] 533
```

Therefore, the most bananas the Gorilla can carry is 533.

30.

Let a be the double digit number, and b be the single digit number a ranges from 10 to 99 b ranges from 0 to 9

```
l = numeric()
for (a in 10:99){
  for (b in 0:9){
    lian = a+b
    dong = a*b
    lian_rev = Reverse_number(lian)
    if(lian_rev == dong){print(c(a,b))}
  }
}
```

```
## [1] 24  3
## [1] 47  2
```

Thus, the two pairs are (24,3) and (47,2).

31.

Find all Fibonacci numbers less than N (N=1000). Find the golden ratios of two consecutive Fibonacci numbers.

First, write a function

```
find_fibs = function(N){
  fib <- c(0,1)
  counter <-3
  while (fib[counter-2]+fib[counter - 1]<N){
    fib[counter]<- fib[counter-2]+fib[counter-1]
    counter = counter+1
  }
  return(fib)
}
```

```
find_fibs(1000)
```

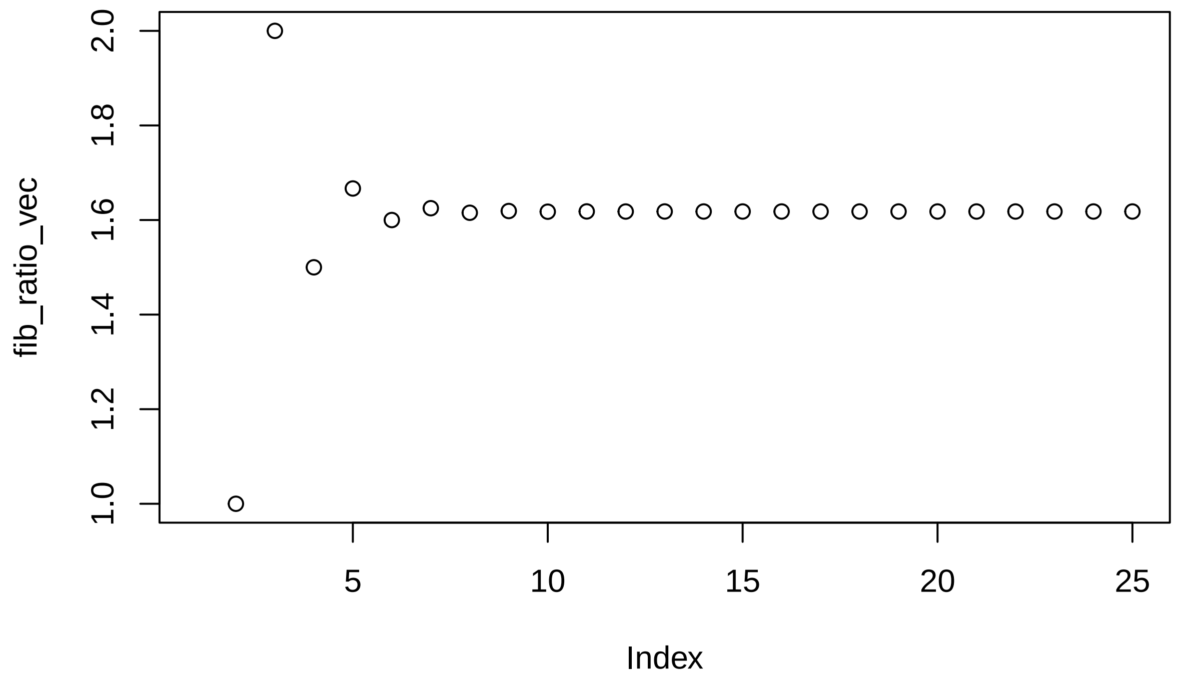
```
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
```

Above is the Fibonacci numbers less than 1000. Next, we will find the golden ratios of two consecutive fibonacci numbers

```
fib_ratio_vec <- numeric()
fib_vec <- find_fibs(100000)
for (i in 1:length(fib_vec)){
  if (i+1 > length(fib_vec)) {break}
  ratio = fib_vec[i+1]/fib_vec[i]
  fib_ratio_vec = append(fib_ratio_vec, ratio)

}
fib_ratio_vec
```

```
##  [1]      Inf 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385
##  [9] 1.619048 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037 1.618033
## [17] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
## [25] 1.618034
```

```
plot(fib_ratio_vec)
```



From the plot and results above, we see that the the ratio of two consecutive fibonacci number converges to 1.618034 which is the golden ratio.

32

a

Find the weird numbers less than 10,000.

```
subsetsum = function(S, t) {
  n = length(S)
  inds = NULL

  x = logical(n)
  y = numeric(t + 1)
  G = logical(t + 1)
  G[1] = TRUE
  for (k in 1:n) {
    H = c(logical(S[k]), G[1:(t + 1 - S[k])])
    H = (G < H)
    j = which(H)
    y[j] = k
    G[j] = TRUE
    if (G[t + 1]) break
  }
  wch = which(G)
  j = wch[length(wch)]
  fmax = j - 1
  while (j > 1) {
    k = y[j]
    x[k] = TRUE
    j = j - S[k]
  }
  inds = which(x)
  return(sum(S[inds]))
}

result=c()
for (i in 2:10000){
  a=seq_len(i)[i%%seq_len(i)==0]
  a=a[-length(a)]
  if (sum(a)>i && subsetsum(a,i)!=i){
    result=c(result,i)
  }
}
result
```

```
## [1]   70  836 4030 5830 7192 7912 9272
```

The weird numbers less than 10,000 are: 70, 836, 4030, 5830, 7192, 7912, 9272.

b

Find narcisistic number less than 10,000

```
for (number in 0:10000){
  digits = get_digits(number)
  n = length(digits)
  sumcube = sum(digits^n)
  if(n != 1 & sumcube == number){print(number)}
}
```

```
## [1] 153
## [1] 370
## [1] 371
## [1] 407
## [1] 1634
## [1] 8208
## [1] 9474
```

From the code above, we have found the 6 narcissist numbers other than 153 and less than 10,000.

33

Find Kaprekar numbers less than 10,000 First, write a number to check if a number is a Kaprekar number:

```
kap_num=c()
for (n in 1:10000){
  sq_n=n^2
  for (i in 1:(nchar(sq_n)-1)){
    n1=sq_n/(10^(i))
    a=as.integer(unlist(strsplit(as.character(n1),"[.]")))
    a[2]=sq_n-(a[1]*10^(i))
    if(sum(a)==n && !(is.na(a[2])) && a[2]!=0){
      kap_num=c(kap_num,n)
      break
    }
  }
}
kap_num
```

```
##  [1]    1    9   45   55   99  297  703  999 2223 2728 4879 4950 5050 5292 7272
## [16] 7777 9999
```

FRom the results above, the Kaprekar number that are less than 10,000 are: 1,9,45,55,99,297,703,999,2223,2728, 4879,4950,5050,5292,7272,7777,9999.

34

Find the integer whose square has no isolated digits

First, write a function

```
N = 2255
d <- get_digits(11222555)
d
```

```
## [1] 1 1 2 2 2 5 5 5
```

```
1 %in% rle(d)$lengths
```

```
## [1] FALSE
```

```
no_iso = function(N){
  d = get_digits(N)

  if(1 %in% rle(d)$lengths){return(FALSE)}
  else(return(TRUE))
}
no_iso(110011)
```

```
## [1] TRUE
```

We know if the squared number is at most 1e6, then the number should be at most 1e3, therefore, we modify the range to 1000 and report the original number.

```
a_vec = numeric()
for (r in 1:1e3){
  rsq = r^2
  if(no_iso(rsq)){a_vec = append(a_vec, r)}
}
a_vec[1]
```

```
## [1] 88
```

From the result above, we see that this number is 88, and 88ˆ2 = 7744 which has no isolated digits. Piano keyboards also have 88 keys.

35

```r
set.seed(2023)
l=c(3, 5, 7, 11, 13, 17, 19, 23)
for (i in 1:100000){
  a=sort(sample(l,3,replace=FALSE),decreasing = FALSE)
  b=sort(sample(l,3,replace=FALSE),decreasing = FALSE)
  c=sort(sample(l,3,replace=FALSE),decreasing = FALSE)
  d=sort(sample(l,3,replace=FALSE),decreasing = FALSE)
  if (!(identical(a,b)) && sum(a)==sum(b)){
    if (!(identical(c,d)) && sum(c)==sum(d)){
      if (!(identical(b,c)) && sum(b)==sum(c)){
        if (!(identical(a,d)) && sum(a)==sum(d)){
          if (!(identical(b,d)) && sum(b)==sum(d)){
            if (!(identical(a,c)) && sum(a)==sum(c)){
              m=k=p=0
              while(m<2){
                for(n in 1:3){
                  if(a[n] %in% c(b,c,d)){
                    m=m+1
                  }
                }
                break
              }
              while(k<2){
                for(n in 1:3){
                  if(b[n] %in% c(a,c,d)){
                    k=k+1
                  }
                }
                break
              }
              while(p<2){
                for(n in 1:3){
                  if(c[n] %in% c(a,b,d)){
                    p=p+1
                  }
                }
                break
              }
              if(m==2 && k==2 && p==2){
                print(paste0("sum :",sum(a)))
                print(a)
                print(b)
                print(c)
                print(d)
                break
              }
            }
          }
        }
      }
    }
  }
}
```

```
## [1] "sum :31"
## [1]  7 11 13
## [1]  5  7 19
## [1]  3 11 17
## [1]  3  5 23
```

```
#3   17   11
#23  xx   13
#5   19    7
#Thus, if we order these 4 lists into our table, we would get the above result
```

36

N queens problem. Here N=7. To solve this problem, we used recursive backtracking.

```
# defina function that
queens = function(n) {
  a = seq(n)
  u = rep(T, 2 * n - 1)
  v = rep(T, 2 * n - 1)
  m = NULL
  aux = function(i) {
    if (i > n) {
      m <<- cbind(m, a)
    } else {
      for (j in seq(i, n)) {
        k = a[[j]]
        p = i - k + n
        q = i + k - 1
        if (u[[p]] && v[[q]]) {
          u[[p]] <<- v[[q]] <<- F
          a[[j]] <<- a[[i]]
          a[[i]] <<- k
          aux(i + 1)
          u[[p]] <<- v[[q]] <<- T
          a[[i]] <<- a[[j]]
          a[[j]] <<- k
        }
      }
    }
  }
  aux(1)
  m
}
library(Matrix)
a=queens(7)
a[,1]
```

```
## [1] 1 3 5 7 2 4 6
```

Above is one of the solutions to the 7 queens problem.

```r
a
```

```
##      a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
## [1,] 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6 6 7
## [2,] 3 4 5 6 4 4 5 5 5 6 7 1 1 5 6 7 7 2 1 1 6 7 7 2 3 1 1 7 7 2 3 3 3 4 4 1 2
## [3,] 5 7 2 4 1 6 3 1 7 3 5 6 6 7 2 4 2 7 3 5 1 3 5 6 1 4 6 2 2 5 5 1 7 2 7 3 4
## [4,] 7 3 6 2 7 1 1 4 4 7 3 4 2 2 5 1 4 5 6 2 3 6 2 3 6 7 4 4 6 1 7 4 4 7 1 5 6
## [5,] 2 6 3 7 5 3 7 7 1 4 1 2 5 4 1 5 6 3 2 6 5 2 6 7 4 3 2 6 3 4 1 7 1 5 3 7 1
## [6,] 4 2 7 5 3 5 4 3 3 1 6 7 7 6 4 2 1 1 7 3 7 5 1 4 2 6 7 1 1 7 4 5 5 3 5 2 3
## [7,] 6 5 4 3 6 7 6 6 6 5 4 5 4 1 7 6 5 6 5 7 2 1 3 1 7 2 3 3 4 3 2 2 2 1 2 4 5
##      a a a
## [1,] 7 7 7
## [2,] 3 4 5
## [3,] 6 1 3
## [4,] 2 5 1
## [5,] 5 2 6
## [6,] 1 6 4
## [7,] 4 3 2
```

Above table is showing that where Queens are for each row. For example, for the first solution (first column from table above): (row1-col1, row2-col3, row3-col5, row4-col7, row5-col2, row6-col4, row7-col6)

```r
length(a[1,])
```

```
## [1] 40
```

There are total 40 solutions to 7 queens problem.

37

The broken stick problem. We try with 10000 cuts

```r
set.seed(37)
L1=runif(10000,0,1)
L2=runif(10000,0,1)
tf=c()
for (i in 1:length(L1)){
  l=sort(c(L1[i],L2[i]),decreasing=FALSE)
  a=l[1]
  b=l[2]-l[1]
  c=1-(a+b)
  tf=c(tf,(a<0.5 && b<0.5 && a+b>c))
}
sum(tf==TRUE)/length(tf)
```

```
## [1] 0.2515
```

```r
#[1] 0.2515
```

Therefore, the probability of the 3 sticks forming a triangle is around 0.25.

38

SEND + MORE = MONEY

```r
sol = function(){
    # letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')
    all_solutions = numeric()
    for (s in 9:0){
        for (e in 9:0){
            for (n in 9:0){
                for (d in 9:0){
                    for (m in 9:0){
                        for (o in 9:0){
                            for (r in 9:0){
                                for (y in 9:0){
                                    if (length(unique(c(s,e,n,d,m,o,r,y)))==8){
                                        send = 1000 * s + 100 * e + 10 * n + d
                                        more = 1000 * m + 100 * o + 10 * r + e
                                        money = 10000 * m + 1000 * o + 100 * n + 10 * e + y

                                        if (send + more == money)
                                            all_solutions = append(all_solutions, c(send, more,money))}

                                }}}}}}}}
    return(all_solutions)
}
```

```
sols <- sol()
```

```
sol_mat <- matrix(sols, ncol = 3,byrow = T)
sol_mat[1,]
```

```
## [1]  9567  1085 10652
```

From the result above, we see that the solution to SEND+MORE=MONEY is: 9567+1085=10652. s = 9; e = 5; n = 6; d = 7; m = 1; o = 0; r = 8; y = 2.

39

Write a program to tell the story from the video - Prime number theorem (the density of primes)

prime density ~ 1/ln(x) number of primes less than x ~ x/(ln(x))

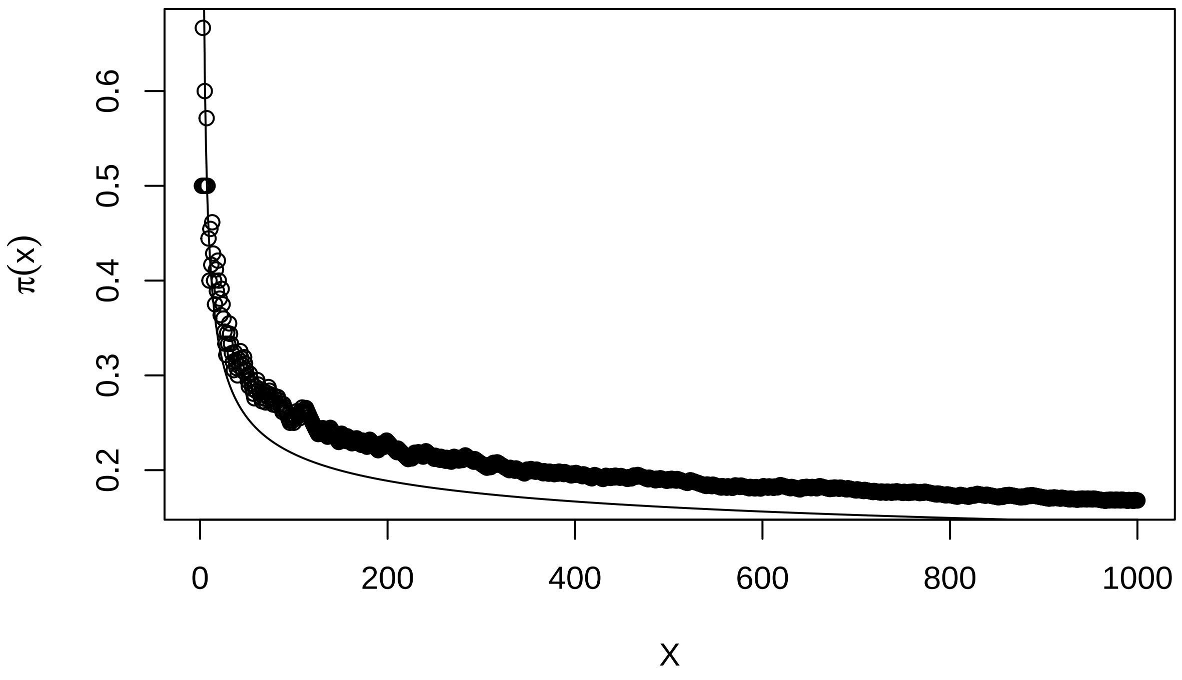
Part 1: we want to show prime density ~ 1/ln(x) First, we write a function that calculates the prime density use formula #number of primes less than N/N

```
prime_density = function(X){
  num_prime = length(find_prime(X))
  dens = num_prime/X
  return(dens)
}
# test
prime_density(100)
```

```
## [1] 0.25
```

```
X <- 2:1000

vec1<- numeric(length(X))
for (i in X){
  dens = prime_density(i)
  vec1[i-1] = dens
}
```

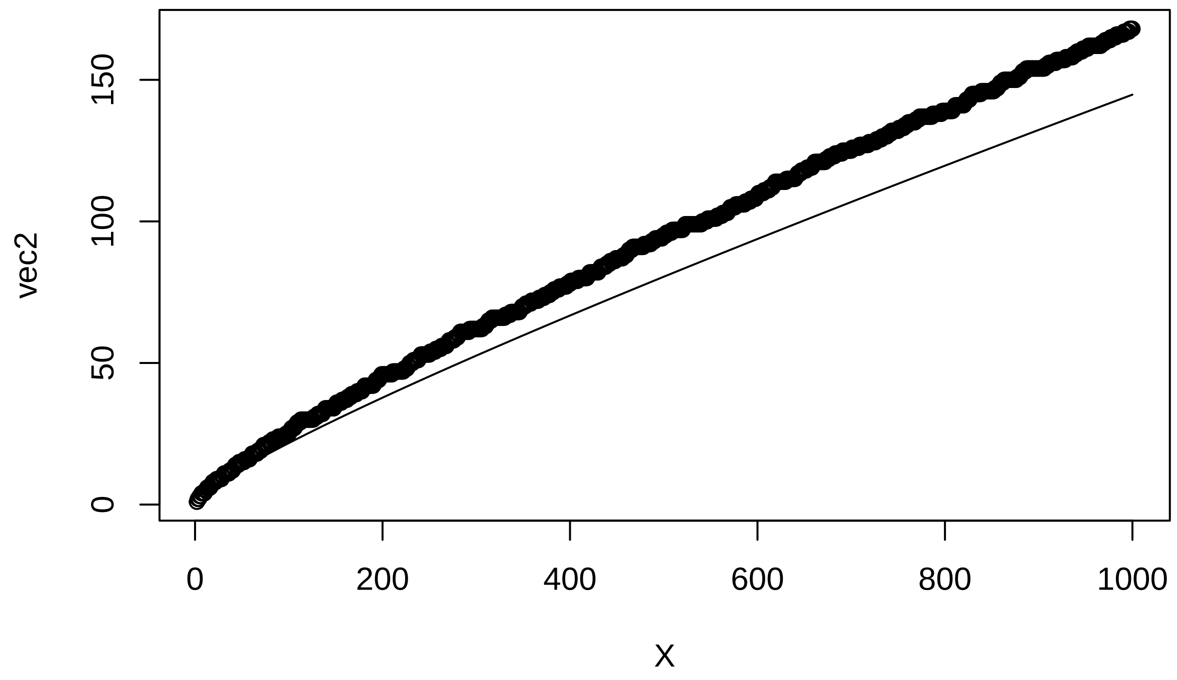
```r
y <- 1/(log(X))
plot(vec1~X,
     ylab=expression(pi(x)))
lines(y ~ X)
```



From the above plot, the dotted line represents the prime density, and the straight line represents the 1/(log(x)) from the video. WE could see as X gets larger, those two lines converges.

```r
X <- 2:1000

vec2 <- numeric(length(X))
for (i in X){
total_prime = length(find_prime(i))
vec2[i-1] = total_prime
}
```

```r
z = X/log(X)
plot(vec2 ~ X)
lines(z ~ X)
```



From the plot above, the dotted line represents the total number of primes less than X, and the straight line represents the X/(log(X)). We can see that as X increases, those two lines tend to converge to each other.

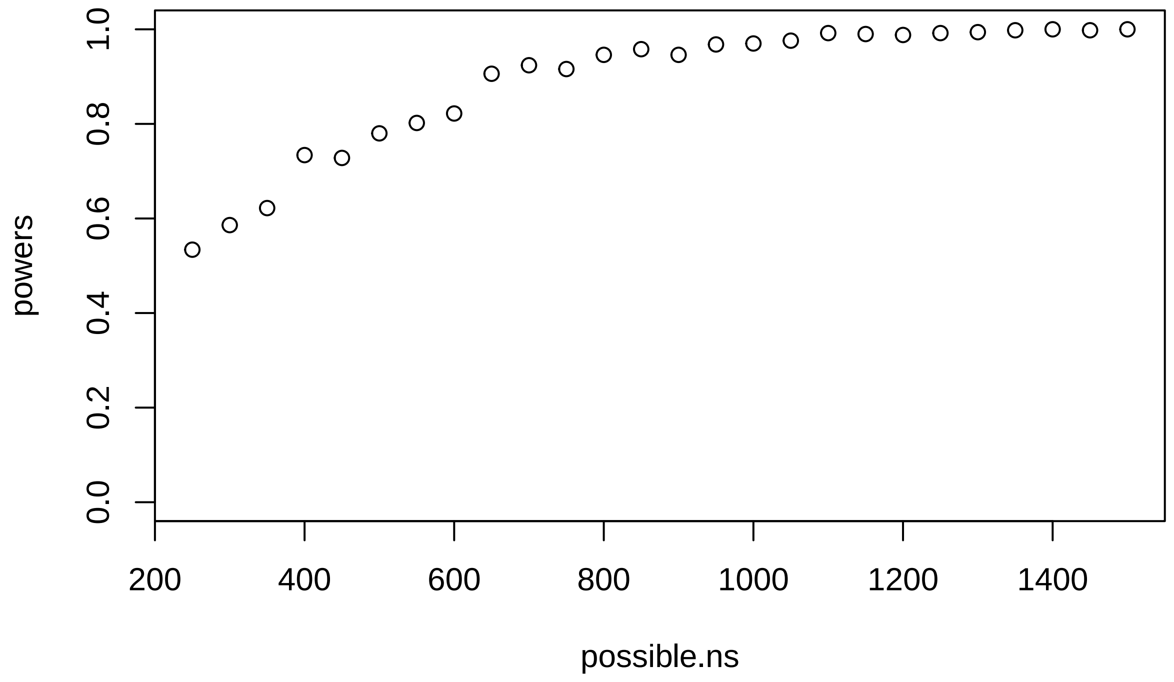
40

Design a simulation study concerning sample size and power.

Below, we designed a simulation study concerning sample size and power. For this study, our sample ranges from 250 to 1500 with step size 50. Our alpha is picked to be 0.05. our number of simulation for each sample size N is picked to be 500. We first generate Y0 from a normal distribution with parameters mean 60 and sd 20, then we hypothesized a treatment effect of 5, then we use Y0+tau to create treatment outcome. Then, we randomly generate N binomial assignments (0 and 1). Then we performed a simple linear regression N times and record the average number of p-values less than alpha level - this is then the power recorded for each sample size. In the end, we also plotted a power curve - pwer against sample size.

```
possible.ns <- seq(from=250, to=1500, by=50)     # The sample sizes we'll be considering
powers <- rep(NA, length(possible.ns))           # Empty object to collect simulation estimates
alpha <- 0.05                                    #significance level
sims <- 500                                      # Number of simulations to conduct for each N

#### vary the number of subjects
for (j in 1:length(possible.ns)){
  N <- possible.ns[j]                           # Pick the jth value for N

  significant.experiments <- rep(NA, sims)     # Empty object to count significant experiments

  #### conduct experiments "sims" times over for each N
```

```r
  for (i in 1:sims){
    Y0 <-  rnorm(n=N, mean=60, sd=20)          # control potential outcome
    tau <- 5                                   # Hypothesize treatment effect
    Y1 <- Y0 + tau                             # treatment potential outcome
    Z.sim <- rbinom(n=N, size=1, prob=.5)      # Do a random assignment
    Y.sim <- Y1*Z.sim + Y0*(1-Z.sim)           # Reveal outcomes according to assignment
    fit.sim <- lm(Y.sim ~ Z.sim)               # perform an analysis using simple linear regression
    p.value <- summary(fit.sim)$coefficients[2,4]  # Extract p-values
    significant.experiments[i] <- (p.value <= alpha) # Determine significance according to p <= 0.05
  }

  powers[j] <- mean(significant.experiments)       # store average success rate (power) for each N
}
plot(possible.ns, powers, ylim=c(0,1))
```



From the power plot above, we could see that as sample size N increases, the power also increases.