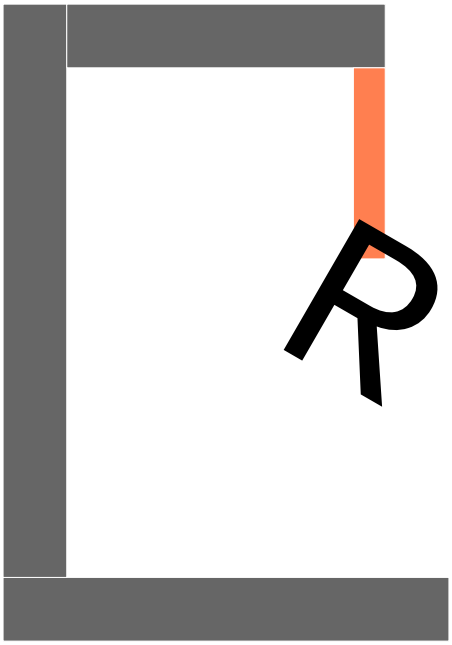
Life After Death by



also known as

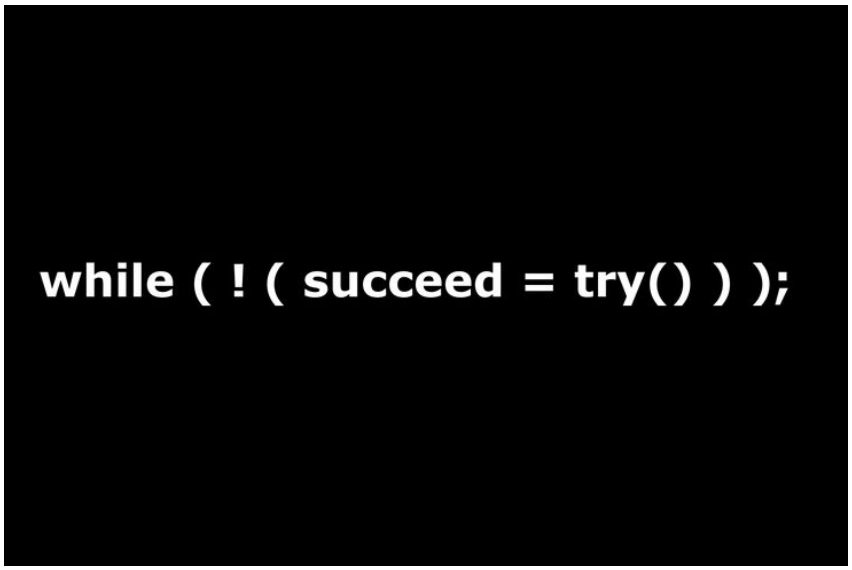
How to solve it using R

Mihir Arjunwadkar

2015

Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

Universal algorithm for solving a problem



provided that a solution exists

Image courtesy: this blog page

Contents

Preface

What is this book about? This book is about the programming language and environment called R. This is a self-learning book for those who like to learn a programming language by solving problems.

The form and content of this book is motivated by the following singleton "data" and the highly biased "inference" that was forged from it: When I ran into R for the first time, all that I was told was `<-`, `c`, `seq`, `:`, and `plot`. The rest was left to me to learn by myself, through guesswork, trial and error, and my own mistakes. This worked because, I believe, there were problems that I needed to solve using R rather quickly. As such, the emphasis was naturally on learning only the bare essentials strictly on a need basis. Contrary to what common sense told me back then, this helped me learn R rather quickly. Now, the hope is that this singular "experiment" is replicable.

The subtitle of this book is clearly indebted to the far deeper book *How to solve it* by mathematician G. Polya, and the book *How to solve it by computer* by R. Geoff Dromey.

Death by R. The book assumes some minimal familiarity with the R programming language: Syntax, basic data types (`numeric`, `integer`, `logical`, and `character`), common container types (`vector`, `matrix`, `data.frame`, and `list`), assignment, loops, conditionals, functions, etc. Some familiarity with programming in general will be a plus. The learner is also assumed to know how to find information about R; e.g., through the R help system (`help`, `?`, `help.search`, `??`, etc.), CRAN, trial and error, intelligent guesswork, big brother Google, asking a friend, asking a foe, etc. This is the unwritten death-by-R prequel to this book.

Life thereafter. With this background, this book tries to expose the learner to more of R through R codes that embody computational solutions to problems. This is the resurrection and redemption part, that is, life after death-by-R.

How to use this book? Like any art that is worth investing time and effort into, programming cannot be learnt theoretically just by reading books; it needs to be learnt through one's own mistakes. Hands-on exploration and practice are thus of paramount importance to this self-learning approach. Essential skills to be learnt include turning an algorithm into code from its description (plain-english, pseudocode, or mathematical), and tracing causality in the flow of computation.

Perhaps the best way to use this book as an aid to learning R would be to try solving a problem oneself first – at least partially, and only then look at the solution or solutions presented. As with any programming language, there can be many routes – good, bad, or ugly – to reach the same end goal. So, by all means, your solution to a problem may very well turn out to be a better one than the ones presented. A comparative study of different solutions, programming styles, and algorithmic thinking styles often leads to better understanding of a programming language, and offers better insight into programming in general.

The internals of the R codes presented here are, by and large, left for the reader to understand and assimilate with the help of interspersed comments, and through the learner's own

resourcefulness in finding information about new (or old) features of R that (s)he may discover occasionally.

There is no particular order to the problem included here, nor have they been ranked by their levels of difficulty. Exercises are sometimes stated imperatively; at other times, they are left half-stated with the view that a sketch is a better stimulant to imagination than a perfected painting.

Copy-paste. If you copy-paste R codes from this PDF book, manual verification of the pasted codes may be necessary for at least two reasons:

- LaTeX (or the `listings` package) often replaces plain-text ASCII quote characters with alternate quote characters. This will cause problems in R if pasted without manual curating.
- LaTeX (or the `listings` package) may have introduced additional spaces in the codes. These may or may not cause syntax errors, but it would be prudent to verify the copy-pasted codes.

Coding style.

- The style of problem-solving here is influenced by ideas and viewpoints related to top-down design and stepwise refinement. Programming style, by and large, is closer to the procedural-imperative end of the spectrum. Maxims that have influenced the programming style in this book include, e.g., *Programs must be written for people to read, and only incidentally for machines to execute*, and *The purpose of computation is insight, not numbers.*
- The occasional practice in this book of putting opening braces '{' on a separate line is intended to bring out the structure of the code better. This practice is not universally accepted. If not used with care, it may cause unintended syntax errors or side effects in R.
- I take the view that the purpose of a `function` is twofolds: avoiding code repetition, and encapsulating a computational element that can be used outside of the context in which it was encountered. As such, the primary channels of communication between a `function` and the caller environment should be the input arguments and the return value, and not global variables. For the same reason, direct input/output, especially to the screen, should be avoided except for critical error conditions, or when the purpose of the `function` is input/output. Error-handling is done, by and large, via `stopifnot`.
- Often, the value of one expression (e.g., a `function` call) is fed to another through nested expressions. This makes it crisp and sometimes efficient, but perhaps a bit difficult to read and understand for a beginner. The way to understand what such nested expressions lead to is to get the expressions evaluated starting from the innermost. Essentially, this is the breaking-down-the-apparent-complexity-of-a-nested-expression approach to understanding nested expressions.
- End-of-`function` calls to `return` can often be redundant in R. Even when redundant, explicit `return` calls are used for the sake of clarity.
- For graphics, I have (mostly) used the bare-basics `graphics` package. Many other packages for producing more sophisticated or special-purpose graphics exist on CRAN. This exercise is, however, best left to the needy learner for the occasion when (s)he needs those special features most desperately.

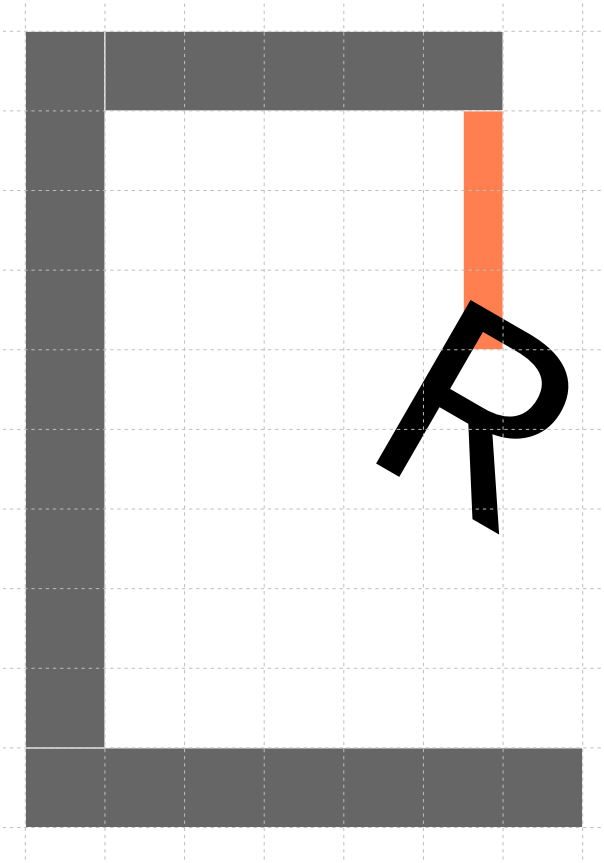
Acknowledgments.

- Abhijat Vichare, Leelavati Narlikar, Dipanjan Mitra, Bhalchandra Gore, Bhalchandra Pujari: for enthusiasm and encouragement, for contributing or suggesting problems, for discussions.
- Students: for willingly suffering death-by-R, then this book, and a terrible teacher all through.

The death-by-R graphic

PROBLEM

How might one re-create the death-by-R title page graphic for this book as faithfully as possible?



APPROACH/ES

The overlaid dotted lines in the graphic above should help get the proportions right. Color of the vertical noose, as well as the color, placement, rotation, font and size of the dead R are left to guesswork and trial-and-error.

POSSIBLE SOLUTION/S

It should not be much of a surprise that the following is the exact code that produced both the death-by-R graphics:

```
pdf( 'death-by-R.pdf', width = 7, height = 10, paper = 'special', useDingbats = F )
 plot.new()

 old.par <- par( mar = rep( 0, 4 ) ) # save the original plot settings
  plot.window( xlim = c( 0, 0.7 ), ylim = c( 0, 1 ) )

  rect( xleft   = c( 0,   0,   0.1, 0.55 ),
        ybottom = c( 0,   0.1, 0.9, 0.6  ),
        xright  = c( 0.7, 0.1, 0.6, 0.6  ),
        ytop    = c( 0.1, 1,   1,   0.9  ),
        density = -1, col = c( rep( 'gray40', 3 ), 'coral' ), border = 'white' )

  text( 0.56, 0.505, 'R', srt = -30, family = 'mono', cex = 20 )

  # overlaid grid for the second death-by-R figure
  # abline( v = 0.1 * ( 0:7 ),  h = 0.1 * ( 0:10 ), lty = 2, lwd = 1, col = 'gray' )
 par( old.par ) # restore plot settings: redundant here, but generally a good practice
dev.off()
```

Counting people from handshakes

PROBLEM

Image courtesy: Wikipedia

A special interdisciplinary meeting is organized where a few computer scientists and some biologists are invited. First, the two groups meet separately: Each person shakes hands with every other person within only her/his own group. In other words, computer scientists shake hands with computer scientists, while biologists shake hands with biologists. There are a total of 102 such handshakes. After that, all computer scientists shake hands with all biologists. There are a total of 108 such handshakes across the two groups. How many computer scientists attended the meeting?

APPROACH / ES

Formulate the problem mathematically first. This should suggest way (or ways) of approaching and solving the problem computationally.

POSSIBLE SOLUTION / S

Formulation. Following the standard mathematical practice of pretending to know the unknown by naming it, let us suppose there are n computer scientists and m biologists. Then the description above tells us that there were

$$\binom{n}{2} + \binom{m}{2} = 102 \text{ within-group handshakes, and} \quad (1)$$

$$nm = 108 \text{ across-group handshakes.} \quad (2)$$

Here, $\binom{k}{2} = k(k-1)/2$, the number of distinct pairs in a group of size k. The two equations above need to be solved for positive integer values of n, m. The two equations are symmetric in n and m, which means that we expect to see even number of solution pairs (n, m). Let us assume that the two equations are solvable; i.e., they do have positive integer solutions for n, m.

Solution 1: pure brutality. Let us define an `R function` that computes the LHSs of the two equations above given n and m, and use this to arrive at the correct values of n, m computationally.

```
handshakes <- function( n, m )
 {
  # enforce assumptions
```

```
  stopifnot( length( n ) == 1, n == as.integer( n ), n > 0,
             length( m ) == 1, m == as.integer( m ), m > 0 )

  return( c( choose( n, 2 ) + choose( m, 2 ), n * m ) )
  # [1]: within-group, [2]: across-group.
 }

k <- 108 # natural upper bound on n and m thanks to the second equation

for ( n in 1:k ) # brutal search through double loop
 for ( m in 1:k )
  if ( all( handshakes( n, m ) == c( 102, 108 ) ) ) cat( n, m, '\n' )
```

Solution 2: solution 1, but less brutal. In the above solution, the double loop is unnecessary, and computation scales as $\texttt{k}^2$. In the enumerative solution below, computation scales as `k` instead. Eliminate m from the two equations above; i.e., put $m = 108/n$ into the first equation. This gives us

$$n^4 - n^3 - 204n^2 - 108n + 108^2 = 0. \tag{3}$$

Positive integer solutions to this equation, together with $m = 108/n$, will give us solutions to the original problem:

```
p <- function( n ) { return( n^4 - n^3 - 204 * n^2 - 108 * n + 108^2 ) }
k <- 108 # upper bound on both n and m thanks to the second equation
n <- 1:k # starting from 1 is important, because we want positive integer solutions

n.star <- which( p( n ) == 0 ) # these are the solutions for n; m is 108 / n
m.star <- 108 / n.star # the two solutions: c(nstar[1],mstar[1]), c(nstar[2],mstar[2])
```

This can be nicely visualized in the form of roots of the above polynomial:

```
nlim <- c( 1, max( n.star ) + 3 ) # for a better-looking plot
plim <- range( p( nlim[1]:nlim[2] ) )

plot( n, p( n ), xlim = nlim, ylim = plim,
      type = 'b', axes = F, col = 'red', pch = 21, bg = 'green' )

abline( h = 0, v = n.star, lty = 2 ) # vertical lines: solutions for n

axis( 1 )
axis( 2, at = 0, tick = F, las = 1 )
axis( 3, at = n.star, tick = F )
```

Solution 3: less brutal of the second kind. Eq. 2 tells us that n, m are integer divisors of 108; i.e., 1, 2, 3, 4, 6, 9, 12, 18, 27, 36, 54, or 108. Pairs of divisors that satisfy Eq. 2 are (1,108), (2,54), (3, 36), (4,27), (6,18), and (9,12). Through trial-and-error, one can see that the two pairs that satisfies Eq. 1 are (9,12) and (12,9). Below is a quick-and-dirty (Q&D) way of finding all divisors of a positive integer, followed by a solution to the handshake problem:

```
k <- 108

divisors <- NULL; for ( i in 1:k ) if ( ( k %% i ) == 0 ) divisors <- c( divisors, i )

for ( n in divisors )
 {
  m <- k / n
  if ( all( handshakes( n, m ) == c( 102, 108 ) ) ) cat( n, m, '\n' )
 }
```

Recognizing and encapsulating a generic element of computation for reuse. In the last approach, we ran into the problem of finding all divisors of a positive integer. Although we solved it in a Q&D way, we recognize that this is a generic problem that may show up in other contexts also. So it would be good to encapsulate this computation in the form of an

R function that can be used elsewhere. Once debugged and validated thoroughly, this helps avoid code replication and saves effort the next time this problem is encountered.

```
divisors <- function( k, as.pairs = FALSE )
 {
  # enforce assumptions about the argument k:
  stopifnot( length( k ) == 1, k > 0, k == as.integer( k ) )
  # http://rosettacode.org/wiki/Factors_of_an_integer#R
  # handles the case length( k ) > 1 recursively.

  # divisors, the inefficient way in R:
  # d <- NULL; for ( i in 1:k ) if ( ( k %% i ) == 0 ) d <- c( d, i )

  # divisors, the efficient way in R but with the same enumerative approach:
  # d <- which( ( k %% ( 1:k ) ) == 0 )

  # algorithmic improvements are certainly possible here.
  # the simplest improvement limits the enumeration to sqrt(k) instead of k:
  d <- which( ( k %% ( 1:sqrt( k ) ) ) == 0 ) # the other set of divisors is k/d.

  # if nothing further needs to be done, then return the divisors in ascending order:
  if ( !as.pairs ) return( c( d, rev( k / d ) ) )

  # it may be useful to arrange divisors as pairs that multiply to k.
  # a convenient representation for this is a 2-column matrix whose rows multiply to k:
  return( matrix( c( d, k / d ), ncol = 2 ) )
 }
```

Solution 3 revised. Using our functions `divisors` and `handshakes`, solution 3 can be redesigned as follows:

```
hs1 <- function( d ) # this is a Q&D implementation; compare with hs2 further on.
 {
  if ( all( handshakes( d[1], d[2] ) == c( 102, 108 ) ) ) return( d )
  return( NULL )
 }

# one solution:
solution1 <- unlist( apply( divisors( 108, as.pairs = TRUE ), MARGIN = 1, hs1 ) )

# the other solution:
solution2 <- rev( solution1 ) # rev is preferable to solution1[length( solution1 ):1]
```

Solving a more general problem. Suppose the RHSs of Eq. 1 and 2 are allowed to take arbitrary positive integer values; i.e.,

$$\binom{n}{2} + \binom{m}{2} \stackrel{?}{=} k_1 \quad \text{(within-group handshakes)} \tag{4}$$

$$nm \stackrel{?}{=} k_2 \quad \text{(across-group handshakes)} \tag{5}$$

where k_1, k_2 are pre-specified positive integers, and the idea is to solve for n, m given k_1, k_2. Not every pair of integers k_1, k_2 can be associated with positive integer values of n, m (which is the reason for the notation $\stackrel{?}{=}$ for questioned equality). Suppose we represent the two integer pairs (k_1, k_2) and (n, m) in the form of 2-component R vectors `k` and `n * m` respectively, then one solution to the generalized problem, based on the revised solution 3 above, could be this:

```
hs2 <- function( nm, k )
 {
  # enforce assumptions about the arguments
  stopifnot( length( k  ) == 2, all( k  == as.integer( k  ) ), all( k  > 0 ),
             length( nm ) == 2, all( nm == as.integer( nm ) ), all( nm > 0 ) )

  # return nm if the arguments nm and k satisfy Eq. 3 & 4.
  if ( all( handshakes( nm[1], nm[2] ) == k ) ) return( nm )
```

```
  # proxy for no solution
  return( NULL )
 }

k  <- c( 102, 108 )
nm <- unlist( apply( divisors( max( k ), TRUE ), MARGIN = 1, FUN = hs2, k = k ) )
# the other solution is rev( nm )

k  <- c( 51, 54 )
nm <- unlist( apply( divisors( max( k ), TRUE ), MARGIN = 1, FUN = hs2, k = k ) )
# the other solution is rev( nm )

k  <- c( 50, 54 )
nm <- unlist( apply( divisors( max( k ), TRUE ), MARGIN = 1, FUN = hs2, k = k ) )
# no solution for this k1, k2; is.null( nm ) == TRUE
```

An alternate computational route: polynomial roots. Eq. 3 (or its more general variant involving k_1, k_2) is a polynomial equation, and the solutions n are zeros/roots of the polynomial on the LHS. All zeros/roots of a polynomial – real or complex – can be obtained using the `R function polyroot`. Eq. 3 is a degree-4 polynomial, so it has 4 zeros/roots. Two of these form the complex-conjugate pair $-10 \pm \sqrt{8}i$, and the other two are the required zeros/roots $n = 9$ and $n = 12$. Incidentally, both pairs of roots multiply to 108. This elegant approach can, in principle, be used for any values on the RHS of Eq. 4 and 5 (see the generalized problem below), including those for which there is no positive integer solution for n, m. However, this approach can be somewhat involved computationally, because `polyroot` returns all zeros/roots as `complex` numbers, and one needs to filter-out the correct (i.e., positive integer) zeros/roots keeping in mind the idiosyncrasies of floating-point arithmetic[1]. An essential building-block for this approach is checking whether the value of a `complex` number `z` is *essentially* positive integer:

```
has.integer.value <- function( z, small = sqrt( .Machine$double.eps ) )
 {
  ( Im( z ) < small ) & ( abs( Re( z ) - round( Re( z ), digits = 0 ) ) < small )
 }

has.positive.integer.value <- function( z, small = sqrt( .Machine$double.eps ) )
 {
  has.integer.value( z ) & ( round( Re( z ), digits = 0 ) > 0 )
 }
```

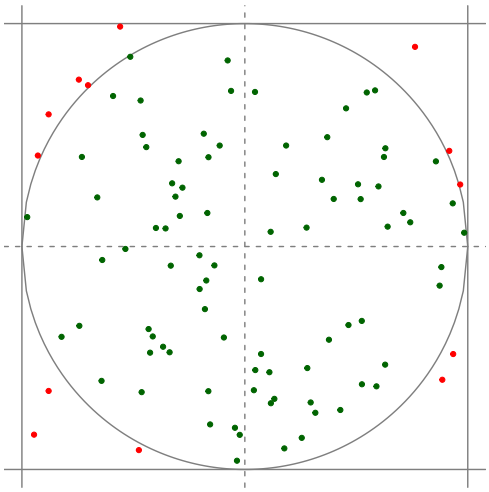
Given this, you are welcome to take this approach to its logical end.

[1] See also: David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, Pages 5–48, Volume 23, Issue 1, March 1991.

π by dartboard

PROBLEM

The celebrated number π can be estimated through darts thrown at random at a simplistic dartboard (see figure). Simulate the experiment of throwing N random darts at this dartboard so that they fill up the dartboard with uniform density. Using this, estimate the value of π.



A random sample of 100 dart hits.

APPROACH / ES

Consider a square dartboard with a circle inscribed inside (see figure). Suppose the dartboard is defined by the corner points $(-1,-1),(-1,+1),(+1,+1),(+1,-1)$, and the inscribed circle is the unit circle $x^2+y^2=1$. A random dart hit (x,y) inside the square can be simulated by generating two uniform random numbers over $[-1,+1]$. Suppose you throw N darts on this dartboard at random so that they fill up the dartboard with uniform density. Out of these N hits, suppose $N_\circ$ fall inside the circle (i.e., $x^2+y^2\leq 1$). Count these $N_\circ$ points inside. The ratio $\widehat{\pi}_N = 4N_\circ/N$ should be a reasonable guess/estimate for π.

WHY WOULD THIS APPROACH WORK?

By assumption, we are filling up the square dartboard with uniform density of dart hits, and the process described above ensures this. Therefore, the proportion of darts hitting inside the circle is (area of the circle) / (area of the square) $= \pi/4$. Therefore, for a sufficiently large number of hits, (number of hits inside the circle) / (total number of hits) $\approx \pi/4$. Inverting this, one gets the above estimator for π.

Because π is different from a guess/estimate of π, we use the notation $\widehat{\pi}_N$ for the estimate. Remember that π is a constant. In contrast, because this prescription involves randomness, every new realization of N dart hits will, in general, give us a different $N_\circ$ and, hence, a different estimate of π. Any single estimate $\widehat{\pi}_N$ we may care to look at is going to be different from π. So should we believe what we get out of this prescription? We should check if it really works.

One route to this goal is to take the formal probability theory route and prove results. The other route is to explore the behaviour of this prescription using computation. Specifically, a useful direction is to see if and how the estimates $\widehat{\pi}_N$ depend on N. Given the randomness, for

each N, it would be useful to work with some large number M of estimates $\widehat{\pi}_N^{(i)}, i = 1, \ldots, M$. The purpose of this large number M, say $M = 1000$, is simply to make sure that any patterns in the variability due to randomness are adequately captured in the collection $\widehat{\pi}_N^{(i)}, i = 1, \ldots, M$. Given a large collection of numbers, how does one make sense out of it? Standard statistical summaries should be of help here. For example, one could summarize a large collection of numbers using 5-number summary (`function quantile`, or a visual representation called the `boxplot`), or the `histogram`. These should help bring out the collective/average behaviour of the prescription for different values of N.

P O S S I B L E S O L U T I O N / S

Code that produced the dartboard plot.

```
N <- 100 # number of dart hits

# a uniform random sample of hits
hits.x <- runif( N, -1, 1 )
hits.y <- runif( N, -1, 1 )

inside  <- which( ( hits.x^2 + hits.y^2 ) <= 1 ) # hits inside the circle
outside <- setdiff( 1:N, inside )                # hits outside the circle

pdf( 'pi-dartboard.pdf', useDingbats = F )
 curve( sqrt( 1 - x * x ), from = -1, to = +1, ylim = c( -1, 1 ), asp = 1, bty = 'n',
        xaxt = 'n', yaxt = 'n', xlab = '', ylab = '', main = '', lwd = 2, col = 'gray' )
 curve( -sqrt( 1 - x * x ), from = -1, to = +1, ylim = c( -1, 1 ),
        add = T, lwd = 2, col = 'gray' )
 # alternative way of drawing a circle: symbols()

 abline( h = c( -1, 1 ), v = c( -1, 1 ), lwd = 2, col = 'gray' )
 abline( h = 0, v = 0, lty = 2, lwd = 2, col = 'gray' )

 points( hits.x[inside],  hits.y[inside],  pch = 20, col = 'darkgreen' )
 points( hits.x[outside], hits.y[outside], pch = 20, col = 'red' )
dev.off()
```

Basic dartboard simulation set-up for one estimate $\widehat{\pi}_N$ of π.

```
N <- 100 # number of dart hits

hit.x <- runif( N, -1, 1 ) # a uniform random sample of hits from the
hit.y <- runif( N, -1, 1 ) # square (-1,-1)-(-1,+1)-(+1,+1)-(+1,-1)

n.inside <- sum( ( hit.x^2 + hit.y^2 ) <= 1 ) # count of hits inside the circle
pi.hat   <- 4 * n.inside / N                  # one estimate of pi from these hits
```

A minimalistic variation. Any one quarter of the dartboard is enough to do this experiment. The top right quarter of the above dartboard is convenient to use:

```
N      <- 100                                                  # number of dart hits
pi.hat <- 4 * sum( ( runif( N )^2 + runif( N )^2 ) <= 1 ) / N # one estimate of pi
```

Replicating the basic dartboard simulation M times for a fixed N.

```
# returns one estimate of pi from N dart hits
estimate.pi <- function( N ) { 4 * sum( ( runif( N )^2 + runif( N )^2 ) <= 1 ) / N }

N      <- 10                                # number of dart hits
M      <- 1000                              # number of replications of the experiment
pi.hat <- replicate( M, estimate.pi( N ) ) # M estimates of pi
```

Making sense out of a large pile of numbers.

```
op <- par( mfrow = c( 2, 1 ) )
 # boxplot of estimates of pi
 boxplot( pi.hat, horizontal = T, axes = F ); axis( 3 )
 title( main = expression( hat( pi )[N] ), line = 3 )
 title( xlab = paste( 'N = ', N, ' | M = ', M, sep = '' ) )

 # pi.hat density histogram overlaid with appropriate normal density
 hist( pi.hat, breaks = 'FD', freq = F, main = '',
       xlab = expression( hat( pi )[N] ), col = 'gray', border = 'white' )
 curve( dnorm( x, mean = mean( pi.hat ), sd = sd( pi.hat ) ),
        from = min( pi.hat ), to = max( pi.hat ), col = 'red', add = T )
 abline( v = pi, col = 'blue', lty = 2 )
 axis( 3, at = pi, labels = expression( pi ), col = 'blue', col.axis = 'blue' )
par( op )
```

N-dependence of the $\widehat{\pi}_N$ distributions.

```
# number of replications of the dartboard experiment
M <- 1000

# how do the pi.hat distributions change with the number N of darts?
N <- c( 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000 )

# M estimates of pi for each N, resulting into a MXN matrix
pi.hat <- NULL
for ( n in N ) pi.hat <- cbind( pi.hat, replicate( M, estimate.pi( n ) ) )

# visualize pi.hat distributions as boxplots
boxplot( pi.hat, axes = F )
axis( 1, at = 1:ncol( pi.hat ), labels = N, las = 3 ); axis( 2 )
title( ylab = expression( hat( pi )[N] ), xlab = 'N' )
abline( h = pi, col = 'red' )
axis( 4, at = pi, labels = expression( pi ), col.axis = 'red', col = 'red', las = 1 )

# pi.hat distributions as histograms overlaid with appropriate normal densitites
op <- par( mfrow = c( 2, 5 ) )
 for ( i in 1:length( N ) )
  {
   hist( pi.hat[,i], breaks = 'FD', freq = F, col = 'gray', border = 'white',
         main = paste( 'N =', N[i] ), xlab = expression( hat( pi )[N] ) )
   curve( dnorm( x, mean = mean( pi.hat[,i] ), sd = sd( pi.hat[,i] ) ),
          from = min( pi.hat[,i] ), to = max( pi.hat[,i] ), col = 'red', add = T )
   abline( v = pi, col = 'blue', lty = 2 )
   axis( 3, at = pi, labels = expression( pi ), col = 'blue', col.axis = 'blue' )
  }
par( op )
```

How would you interpret and understand these results?

Scrambled eggs under gravity

P R O B L E M

Image courtesy: Batman

A hypothetical comic-book planet with normal Newtonian gravity happens to have an atmosphere consisting of two entities, e and g, in the 1:2 proportion. The two entities have the same mass, and behave as ideal gases, and do not interact with each other in any way. Create a two-dimensional snapshot of this atmosphere, where one of the dimensions is the direction of gravity (i.e., the vertical direction), and the other one is any direction that is perpendicular to the vertical.

A P P R O A C H / E S

The density of an ideal gas under gravity varies with the height y (measured from the surface upwards) as

$$\rho(y) = \rho_0 \exp\left(-\frac{mg}{k_B T} y\right),$$

where m is the mass of a molecule of the ideal gas, g is the gravitational acceleration at the surface, k_B is the Boltzmann constant, T is the temperature of the gas, and ρ_0 is the density at the ground level, $y = 0$. Let us assume that the e and g entities forming the atmosphere of this hypothetical comic-book planet do not interact with each other. Furthermore, let us assume that the motions of the two entities can be best described by the adjective *random*. Under these assumptions, we can say that the probability density function (PDF) for the entities constituting the comic-book atmosphere is $f(y) \propto \exp(-\lambda y)$, where we have defined $\lambda = mg/k_B T$. Apart from the normalization constant, this form is same as that for the PDF of the exponential distribution:

$$f(y) = \lambda \exp(-\lambda y), \quad y \geq 0,\ \lambda > 0.$$

Further, the two entities do not interact with each other, i.e., they are *independent*, which means that the above PDF describes both separately. So, to create a snapshot of the comic-book atmosphere, all that one needs to do is:

1. Choose the number N of the e entities, say $N = 50$. The number of the g entities is $2N$. Choose some value for the constant λ, say $\lambda = 1$.
2. Generate N exponential random numbers (`function rexp`) to represent the heights of the N e entities. Generate another $2N$ exponential random numbers to represent the heights of the $2N$ g entities.

3. Generate $3N$ uniform random numbers (`function runif`) to represent the horizontal coordinate of the $3N$ entities. Since gravity is assumed to be the uniform in the horizontal direction, we expect a uniform distribution of the two entities along this direction.
4. Make a scatterplot using symbols `e` and `g`. Any other way of representing the two entities will be equally good (or bad).
5. Try creating a similar 3D snapshot with x, y as the horizontal dimensions and z as the vertical. You will need to explore an `R` package that provides `functions` for making 3D scatterplots (e.g., `rgl`, `misc3d`, etc.).

POSSIBLE SOLUTION/S

```
N <- 50 # number of the e-entities

xlim <- c( 0, 1 )
ylim <- c( 0, 3 )

op <- par( mfrow = c( 1, 3 ), bg = 'gray40' )
 for ( lambda in 1:3 ) # parameter for the exponential distribution
  {
   plot.new(); plot.window( xlim = xlim, ylim = ylim, asp = 1 )

   abline( h = ylim[1], v = xlim, lty = 2 )

   title( ylab = 'Height from the surface',
          xlab = 'Horizontal coordinate',
          main = parse( text = paste( 'lambda ==', round( lambda, 3 ) ) ),
          col.main = 'green', col.lab = 'green' )

   # <<<
   for ( i in 1:N )
    {
     text( runif( 1 ), rexp( 1, lambda ), 'e',
           srt = runif( 1, 0, 360 ), cex = 1.5,  col = 'yellow' )
     text( runif( 1 ), rexp( 1, lambda ), 'g',
           srt = runif( 1, 0, 360 ), cex = 1.25, col = 'white' )
     text( runif( 1 ), rexp( 1, lambda ), 'g',
           srt = runif( 1, 0, 360 ), cex = 1.25, col = 'white' )
    } # need this explicit loop because srt does not take a vector of values. so sad.
   # >>>
  }
par( op )
```

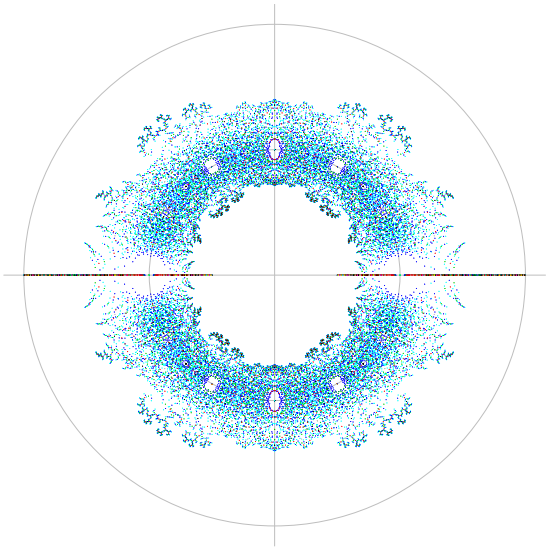
A small attempted twist in the above solution is giving the `e`s and the `g`s a random rotation. Unfortunately, this twist makes it necessary to use an explicit loop[2], which makes it slow. Without this twist, the inner `for` loop in the above solution could have been replaced with:

```
   # <<<
   points( runif( N ), rexp( N, lambda ), pch = 19, col = 'yellow' )
   points( runif( 2 * N ), rexp( 2 * N, lambda ), pch = 19, col = 'white' )
   # >>>
```

[2] Why?

Hol(e)y polynomial doughnut

P R O B L E M



What is plotted in this figure are the complex roots of all polynomials with degree between 1 and 11 and coefficients = ±1.

Polynomial roots have important roles to play in diverse domains such as filter design for signal processing (e.g., pole-zero plot), time series analysis (e.g., autoregressive models, unit root, etc.), theoretical computer science (see, e.g., this), statistical mechanics (e.g., Yang-Lee zeros), etc.

A degree-m polynomial $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$ has m zeros or roots, complex or real. Given any degree-m polynomial $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$, find all its zeros/roots. Using this, produce a plot similar to the one on the left.

A P P R O A C H / E S

The first part of the problem deals with polynomial zeros. The zeros/roots of a degree-m polynomial $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_mx^m$, i.e., solutions to the equation $p(x) = 0$, happen to be the eigenvalues of the $m \times m$ matrix

$$H_p = \begin{bmatrix} -\frac{a_{m-1}}{a_m} & -\frac{a_{m-2}}{a_m} & \cdots & -\frac{a_1}{a_m} & -\frac{a_0}{a_m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

To get all the roots, compute this matrix for the given polynomial, and diagonalize it using standard eigenvalue methods which are available on almost all computing platforms. This approach can be computationally expensive for large m, but allows computing all the roots including closely-spaced ones.

The second part of the problem, which is related to the above figure, deals with computing zeros of all polynomials with degree between 1 and some M, and coefficients = ±1. For a fixed degree m, the set of all polynomials with coefficients = ±1 has 2^{m+1} members. How does one generate all these 2^{m+1} possible coefficient vectors? We notice that 2^{m+1} is also the number of possible $(m+1)$-bit integers, and these integers contain all possible arrangements of 0s and 1s

across these $(m+1)$ bits. Therefore, taking the $(m+1)$-bit binary representations of all integers between 0 and $2^{m+1}-1$, and replacing all the 0s with -1s will give us all possible vectors with ± 1 entries. All that remains now is to repeat this for all degrees m between 1 and M, and plot the resulting zeros to create a plot similar to the one above.

WHY WOULD THIS APPROACH WORK?

Here is the rationale for the first part. Let us work this out for $m = 3$, i.e., $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$; the same argument can be extended to any m. The eigenvalue equation is

$$\begin{bmatrix} -\frac{a_2}{a_3} & -\frac{a_1}{a_3} & -\frac{a_0}{a_3} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = x \begin{bmatrix} t \\ u \\ v \end{bmatrix}$$

where x is an eigenvalue and $(t, u, v)^T$ is the corresponding eigenvector. The second and third rows tell us that $u = xv$ and $t = xu = x^2 v$. With these substitutions, the first row/equation becomes

$$-\frac{a_2}{a_3} x^2 v - \frac{a_1}{a_3} xv - \frac{a_0}{a_3} v = x^3 v$$

which implies

$$p(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0,$$

which is the condition for x to be a zero/root of the above polynomial.

POSSIBLE SOLUTION/S

Suppose that the polynomial is specified as an `R` vector `a` consisting of the coefficients `a[1]` $\equiv a_0$, `a[2]` $\equiv a_1$, ..., `a[m+1]` $\equiv a_m$. This convention for representing a polynomial is same as that for the in-built `function polyroot`. For example,

```
a <- c( 108^2, -108, -204, -1, 1 ) # p(x) = x^4 - x^3 - 204 x^2 - 108 x + 108^2
```

for which the zeros/roots are $9, 12, 10 \pm i\sqrt{8}$. To solve his problem using the method above, we need to compute the above matrix from `a`, and then use the built-in `function eigen` to get the eigenvalues of the H_p matrix:

```
# degree of the polynomial
m <- length( a ) - 1

# enforce assumptions
stopifnot( m > 0 )

# compute the H matrix
Hp <- matrix( 0, m, m )                 # m X m matrix filled with zeros
Hp[1,] <- - a[m:1] / a[m+1]             # first row
Hp[( col( Hp ) + 1 ) == row( Hp )] <- 1 # first sub-diagonal

# compute eigenvalues of H, i.e., roots of the polynomial
roots <- eigen( Hp, only.values = T )$values
```

Recognizing and encapsulating a generic element of computation for reuse. Computing all zeros/roots of a polynomial is a generic computation that may be required elsewhere. So this is a good candidate for a stand-alone `function` – although redundant, because `R` already provides the `function polyroot`:

```
polyroot.h <- function( a )
 {
  # this function computes all roots of the polynomial
  # a[1] + a[2] * x + ... + a[length( a )] * x^( length( a ) - 1 )
```

```
 m <- length( a ) - 1 # degree of the polynomial
 stopifnot( m > 0 )   # enforce assumptions

 # compute the Hp matrix
 Hp <- matrix( 0, m, m )                 # m X m matrix filled with zeros
 Hp[1,] <- - a[m:1] / a[m+1]             # first row
 Hp[( col( Hp ) + 1 ) == row( Hp )] <- 1 # first sub-diagonal

 # compute and return all eigenvalues of Hp, i.e., roots of the polynomial
 return( eigen( Hp, only.values = T )$values )
}
```

Recipe for a hol(e)y polynomial doughnut: `bits` and pieces.

```
bits1 <- function( x, length.out = 0 )
 {
  # Return the bits of a positive integer x, in the form of a vector.
  # Computed bit vector is zero-padded to match the given output length.

  # assumptions
  stopifnot( length( x ) == 1, x == as.integer( x ), x >= 0,
             length.out >= 0, length.out == as.integer( length.out ) )

  # trivial case bypassed
  if ( x == 0 ) return( rep( 0, max( length.out, 1 ) ) )

  # compute bits
  b <- NULL; while ( x ) { b <- c( b, x %% 2 ); x <- x %/% 2 }

  # b[1] is LSB, appropriate # of zeros padded beyond MSB
  return( c( b, rep( 0, max( 0, length.out - length( b ) ) ) ) )
 }
```

Here is another implementation that relies on bitwise operations:

```
bits <- function( x, length.out = 0 )
 {
  # Return the bits of a positive integer x, in the form of a vector.
  # Computed bit vector is zero-padded to match the given output length.
  # This implementation relies on R integers being signed 32-bit integers.

  # assumptions
  stopifnot( length( x ) == 1, x == as.integer( x ), x >= 0,
             length.out >= 0, length.out == as.integer( length.out ) )

  # trivial case bypassed
  if ( x == 0 ) return( rep( 0, max( length.out, 1 ) ) )

  # compute bits
  b   <- bitwAnd( x, as.integer( 2^( 0:30 ) ) ); b[b > 0] <- 1
  msb <- 32 - which( rev( b ) == 1 )[1] # most significant nonzero bit

  # b[1] is LSB, appropriate # of zeros padded beyond MSB
  return( b[1:min( 31, max( msb, length.out ) )] )
 }
```

After all this effort of writing our own `functions` to get the bit representation of an integer, it turns out that there is an in-built `function` called `intToBits` which does the same job[3]. In hindsight, this should not be surprising given that the operation under consideration is so basic. Here is how one might compare the performance of our two implementations and that of the in-built `intToBits`:

```
# Below, while is used be in place of for/sapply, because for/sapply
# require actual lists/vectors to iterate over, and 2^e storage can
# exhaust the RAM large for large enough e.

for ( e in c( 5, 10, 15, 20 ) ) # n = 0, ..., 2^e - 1
```

[3]Thanks to Atish for pointing this out.

```
{
 cat( 'e', e )

 t1 <- Sys.time()
  n <- 0; while ( n < 2^e ) { bits1( n ); n <- n + 1 }
 t2 <- Sys.time()
 dt <- t2 - t1
 cat( ' bits1', dt, attributes( dt )$units )

 t1 <- Sys.time()
  n <- 0; while ( n < 2^e ) { bits1( n ); n <- n + 1 }
 t2 <- Sys.time()
 dt <- t2 - t1
 cat( ' bits ', dt, attributes( dt )$units )

 t1 <- Sys.time()
  n <- 0; while ( n < 2^e ) { as.integer( intToBits( n ) ); n <- n + 1 } # in-built
 t2 <- Sys.time()
 dt <- t2 - t1
 cat( ' intTobits ', dt, attributes( dt )$units, '\n' )
}
```

Contrary to expectation, the performance of `bits` and `bits1` is not that different. Not surprisingly, the in-built `intToBits` is an order-of-magnitude faster than our two implementations. So much for excessive self-reliance.

Yet another route[4] to the sub-problem of generating all possible coefficient vectors with $+1, -1$ entries is to use `expand.grid`:

```
a <- expand.grid( rep( list( c( -1, 1 ) ), m + 1 ) ) # m-th degree polynomial
```

Each row of `a` is one of the 2^{m+1} coefficient vectors. While this approach might reduce the overall coding effort, it entails a (formidable) memory cost of storing a $2^{m+1} \times (m+1)$ array.

Complete recipe, finally. We need to plot complex roots of all polynomials with degree between 1 and 11 and coefficients $= \pm 1$. How does one do this? For degree m, generating all possible vectors of length $m+1$ with elements $= \pm 1$ is same as finding the binary representations of all integers from 0 to $2^{m+1} - 1$, and replacing 0s therein with -1s. This is where `function intToBits` comes handy. Roots are computed using the in-built `polyroot` instead of our own `polyroot.h` above.

```
m.min <- 13 # m.min < degree <= m.max; m.min > 0
m.max <- 13 # m.min < degree <= m.max; m.min > 0

# pdf( 'polynomial-roots.pdf', useDingbats = F )
plot.new()
op <- par( mar = rep( 0, 4 ) )

 plot.window( asp = 1, xlim = c( -2, +2 ), ylim = c( -2, +2 ) )
 abline( h = 0, v = 0, col = 'gray', lwd = 0.5 )
 symbols( rep( 0, 2 ), rep( 0, 2 ), circles = c( 1, 2 ),
          inches = F, fg = 'gray', lwd = 0.5, add = T )

 for ( m in m.max:m.min )
  {
   cat( m ) # have patience for large m

   t1 <- Sys.time()

   # <<<
   roots <- NULL
   for ( i in 0:( 2^( m + 1 ) - 1 ) ) # loop over all possible +-1 coefficients
    {
     # z <- bits( i, m + 1 ) # our own slower implementation
     z <- as.integer( intToBits( i )[1:( m + 1 )] ) # in-built, faster
```

[4] Thanks to Prajakta for pointing this out.

```
     z[z == 0] <- -1 # +-1 coefficient vector
     roots <- cbind( roots, polyroot( z ) )
    }
   # >>>

   t2 <- Sys.time()
   dt <- t2 - t1
   cat( '', dt, attributes( dt )$units, '\n' )

   points( Re( roots ), Im( roots ), pch = 19, cex = 0.01, col = m.max - m + 1 )
   # print( max( Mod( roots ) ) ) # --> 2 as m --> Inf?
  }

par( op )
# dev.off()
```

Explicit loops can be quite costly in R. In the above code, the inner loop can be internalized through a sapply call, as follows. Check for yourselves which one is faster.

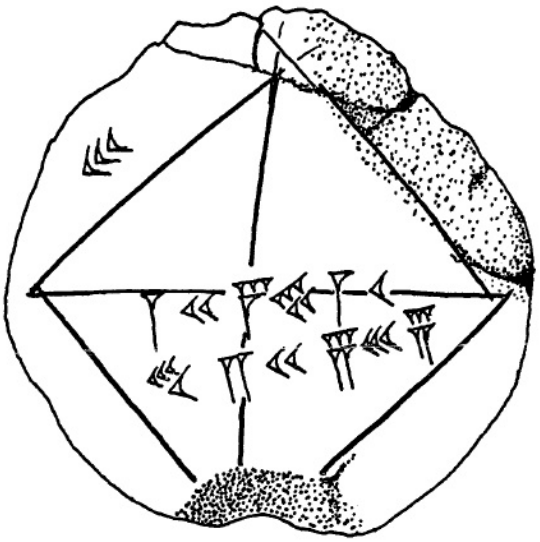
```
    # <<<
    roots <- sapply( 0:( 2^( m + 1 ) - 1 ),
                     function( i )
                      {
                       # z <- bits( i, m + 1 ) # our own slower implementation
                       z <- as.integer( intToBits( i )[1:( m + 1 )] ) # in-built, faster

                       z[z == 0] <- -1
                       return( polyroot( z ) )
                      }
                   )
    # >>>
```

Be it for or sapply, the price to pay is the storage for the vector `0:( 2^( m + 1 ) - 1 )`, which can exhaust all available memory for large enough m. This storage cost can be avoided with a while loop, bringing things back to the square one:

```
    # <<<
    roots <- NULL; i <- 0
    while ( i < 2^( m + 1 ) )
     {
      # z <- bits( i, m + 1 ) # our own slower implementation
      z <- as.integer( intToBits( i )[1:( m + 1 )] ) # in-built, faster
      z[z == 0] <- -1
      roots <- cbind ( roots , polyroot ( z ) )
      i <- i + 1
     }
    # >>>
```

Square root, the ancient way

P R O B L E M



Babylonian tablet YBC 7289, circa 1800-1600 BC, showing approximate value of $\sqrt{2}$. Image courtesy: MAA.

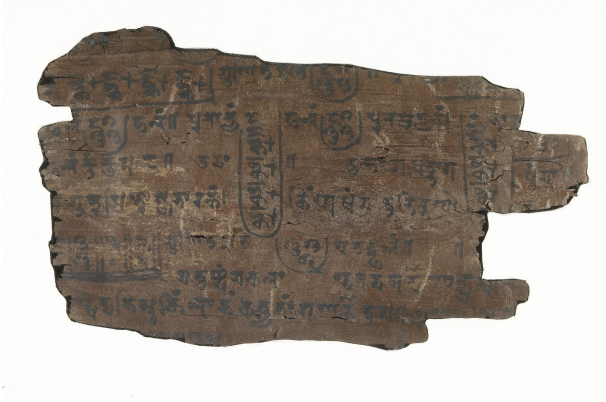
The Babylonian method for computing the square root of a positive real number S: Start with any arbitrary $x_0 > 0$, and iterate ($n = 0, 1, \ldots$)

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

This method is also known as the *divide-and-average* method.

An Indian method for computing the square root of a positive real number S: Start with any arbitrary $x_0 > 0$, and iterate ($n = 0, 1, \ldots$)

$$\begin{aligned} a_n &= \frac{S - x_n^2}{2x_n} \\ x_{n+1} &= x_n + a_n + \frac{1}{2}\frac{a_n^2}{x_n + a_n} \end{aligned}$$



A page from the Bakhshali manuscript. Date uncertain, but considered to be no later than 12th century. Figure 3 in this article shows another page that illustrates the calculation of square root. Image courtesy: Oxford University.

Let us remember that both methods are stated here in the modern mathematical language and

notation, and not the way the ancients might have chosen to express them. For both methods, successive iterates x_n get closer and closer to $\sqrt{S}$ – eventually, but thankfully, reasonably quickly. The second method involves more computation at every iteration, but converges at a much faster rate than the first (here is a proof) – that is, in absence of finite-precision arithmetic artifacts.

Some perspective should help here: The ancients probably used these methods using integer arithmetic and for hand computation, and their intent was probably to obtain what we would call rational approximations to square roots. In any case, they neither had the benefit of R, nor had to cope up with the quirks of the modern-day finite-precision arithmetic.

A P P R O A C H / E S

Both methods are fixed-point iteration methods which solve equations of the form $x = f(x)$.

The circled points in the figure on the right are solutions to the equation $x = f(x)$. They are also solutions of the equation $x - f(x) = 0$, i.e., zeros or roots of the function $g(x) = x - f(x)$.

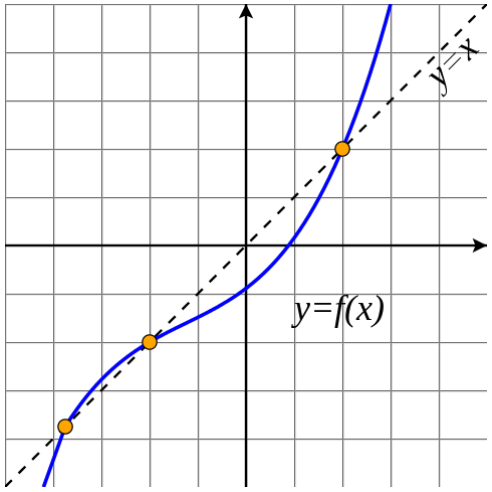


Image courtesy: Wikipedia

Given an initial point x_0, the method produces successive approximations $x_1, x_2, \ldots$ to the solution of this equation, where

$$x_i = f(x_{i-1}) \text{ for } i = 1, 2, \ldots.$$

Iteration is terminated when

$$|x_i - x_{i-1}| \leq \epsilon \quad \text{or} \quad i \geq N$$

for some pre-specified values ϵ and N.

For the Babylonian method, $f(x) = (x + S/x)/2$. For the Indian method, $f(x) = x + a(x) + a^2(x)/2(x + a(x))$ with $a(x) = (S - x^2)/2x$.

Algorithm 1 Fixed-point iteration to solve an equation of the form $x = f(x)$.

Require: function f, starting point x, maximum iterations $N \geq 0$, closeness threshold $\epsilon > 0$

1: $i \leftarrow 0$
2: **repeat**
3: $\quad i \leftarrow i + 1$
4: $\quad t \leftarrow x$
5: $\quad x \leftarrow f(x)$
6: **until** $i = N$ **or** $|x - t| \leq \epsilon$

W H Y W O U L D T H I S A P P R O A C H W O R K ?

The values of x that satisfy $x = f(x)$ are called fixed points of the iterative process $x \mapsto f(x)$. The notation $x \mapsto f(x)$ is a short-hand for the above iterative process; i.e., $x_1 = f(x_0), x_2 = f(x_1), \ldots$. For example, $x = 0, 1$ are the fixed points of the process $x \mapsto \sqrt{x}$. That is, $0 = \sqrt{0}$ and $1 = \sqrt{1}$. It is easy to see (e.g., with the help of a calculator, or R) that these two fixed

points of $x \mapsto \sqrt{x}$ have a very different character. 0 can be reached only if one starts at 0 – even a tiny deviation away from 0 drives the process away from 0. On the other hand, taking repeated square root of any positive number except 0 will eventually take it to 1. 1 is thus an *attracting/stable* fixed point, whereas 0 is a *repelling/unstable* fixed point. The above method finds, depending on the initial value x_0, one of the *attracting* fixed points of the process $x \mapsto f(x)$, if there is any. What decides the nature (attracting or repelling) of a fixed point is whether $|f'(x)|$ at the fixed point is larger than 1 (repelling) or less than 1 (attracting). The theory of iterated functions can be found in text books on chaos and nonlinear dynamics – E.g., Chaos: an introduction to dynamical systems by K.T. Alligood, T.D. Sauer, and J.A. Yorke.

The reason why either method converges to $\sqrt{S}$ is twofold: One, $x = f(x)$ is equivalent to $x^2 = S$. Two, $\sqrt{S}$ is the super-stable attracting fixed-point of the process $x \mapsto f(x)$ because $f'(\sqrt{S}) = 0$. Any starting value $x_0 > 0$ converges to $\sqrt{S}$ under this mapping. However, the rate of convergence depends on the initial value and the method.

Interestingly, if we use Newton's method to solve $x^2 - S = 0$, the form of Newton iteration turns out to be the same as the Babylonian method. The Indian method, on the other hand, is shown here to be equivalent to performing two consecutive iterations of the Newton method for the same problem.

(More generally, Newton's method can be thought of as fixed-point iteration for the process $x \mapsto x - f(x)/f'(x)$.)

P O S S I B L E S O L U T I O N / S

Some visualization first.

```
S <- 125348 # need square root of S

f.babylonian <- function( x, S ) { return( 0.5 * ( x + S / x ) ) }

f.indian <- function( x, S )
 {
  a <- function( x, S ) { 0.5 * ( S - x^2 ) / x }
  return( x + a( x, S ) + 0.5 * a( x, S )^2 / ( x + a( x, S ) ) )
 }

xlim <- c( 0.1, 2 ) * sqrt( S )
ylim <- c( 0, max( f.babylonian( xlim[1], S ), f.indian( xlim[1], S ) ) )

# pdf( 'square-root.pdf', useDingbats = F )
 op <- par( lwd = 2, bg = 'gray' )
  curve( f.babylonian( x, S ), from = xlim[1], to = xlim[2], n = 501,
         bty = 'n', ylim = ylim, ylab = 'y', col = 'darkgreen' )
  text( 1.2 * xlim[1], f.babylonian( xlim[1], S ),
        expression( y == ( x + S / x ) / 2 ),
        pos = 4, col = 'darkgreen', cex = 1.3 )

  curve( f.indian( x, S ), from = xlim[1], to = xlim[2], n = 501,
         bty = 'n', ylim = ylim, ylab = 'y', col = 'white', add = T )
  text( xlim[1], f.indian( xlim[1], S ),
        expression( y == x + a(x) + a^2 * (x) / 2 ( x + a(x) ) ),
        pos = 4, col = 'white', cex = 1.3 )

  abline( v = sqrt( S ), lty = 3 )
  axis( 3, at = sqrt( S ), label = expression( sqrt(S) ), cex.axis = 1.3 )

  abline( c( 0, 1 ), col = 'darkorange' )
  text( xlim[2], 1.1 * xlim[2], 'y = x', pos = 2, col = 'darkorange', cex = 1.3 )

  title( main = paste( 'S =', round( S, 10 ) ), line = 3 )
 par( op )
```

```
# dev.off()
```

Fixed-point iteration. Because fixed-point iteration has a utility outside of the present context, it is best implemented in the form of a standalone `function`. Below is a `C`-like implementation (along the lines of the `function` `newton` implemented elsewhere in this book).

```
fpi <- function( x0, f, ..., trace = F, eps = .Machine$double.eps, max.iter = 50 )
 {
  i    <- 0
  x    <- x0
  done <- ( i >= max.iter )

  x.trace <- if ( trace ) x else NULL

  while ( !done )
   {
    i <- i + 1
    t <- x
    x <- f( x, ... )

    done <- ( i >= max.iter ) || ( abs( t - x ) <= eps ) || !is.finite( x )

    if ( trace ) x.trace <- c( x.trace, x )
   }

  # attach additional information to the object being returned:
  attr( x, 'trace'   ) <- x.trace
  attr( x, 'success' ) <- ( abs( t - x ) <= eps ) & is.finite( x )

  return( x )
 }
```

Square root, the Babylonian way.

```
# square root of a number the Babylonian way
 S <- 336009 # need square root of S
 S <- 125348 # need square root of S

# the function defining fixed-point iteration
 f <- function( x, S ) { return( 0.5 * ( x + S / x ) ) }

# different starting values
 s1 <- fpi( 0.5 * ( 1 + S ),     f, S = S, trace = T )
 s2 <- fpi( 1,                   f, S = S, trace = T )
 s3 <- fpi( .Machine$double.eps, f, S = S, trace = T, max.iter = 1000 ) # slow

# examine the computed square roots and the traces to full precision
 oo <- options( digits = 16 )
  print( sqrt( S ) ); print( s1 ); print( s2 ); print( s3 )
 options( oo )

# tempting, but pointless, to create another function for the square root
#
# sqrt.babylonian <- function( x )
#  {
#   stopifnot( length( x ) == 1, x >= 0 )
#   if ( ( x == 0 ) || ( x == 1 ) ) return( x )
#   return( fpi( 0.5 * ( 1 + x ), function( t, u ) { 0.5 * ( t + u / t ) }, u = x ) )
#   # for faster convergence, the initial guess could perhaps be improved upon
#  }
```

Square root, the Indian way.

```
# square root of a number the Indian way
 S <- 336009 # need square root of S
 S <- 125348 # need square root of S

# the function defining fixed-point iteration
```

```
f <- function( x, S )
 {
  a <- 0.5 * ( S - x^2 ) / x
  return( x + a + 0.5 * a^2 / ( x + a ) )
 }

# different starting values
s1 <- fpi( 0.5 * ( 1 + S ),     f, S = S, trace = T )
s2 <- fpi( 1,                   f, S = S, trace = T )
s3 <- fpi( .Machine$double.eps, f, S = S, trace = T, max.iter = 1000 ) # slow

# examine the computed square roots and the traces to full precision
oo <- options( digits = 16 )
 print( sqrt( S ) ); print( s1 ); print( s2 ); print( s3 )
options( oo )
```

Convergence characteristics. A comparison of the convergence of the two methods (Babylonian, Indian) under identical conditions (i.e., same S, same starting guess, same termination criteria) might reveal occasional suprises despite formal proofs. The reason is that exact proofs rarely consider the rounding error which is omnipresent in finite-precision arithmetic. How the rounding behaviour behaves depends on the details of computation and implementation.

A functional approach. An elegant way of implementing fixed-point iteration is through the functional programming features in `R`; specifically, using the `function Reduce`. An example on the help page for `Reduce` shows how to implement a fixed number of iterations of the Babylonian method.

Graphics with granddaddy

P R O B L E M

Image courtesy: Wikipedia

"We might call [the Euclidean algorithm] the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day." – Donald Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 2nd edition (1981), p. 318.

This exercise is about two interesting graphics that can be produced with granddaddy: (A) coprime/relative prime pairs (a, b), and (B) the number of steps/iterations of the algorithm for each integer pair (a, b). In either case, a, b are both between 0 and some integer n. The resulting $(n+1) \times (n+1)$ matrices have interesting structure that can visualized in the form of color-coded images.

A P P R O A C H / E S

Granddaddy is too well-known and should not need any introduction. Here is the pseudocode for the division variant of the algorithm: Similar bare-basics implementations of the algorithm

Algorithm 2 Euclid's algorithm

```
1: function GCD(a, b)                          ▷ Compute gcd of a and b
2:     while b ≠ 0 do                          ▷ a is the gcd if b is 0
3:         r ← a mod b
4:         a ← b
5:         b ← r
6:     end while
7:     return a
8: end function
```

(such as the variants described here) are straightforward in R. Small tweaks are required to deal with end-cases ($a = 0$ or $b = 0$), assumptions (any integers or positive integers), and for counting steps to convergence (required for exercise (B) above). For the exercise (A) above, testing if two integers a, b are coprime/relative prime is same as checking whether the GCD of a, b is 1 or not.

Computation for either exercise has the structure of a double loop over a, b. The key to speed is accomplishing this part in R without the use of explicit loop structures. Visualize these two

matrices using an appropriate `R function` (find it!).

As an additional twist, you can compute the *coprime fraction*, i.e., the ratio of the number of coprimes in the set $\{(i,j), 0 \leq i,j \leq n\}$ to the size of this set, i.e.,

$$f(n) = \frac{\#\text{ of coprime pairs in the set } \{(i,j), 0 \leq i,j \leq n\}}{\text{size of the set } \{(i,j), 0 \leq i,j \leq n\}},$$

plot it as a function of n, and try to find information about its interpretations and deeper mathematical connections.

P O S S I B L E S O L U T I O N / S

Portraits of granddaddy.
The division version of Euclid's algorithm:

```
gcd.d <- function( a, b )
 {
  # GCD of two arbitrary integers:
  # division version of Euclid's algorithm
  # Algorithm A in Knuth, TAoCP-II (1981), p.320

  # assumptions
  stopifnot( length( a ) == 1, a == as.integer( a ),
             length( b ) == 1, b == as.integer( b ) )

  # bypass special cases (Knuth, TAoCP-II (1981), p.316)
  if ( b == 0 ) { attr( a, 'steps' ) <- 0; return( abs( a ) ) }
  if ( a == 0 ) { attr( b, 'steps' ) <- 0; return( abs( b ) ) }

  # see comments at the beginning of
  # Algorithm A in Knuth, TAoCP-II (1981), p.320
  a <- abs( a ); b <- abs( b )

  # count of steps to convergence
  i <- 0

  # the core Euclidian algorithm
  while ( b != 0 ) { r <- a %% b; a <- b; b <- r; i <- i + 1 }

  # attach additional information to the value being returned
  attr( a, 'steps' ) <- i

  return( a )
 }
```

The subtraction version of Euclid's algorithm:

```
gcd.s <- function( a, b )
 {
  # GCD of two integers:
  # subtraction version of Euclid's algorithm
  # https://en.wikipedia.org/wiki/Euclidean_algorithm#Implementations

  # assumptions
  stopifnot( length( a ) == 1, a == as.integer( a ),
             length( b ) == 1, b == as.integer( b ) )

  # bypass special cases (Knuth, TAoCP-II (1981), p.316)
  if ( b == 0 ) { attr( a, 'steps' ) <- 0; return( abs( a ) ) }
  if ( a == 0 ) { attr( b, 'steps' ) <- 0; return( abs( b ) ) }

  # see comments at the beginning of
  # Algorithm A in Knuth, TAoCP-II (1981), p.320
  a <- abs( a ); b <- abs( b )

  # count of steps to convergence
  i <- 0
```

```
 # subtraction version of the Euclidian algorithm
 while ( a != b ) { if ( a > b ) { a <- a - b } else { b <- b - a }; i <- i + 1 }

 # attach additional information to the value being returned
 attr( a, 'steps' ) <- i

 return( a )
}
```

The division version requires a much smaller number of iterations/steps to converge, whereas the subtraction version involves simpler computation at every iteration/step. More portraits of granddaddy (e.g., the extended Euclid algorithm) can be found in contributed R packages.

Visualizing coprimes.

```
# test if two integers are coprime
#
 is.coprime <- function( a, b, gcd = gcd.d ) { return( gcd( a, b ) == 1 ) }

# n below defines the integer lattice [0,n] X [0,n] that will be scanned
#
 n <- 11

# sapply() used below is much faster than an explicit loops.
# further performance improvement is possible if we use the
# fact that is.coprime(a,b) == is.coprime(b,a).
#
 coprimes <- sapply( 0:n,
                     function( a )
                      {
                       sapply( 0:n, function( b ) { is.coprime( a, b ) } )
                      }
                   )

# visualize
#
 image( 0:n, 0:n, coprimes, asp = 1, axes = F, col = c( 'white', 'black' ),
        xlab = 'a', ylab = 'b', main = 'Coprime Pairs' )
 axis( 1, at = pretty( 0:n ), line = 1 )
 axis( 2, at = pretty( 0:n ), line = 1 )

 dev.copy( pdf, 'coprime-pairs.pdf', useDingbats = FALSE )
 dev.off()
```

Visualizing steps to convergence.

```
# Count steps to convergence of Euclid's algorithm.
# Any of the two implementations can be supplied through the argument gcd.
#
 steps.gcd <- function( a, b, gcd = gcd.d ) { attr( gcd( a, b ), 'steps' ) }

# n below defines the integer lattice [0,n] X [0,n] that will be scanned
#
 n <- 397

# convergence behaviour of gcd.d() and gcd.s() can be different; try both.
#
 gcd <- gcd.s
 gcd <- gcd.d

# sapply() used below is much faster than an explicit loops.
# further performance improvement is possible if we use the
# fact that is.coprime(a,b) == is.coprime(b,a).
#
 steps <- sapply( 0:n,
                  function( a )
                   {
                    sapply( 0:n, function( b ) { steps.gcd( a, b, gcd ) } )
                   }
```

```
                )

# visualize
#
 image( 0:n, 0:n, steps, asp = 1, axes = F,
        xlab = 'x', ylab = 'y', main = '# of step to compute GCD(x,y)',
        col = terrain.colors( diff( range( steps ) ) + 1 ), xlim = c( 0, 1.1 * n ) )
 axis( 1, at = pretty( 0:n ), line = 1 )
 axis( 2, at = pretty( 0:n ), line = 1 )

 colorbar( c( 1.05 * n, 0 ), 0.05 * n, n, zrange = range( steps ),
           col = terrain.colors( diff( range( steps ) ) + 1 ) )
```

The overall structure of the above code is same as that for the code for producing an image of coprimes. The last expression adds a color legend to the plot. The x-range in the image is already adjusted (through argument `xlim`) to accommodate the color legend. The arcane and terse implementation of `colorbar` below relies on `rect` to draw rectangles:

```
colorbar <- function( bottomleft, width, height, col = rainbow( 101 ),
                      side = +1, zrange, horizontal = NULL, ... )
 {
  if ( is.null( horizontal ) ) horizontal <- ( width > height )

  n <- length( col )

  if ( !( side %in% c( -1, +1 ) ) ) side <- +1 # set membership; try help( '%in%' )

  if ( horizontal )
   {
    x <- seq( bottomleft[1], bottomleft[1] + width, length = n + 1 )
    y <- rep( bottomleft[2], n + 1 )
    side <- 2 + side
    rect( x[-( n + 1 )], y[-( n + 1 )], x[-1], y[-1] + height, col = col, border = F )
    axis( side, pos = bottomleft[2] + ( side == 3 ) * height,
          at = seq( bottomleft[1], bottomleft[1] + width, length = 5 ),
          labels = format( seq( zrange[1], zrange[2], length = 5 ), digits = 2 ), ... )

    return()
   }

  # vertical
  x <- rep( bottomleft[1], n + 1 )
  y <- seq( bottomleft[2], bottomleft[2] + height, length = n + 1 )

  side <- 3 + side

  rect( x[-( n + 1 )], y[-( n + 1 )], x[-1] + width, y[-1], col = col, border = F )

  axis( side, pos = bottomleft[1] + ( side == 4 ) * width,
        at = seq( bottomleft[2], bottomleft[2] + height, length = 5 ),
        labels = format( seq( zrange[1], zrange[2], length = 5 ), digits = 2 ), ... )
 }
```

The above is a classic example of something that is made to work in an hour of need, with the best of intent and effort, but left undocumented when that need was fulfilled. Notice the use of negative indexing of `vector`s `x` and `y` to imply exclusion. More `color pallets` are available in the `grDevices` package in the `R` standard library. Contributed packages provide additional pallets; see, for example, `function tim.colors` in the package `fields`.

Coprime fraction and π. If you compute the coprime fraction $f(n)$ defined earlier and plot it as a function of n,

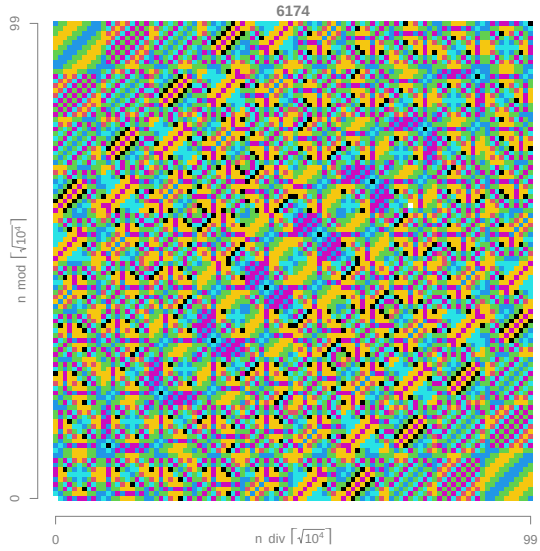
```
N <- 1:97 # scan integer lattices [0,n] X [0,n], where n takes values from N

coprime.fraction <- NULL
for ( n in N )
 {
  coprimes <- sapply( 0:n,
                      function( a )
```

```
                        {
                         sapply( 0:n, function( b ) { is.coprime( a, b ) } )
                        }
                    )
  coprime.fraction <- c( coprime.fraction, sum( coprimes ) / ( n + 1 )^2 )
 }

plot( N, coprime.fraction, type = 'b', bty = 'n', xlab = 'n',
      ylab = 'Coprime Fraction', pch = 21, bg = 'coral' )
abline( h = 6 / pi^2, lty = 2, col = 'red' )
axis( 4, at = 6 / pi^2, labels = expression( frac( 6, pi^2 ) ),
      las = 1, col = 'red', col.axis = 'red' )

dev.copy( pdf, 'coprime-fraction.pdf', useDingbats = FALSE )
dev.off()
```

then you might notice that it gets dangerously close to $6/\pi^2$ as n becomes larger and larger. This is no coincidence; see, e.g., this or this. On a different note, this $6/\pi^2$ limit could be used to concoct yet another procedure to *estimate* π. Happy exploration!

Kaprekar's 6174

P R O B L E M



Here, color represents the number of steps required for the Kaprekar routine to reach a fixed point (0 or 6174) starting from integers $n = 0, \ldots, 9999$. The axes are related to n, the base ($b = 10$), and the number of digits ($k = 4$). "a div b" represents the integer division of a by b, "a mod b" stands for the integer remainder after integer division of a by b, and $\lceil x \rceil$ means the smallest integer $\geq x$.

In 1949, D. R. Kaprekar discovered a procedure, now known as the Kaprekar routine, which quickly takes a 4-decimal-digit number (which has at least two distinct digits) to the number 6174 in at most 7 steps of his procedure. Applying this procedure to 6174 produces 6174. Numbers of the form *dddd* all go to 0 in one step.

One step of the Kaprekar routine goes this way: Choose any positive k-digit integer n_1. If n_1 has fewer than k digits, then take the leading digits to be 0 (i.e, consider 0s padded to n_1 where they don't matter). Two integers n_1' and n_1'' can be formed out of the k digits of n_1; namely, by sorting the digits in ascending and descending orders. To get the next number n_2 in this sequence, take the difference of n_1' and n_1''. That is, $n_2 = \max(n_1', n_1'') - \min(n_1', n_1'')$. Now apply the same routine to n_2 to get n_3, and so on. Any positive integer base other than 10 can also be used.

A variant of the routine omits the zero-padding step above. Depending on k and b, the two variants may show different behaviours.

One goal of this exercise is to produce a plot similar to the one above. Another goal is to computationally characterize the cyclic patterns produced by the Kaprekar routine. For example, 4-digits integers produce two distinct cyclic patterns: 0000 (integers of the form *dddd*) and 6174 (all other integers). Both these cycles have period = 1. Two-digit numbers produce two distinct cyclic patterns: period-1 cycle 00 (integers of the form *dd*), and the period-5 cycle 09, 81, 63, 27, 45 (all other integers). Fix the base b (say, to 10), and the number of digits k (say, to 4). Run the Kaprekar routine for each integer between 0 and $b^k - 1$. Assume that this routine produces, starting from any integer, a sequence of numbers that eventually rolls into a cyclic pattern. Find out how many distinct cyclic patterns are produced for the given combination of b and k.

D. R. Kaprekar, 1905-86. Image courtesy: Wikipedia

APPROACH / ES

Clearly, the three key computational elements of one Kaprekar step are: (a) disassemble an integer into a sequence of k digits with respect to some positive integer base $b > 1$; (b) assemble a sequence of k base-b digits into an integer; and (c) sort a sequence in ascending or descending order. The last element (c) is available in most programming languages as a canned routine: In R, this `function` is called `sort`. The Kaprekar routine repeats the Kaprekar step until a cycle is detected. Cycles can be of length $1, 2, \ldots$. Here are the relevant pseudocodes:

Algorithm 3 Compute the base-b digits of a positive integer n

function INT2DIGITS(n, b)
  Create an empty (i.e., length-0) vector d ▷ an R `vector`
  while $n \neq 0$ **do**
    $d \leftarrow$ APPEND(d, a mod b) ▷ append next digit at the end
    $n \leftarrow n$ div b ▷ div stands for integer division; `%/%` in R
  end while
  return d ▷ $d_1 :: b^0, d_2 :: b^1, \ldots$
end function

Algorithm 4 Compute a positive integer n given an array d of its the base-b digits

function DIGITS2INT(d, b)
  $n \leftarrow 0$
  for $i \leftarrow 1, \ldots,$ LENGTH(d) **do**
    $n \leftarrow n + d_i \times b^{i-1}$ ▷ $d_1 :: b^0, d_2 :: b^1, \ldots$
  end for
  return n
end function

Algorithm 5 Apply one Kaprekar step to integer n with at most k base-b digits

function KAPREKAR.STEP(n, b, k) ▷ b: base, k: # of digits
  $d \leftarrow$ SORT(INT2DIGITS(n, b)) ▷ assume LENGTH(d) $\leq k$; i.e., $0 \leq n < b^k$
  Pad 0s at the end of d so that LENGTH(d) $= k$ ▷ To pad or not to pad? Explore!
  return DIGITS2INT(d) − DIGITS2INT(REVERSE(d))
end function

POSSIBLE SOLUTION / S

The previous four pseudocodes translate to the following four `function`s:

```
int2digits <- function( n, base = 10 )
 {
  # stopifnot( length( base ) == 1, base == abs( as.integer( base ) ), base >= 2,
  #            length( n ) == 1, n == abs( as.integer( n ) ) )

  digits <- 0

  if ( n > 0 )
   {
    digits <- NULL
    while ( n )
     {
      digits <- c( digits, n %% base )
      n <- n %/% base
     }
```

Algorithm 6 Apply the Kaprekar routine to integer n with at most k base-b digits

1: **function** KAPREKAR.ROUTINE(n, b, k) ▷ b: base, k: # of digits
2: Create an empty (i.e., length-0) vector m ▷ an R vector
3: **while** n is not found in m **do** ▷ terminate when a cycle is detected
4: $m \leftarrow$ APPEND(m, n) ▷ append n at the end of m
5: $n \leftarrow$ KAPREKAR.STEP(n, b, k)
6: **end while**
7: $m \leftarrow$ APPEND(m, n) ▷ repeated value n for identifying periodic cycle and initial transient
8: **return** m ▷ additional useful information may also be returned
9: **end function**

```
  }

 return( digits )
}

digits2int <- function( digits, base = 10 )
 {
  # stopifnot( length( base ) == 1, base == abs( as.integer( base ) ), base >= 2,
  #            all( digits %in% 0:( base - 1 ) ) )

  if ( length( digits ) == 0 ) return( NA )

  return( sum( digits * base^( 0:( length( digits ) - 1 ) ) ) )
 }

kaprekar.step <- function( n, ndigits = 4, base = 10, pad = TRUE )
 {
  # One step of the Kaprekar routine
  # http://mathworld.wolfram.com/KaprekarRoutine.html

  # stopifnot( length( base ) == 1, base == abs( as.integer( base ) ), base >= 2,
  #            length( n ) == 1, n == abs( as.integer( n ) ) ) # sanity check

  n <- sort( int2digits( n, base = base ) )

  # stopifnot( length( n ) <= ndigits ) # sanity check

  if ( pad ) n <- c( rep( 0, max( 0, ndigits - length( n ) ) ), n )

  return( digits2int( n, base = base ) - digits2int( rev( n ), base = base ) )
 }

kaprekar.routine <- function( n, ... )
 {
  # The Kaprekar routine
  # http://mathworld.wolfram.com/KaprekarRoutine.html

  orbit <- NULL
  while ( !( n %in% orbit ) ) # terminate when a cycle is detected
   {
    orbit <- c( orbit, n )
    n  <- kaprekar.step( n, ... )
   }
  orbit <- c( orbit, n )
  i  <- which( n == orbit )[1]

  return( list( orbit = orbit, transient.length = i - 1, cycle.start = i,
                cycle.end = length( orbit ) - 1, cycle.period = length( orbit ) - i ) )
 }
```

Notice that the function `kaprekar.routine` returns an R list that contains not only the "orbit" of the given number n but also some additional information; namely, the start and end of the cyclic part in the "orbit", length of the initial transient part, and the cycle period. The plot at the beginning of this exercise requires, for each integer n, the length of the initial

transient. This can be done as follows:

```
base    <-    10 # base
ndigits <-     4 # number of digits
pad     <-  TRUE # pad 0s to ensure exactly ndigits digits?

# Kaprekar orbits for each of the integers, with respect
# to given base and with the above number of digits
#
 orbits <- t( sapply( 0:( base^ndigits - 1 ),
                      function( n ) { kaprekar.routine( n, ndigits, base, pad ) } ) )

 # is.matrix( orbits ) == TRUE. Try: dim( orbits ), colnames( orbits )
 #
 # orbit periods: orbits[,'cycle.period']
 # transient lengths: orbits[,'transient.length']
 # ...

# lengths of the initial transients, arranged in a peculier fashion
#
 n.rowcol   <- ceiling( sqrt( base^ndigits ) )
 transients <- matrix( nrow = n.rowcol, ncol = n.rowcol )
 for ( i in 0:( nrow( orbits ) - 1 ) )
  {
   j <- 1 + ( i %%  n.rowcol )
   k <- 1 + ( i %/% n.rowcol )
   transients[j,k] <- orbits[[i + 1,'transient.length']]
  }
```

This is how the plot was made:

```
pdf( 'kaprekar.pdf', useDingbats = F )
 op <- par( mar = c( 3, 3, 1, 1 ) )
  image( 0:( n.rowcol - 1 ), 0:( n.rowcol - 1 ), transients,
         # asp = 1, col = terrain.colors( length( unique( c( transients ) ) ) ),
         asp = 1, col = 0:7,
         axes = F, xlab = '', ylab = '' )

  axis( 1, at = c( 0, n.rowcol - 1 ), line = 1, col.axis = 'gray50', col = 'gray50' )
  axis( 2, at = c( 0, n.rowcol - 1 ), line = 1, col.axis = 'gray50', col = 'gray50' )

  title( main = '6174', line = 0.3, col.main = 'gray50' )

  title( xlab = parse( text = paste( 'n ~~', 'div', '~~ group(lceil, sqrt(',
                                     base, '^', ndigits, '), rceil)', sep = '' ) ),
         line = 2, col.lab = 'gray50', col = 'gray50' )

  title( ylab = parse( text = paste( 'n ~~', 'mod', '~~ group(lceil, sqrt(',
                                      base, '^', ndigits, '), rceil)', sep = '' ) ),
         line = 1.5, col.lab = 'gray50', col = 'gray50' )
 par( op )
dev.off()
```

Characterizing the cyclic orbits for given number of digits k requires some more work. First, to determine if two cycles are the same, one needs to arrange them in some standard form. Below, the convention used arranges all cycles to start with the smallest value in the cycle. This way, two cycles can be compared directly to determine if they are the same or not.

```
standard.cycle <- function( cycle )
 {
  # standardized cycle: rearranges a vector representing a cyclic pattern
  # of numbers so that it starts at the minimal value along the cycle

  i <- which( cycle == min( cycle ) )

  return( cycle[c( i:length( cycle ), seq( 1, i - 1, length = i - 1 ) )] )
 }

cycles <- apply( orbits, 1,
                 function( o )
                  {
                   standard.cycle( o$orbit[o$cycle.start:o$cycle.end] )
```

```
                }
            ) # extract the cycles from previously computed orbits
```

Finally, some more work gives us the unique cycles for a given number of digits k, together with the integers n that converge to that cycle.

```
# find the unique set of standardized cycles, together
# with and the integers that converge to that cycle
#
 is.in.list <- function( x, list )
  {
   if ( length( list ) == 0 ) return( FALSE )
   any( sapply( list, function( l ) { all( x == l ) } ) )
  }

 which.in.list <- function( x, list )
  {
   if ( length( list ) == 0 ) return( integer( 0 ) )
   which( sapply( list, function( l ) { all( x == l ) } ) )
  }

 cycles.unique <- list()
 number.groups <- list()

 k <- 0
 for ( i in 1:length( cycles ) )
  {
   j <- which.in.list( cycles[[i]], cycles.unique )
   if ( length( j ) == 0 )
    {
     k                    <- k + 1
     cycles.unique[[k]] <- cycles[[i]]
     number.groups[[k]] <- i - 1
    }
   else
    {
     number.groups[[j]] <- c( number.groups[[j]], i - 1 )
    }
  }
```

As $k = 4$ integers with at least two distinct digits go to the number 6174, all $k = 3$ integers with at least two distinct digits go to the number 495. Interestingly, with 0-padding, all $k = 2$ integers (except those of the form dd) get locked into the period-5 cycle $9, 81, 63, 27, 45, 9, \ldots$. More details can be found, e.g., here and here.

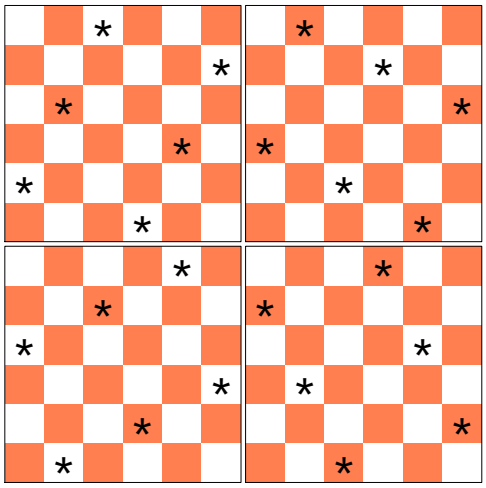
Performance considerations. Firstly, the above implementation stores *all* orbits. Sooner or later, i.e., for sufficiently large k, this will surely become unwieldy. Secondly, for base-10 integers, the same procedure can be implemented using `character` operations, which should make it faster:

```
kaprekar.step.10 <- function( n, ndigits = 4, pad = TRUE )
 {
  # Kaprekar step for base 10.

  # sanity check
  # stopifnot( length( n ) == 1, n == abs( as.integer( n ) ) )

  # convert to character, separate digits, sort them
  n <- sort( unlist( strsplit( as.character( n ), '' ) ) )

  # sanity check
  # stopifnot( length( n ) <= ndigits )

  # pad zeros
  if ( pad ) n <- c( rep( '0', max( 0, ndigits - length( n ) ) ), n )

  # reassemble integers and return their difference
  return(   as.integer( paste0( rev( n ), collapse = '' ) )
          - as.integer( paste0(      n,   collapse = '' ) ) )
```

```
}
```

Long live the Queens!

P R O B L E M



All four conflict-free arrangements of 6 queens on a 6 × 6 chessboard

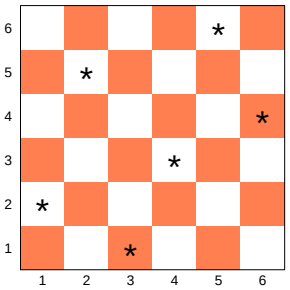
The problem of placing N queens on a $N \times N$ chessboard so that they do not threaten each other is the celebrated N-queens problem. For this problem, color of the queen does not matter; they are all identical, highly territorial, and equally powerful. Peace prevails in this world only when the N queens do not see one another at all.

Turing award winner Niklaus Wirth used this problem to illustrate a program design methodology called stepwise refinement in his 1995 article *Program Development by Stepwise Refinement*.

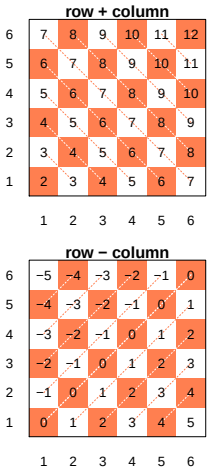
Try solving this problem in R using any approach, your own or from literature. There can be different flavours to this exercise: (A) find *all* the peaceful arrangements of N queens on a $N \times N$ chessboard, or (B) find *one* peaceful arrangement of queens as quickly as possible.

A P P R O A C H / E S

Representation. Recall that a chess queen exerts her influence along the vertical, the horizontal, and the two diagonal lines that cross at the her position on the chessboard. The vertical and horizontal constraints can be fulfilled by ensuring that there is exactly one queen in any row and in any column. Therefore, positions of the N queens can be specified through a vector π of size N, where π_i is the column index of a cell in the ith row where a queen is placed – That is, (i, π_i) are the (row,column) indices of the queen in the ith row. See figure on the right: With rows running horizontal and columns running vertical in the figure, this configuration is $\pi = (2, 5, 1, 3, 6, 4)$. This is a "hostile" placement, with three queen pairs in conflict: These are at cell locations $(1, 2)$ and $(5, 6)$, $(2, 5)$ and $(4, 3)$, and $(3, 1)$ and $(6, 4)$. Notice two things about this representation: (1) i can also be taken as the index of a queen: "ith queen" is same as "queen in the ith row". (2) Possible placements are permutations of $1, \ldots, N$ (hence $N!$ in number).



In these chessboard plots, "rows" are vertical and "columns" are horizontal.



Is a placement safe? How do we detect pairwise "collisions", "conflicts", or "hostilities" in a particular placement? We only need to detect diagonal collisions now, because vertical and horizontal collisions are already avoided as explained above. Consider queens i and j with coordinates (i, π_i) and (j, π_j) respectively. If they lie along the same anti-diagonal, then $i + \pi_i = j + \pi_j$, implying $i - j = -(\pi_i - \pi_j)$. If they lie along the same diagonal, then $i - \pi_i = \pm(j - \pi_j)$, implying $i - j = \pm(\pi_i - \pi_j)$. Both these conditions (i.e., diagonal or anti-diagonal collision) can be combined together as

$$|i - j| = |\pi_i - \pi_j|.$$

A placement is "safe" if it has no pairwise collisions. The following pseudocode expresses the essential logic of the safety check:

Algorithm 7 Check safety of a placement of N queens on a $N \times N$ chessboard

```
1:  function SAFE(p)                    ▷ p is a permutation of 1,...,N representing a placement of queens
2:      N ← LENGTH(p)
3:      for i = 1,...,N do
4:          for j = i+1,...,N do
5:              if |i − j| = |p_i − p_j| then
6:                  return false                                  ▷ Placement is unsafe
7:              end if
8:          end for
9:      end for
10:     return true                                               ▷ Placement is safe
11: end function
```

Approach 1: Brute-force search. Because different placements of N queens are permutations of $1, \ldots, N$, the brute-force solution to finding all safe placements is to scan all these $N!$ permutations one by one, and select those that have no collisions. An algorithm that generates permutations one-by-one in the lexicographic order is ideally suited for this approach.

Approach 2: Backtracking. Try solving this problem using R with the backtracking approach illustrated in Niklaus Wirth's *Program Development by Stepwise Refinement*.

Approach 3: Finding one safe placement – quickly. This paper presents a heuristic algorithm to find one safe placement in a fast fashion for very large N. This algorithm requires identifying the queens that are in conflict with one another. We already have an algorithm to check if a placement is safe. Some tweaking should lead to a way to identify queens in conflict.

POSSIBLE SOLUTION/S

Is a placement safe? Below is an R-friendly implementation of the safety algorithm. At the cost of additional but faster computation, this implementation internalizes the two costly explicit loops through `function outer`:

```
n.hostile <- function( p )
 {
  # returns number of hostile queens pairs

  n <- length( p ); stopifnot( setequal( p, 1:n ) ) # p should be a permutation of 1:n
```

```
  # 0.5 * for double counting over upper+lower triangle; - n for the diagonal.
  0.5 * ( sum( abs( outer( p, p, '-' ) ) == abs( outer( 1:n, 1:n, '-' ) ) ) - n )
 }

is.safe.placement <- function( p ) { return( n.hostile( p ) == 0 ) }
```

Approach 1: Brute-force search. Here is an R translation of a plain-English description of an algorithm to generate permutations one-by-one in the lexicographic order:

```
next.perm <- function( p )
 {
  # Given one permutation p of numbers 1:n, this function generates the next permutation
  # that is lexicographically the next permutation to p.
  # If p is the last permutation length( p ):1, then this function returns NULL.
  #
  # Source: http://en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order

  n <- length( p ); stopifnot( setequal( p, 1:n ) ) # p should be a permutation of 1:n

  # 1. Find the largest index k such that p[k] < p[k + 1].
  k <- which( p[-n] < p[-1] )
  # If no such index exists, the permutation is the last permutation.
  if ( length( k ) == 0 ) return( NULL )
  k <- max( k ) # largest index

  # 2. Find the largest index l greater than k such that p[k] < p[l].
  l <- k + max( which( p[-( 1:k )] > p[k] ) )

  # 3. Swap the value of p[k] with that of p[l].
  p[c( k, l )] <- p[c( l, k )]

  # 4. Reverse the sequence from p[k + 1] up to and including the final element p[n].
  return( c( p[1:k], rev( p[-( 1:k )] ) ) )
 }
```

Below is an example of usage of this function. In this example, all permutations are stored and returned – which is not a particularly happy prospect for large n:

```
all.perm <- function( n )
 {
  # this function returns all permutations of 1:n.
  # it is intended only for illustrative purposes and for sufficiently small n.
  # use with caution: n! grows very fast!

  all.perm <- NULL

  p  <- 1:n # start with the lexicographically-first permutation
  while ( !is.null( p ) )
   {
    all.perm <- rbind( all.perm, p ) # store it
    p  <- next.perm( p ) # get the next one in the lexicographic order
   }

  rownames( all.perm ) <- NULL

  return( all.perm ) # rows of this matrix are the individual permutations
 }
```

Fitting all the pieces together, here is the brute-force search for all safe placements of N queens on a $N \times N$ chessboard:

```
# n-queens problem: brute force search

next.safe.placement <- function( p )
 {
  if ( is.null( p ) ) return( p ) # nothing further to do

  n <- length( p ); stopifnot( setequal( p, 1:n ) )

  repeat
```

```
   {
    p <- next.perm( p )
    if ( is.null( p ) || is.safe.placement( p ) ) return( p )
   }

  return( NULL ) # no more permutations to be scanned
 }

all.safe.placements <- function( n )
 {
  # use with caution; function intended for sufficiently small n

  stopifnot( n == as.integer( n ), length( n ) == 1, n > 0 )

  if ( n == 1 ) return( 1 ) # trivial case

  p <- 1:n
  QP <- if ( is.safe.placement( p ) ) p else NULL

  while ( !is.null( p ) )
   {
    p  <- next.safe.placement( p )
    QP <- rbind( QP, p )
   }

  rownames( QP ) <- NULL

  return( QP ) # rows of QP are the safe placements
 }
```

Visualization. The first `function` below draws a $N \times N$ chessboard:

```
draw.chessboard <- function( n, axes = F, xlabels = letters,
                             col = c( 'coral', 'white' ), ... )
 {
  stopifnot( n == as.integer( n ), length( n ) == 1, n > 0 )

  { # << this extra pair of braces ensures that the if {} else {} block which ...
   if ( n %% 2 )
    {
     oo <- options( warn = -1 ) # suppress warnings; they are expected for odd n
      board <- matrix( rep( 0:1, ceiling( 0.5 * n^2 ) ), n, n )
     options( oo )
    }
   else
    board <- matrix( rep( c( rep( 0:1, 0.5 * n ), rep( 1:0, 0.5 * n ) ), 0.5 * n ),
                     nrow = n, ncol = n )
  } # ... is broken up across lines for readability is parsed correctly >>

  image( 1:n, 1:n, board, axes = F, col = col, xlab = '', ylab = '' )
  box()

  if ( axes )
   {
    axis( 1, at = 1:n, labels = if ( n <= length( xlabels ) ) xlabels[1:n] else 1:n,
          tick = F, ... )
    axis( 2, at = 1:n, tick = F, las = 1, ... )
   }
 }
```

The next one places N queens on it:

```
place.queens <- function( p, queen = '\U265B', ... ) # \U265B is Unicode for black queen
 {
  # assumption/s stated but not enforced:
  # draw.chessboard( n ) has been invoked before calling this function

  n <- length( p ); stopifnot( setequal( p, 1:n ) ) # p should be a permutation of 1:n

  text( 1:n, p, queen, ... )
 }
```