Scientific Programming assignment 2

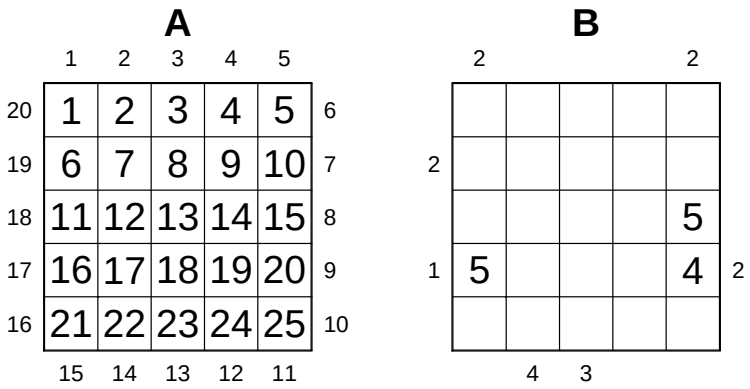
303034908

November 22, 2018

1 Skyscarper [10 marks]

The function `skyplot` was created by combining four functions. `edges.to.coord`, `plot.edges`, `soln.to.coord`, and `plot.solution`. These four functions create the coordinates to then plot using the `text()` function onto the existing plot.

```
skyplot <- function(edges, soln, ...){
  ## EDGES is a vector of length 4N containing the numbers clockwise
  ## around the grid, starting in the top left corner. SOLN is a
  ## vector of length N^2, storing on a row-by-row basis, starting in
  ## top-left corner. Any elements containing 0 are meant to indiciate blank.
  ## N could be between 4 and 9.
  axmax <- (length(edges)/4) #length of a side
  par(pty="s") #set aspect ratio as square
  plot(1, 1,  pch='.', xaxt='n', yaxt='n', xlab="", ylab="",
       xlim=c(0.17,(axmax-0.17)), ylim=c(0.17,(axmax-0.17)))
  title(... , line=2, cex.main = 2)#set the xlim and y lim to contact with the
  #vertical lines at 0 and axmax creating a slightly bolder outside
  abline(v=c(0:axmax), h=c(0:axmax)) #add horizonal and vertical lines to plot
  edges.coord <- edges.to.coord(edges) #create coordinates to plot edges numbers
  plot.edges(edges.coord) # plot edges numbers : black if non-0, white if 0.
  soln.coord <- soln.to.coord(soln)
  plot.solutions(soln.coord)
}



par(mfrow=c(1,2),mar=c(1,3,1,3))
skyplot( 1:20, 1:25, main="A")
skyplot( c(2,0,0,0,2, 0,0,0,2,0, 0,0,3,4,0, 0,1,0,2,0),
         c(0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,5, 5,0,0,0,4, 0,0,0,0,0), main="B")
```



Solving the skyplot can be attempted by brute force. Firstly, all possible skyplot solutions are created. Then each one is tested for correctness.

The possible solution is correct if there are no duplicated numbers in each row/column, and if the edges created by the possible solution match with the non-zero values of the edge values given.

The first check was implemented with the `check.duplicates` function, that took the sum of duplicated values of each row and column, and if the sum was 0, returned `TRUE`.

The second method of checking required three functions, two to create the edges numbers for a given solution and the third to check the match with the edge numbers of the correct solutions. As hinted by the names, `bigger.than.first` is a function that returns the number of "skyscrapers" seen from the beginning of the vector. `create.edge.numbers` uses `bigger.than.first` to recreate a 4N sized vector of edges which can then be tested and matched against the non-zeros values of the original vector. A correct match returns `TRUE`.

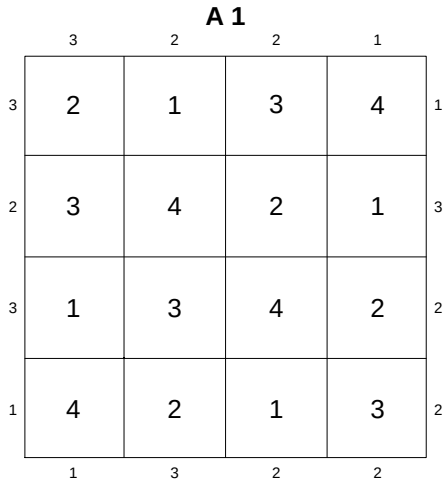
Finally, to create the possible solution matrices, the `permutation` and the `combn` functions are both used. As an example, `permutation(4)` returns matrix A with 4 columns and 24 rows. The aim is to select 4 rows from matrix A to create a possible solution. To index over the rows of matrix A, `combn(24,4)` is used to create a 4x10626 matrix B with all possible row combinations of the 4 from the 24 rows. Finally, as this combination of index does not take into account the order of values in each row of matrix A, a further `permutation(4)`, is run to index across all orders of columns for each row in matrix B.

```
#function3
#create all permutations and call functions 1 and 2 to test
#return list of all possible solutions

check.solutions <- function(soln.mat, edges){
  ### given a possible solution matrix and the correct vector of edges
  ### check if the solution matrix passes both the row/col duplicates test
  ### and the edges matching test
  if (check.duplicates(soln.mat) == TRUE &&
      check.edges(soln.mat, edges) == TRUE){
```

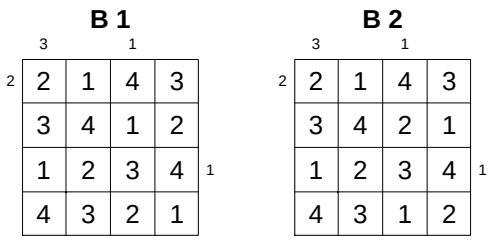
```
    return(TRUE)
  }else{
    return( FALSE)
  }
}

find.final.solutions <- function(edges){
  ### given the EDGES as vector, find all the possible skyplot solutions
  ### Output as list of matrices
  all.permutations <- permutations(4)
  possible.row.index <- combn(dim(all.permutations)[1],
                              dim(all.permutations)[2])
  possible.col.index <- permutations(4)
  final.soln <- vector("list", 1)
  n<-1
  for (ii in 1:(dim(possible.row.index)[2])){
    for (jj in 1:dim(possible.col.index)[1]){
      poss.soln <- all.permutations[c(possible.row.index
                                      [c(possible.col.index[jj,]),ii]), ]
      if (check.solutions(poss.soln, edges)){
        #print(poss.soln)
        final.soln[[n]] <-poss.soln
        n <- n+1
      }
    }
  }
  return(final.soln)
}
```

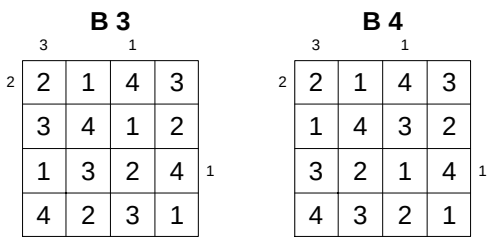
The possible solutions are then tested, and if positive, returned as a list of matrices. The figure below shows all the correct solutions for problem A and B.

```
e1 = scan(comment.char="#", quiet=T,
          "https://raw.githubusercontent.com/sje30/rpc2018/master/a2/e1.dat")
final.solutions.A <- find.final.solutions(e1)
#convert solutions from matrices to vectors
final.solutions.A <- lapply(final.solutions.A, function(x) as.vector(t(x)))
if (length(final.solutions.A) == 1 ){
  par(mfrow=c(1,1))
}else {
  par(mfrow=c(ceiling(length(final.solutions.A)/2), 2))
}
for (num in 1:length(final.solutions.A)){
  skyplot(e1, final.solutions.A[[num]], main=paste("A", num))
}
```




```
e2 = c(3,0,1,0, 0,0,1,0, 0,0,0,0, 0,0,0,2)
final.solutions.B <- find.final.solutions(e2)
final.solutions.B <- lapply(final.solutions.B, function(x) as.vector(t(x)))
#plot the solutions
par(mfrow=c(ceiling(length(final.solutions.A)/2), 2))
for (num in 1:length(final.solutions.B)){
  skyplot(e2, final.solutions.B[[num]], main=paste("B", num))
}
```





2 Roulette

2.1 Simulation

For each game a function (with no inputs) that plays the game once and returns a vector of length two consisting of the amount won/lost and how many bets were made was written. The functions can be seen below.

```
betting.on.red <- function(){
  wallet <- 1
  num.bets <- 0
  final.wallet <- 0
  bet.money <- 1
  while (num.bets < 1) {
    num.bets <- num.bets +1
    #result <- as.integer(runif(1, min=0, max=36))
    result <- sample(0:36, 1, replace=TRUE)
    if (1 <= result & result <= 18){
      wallet <- wallet + bet.money
    }else{
      wallet <- wallet - bet.money
    }
  }
  final.wallet <- wallet -1
  return(c(final.wallet,num.bets))
}


betting.on.number <- function(){
  wallet <- 1
  num.bets <- 0
  bet.money <- 1
  while (num.bets < 1){
    num.bets <- num.bets +1
    #chosen.number <- as.integer(runif(1, min=0, max=36))
    chosen.number <- sample(0:36, 1, replace=TRUE)
    #result <- as.integer(runif(1, min=0, max=36))
    result <- sample(0:36, 1, replace=TRUE)
    if (all.equal(chosen.number,result) == TRUE){
      wallet <- wallet + 35
    }else {
      wallet <- wallet - bet.money
    }
  }
  final.wallet <- wallet -1
  return(c(final.wallet, num.bets))
}


martingale.system <- function(){
  wallet <- 1
  bet.money <- 1
```

```
  num.bets <- 0
  final.wallet <- 0
  while (wallet < 10 & bet.money <= 100){
    num.bets <- num.bets + 1
    #result <- as.integer(runif(1, min=0, max=36))
    result <- sample(0:36, 1, replace=TRUE)
    if (1 <= result & result <= 18 ){
      wallet <- wallet + bet.money
      bet.money <- 1
    }else {
      wallet <- wallet - bet.money
      bet.money <- bet.money*2
    }
  }
  final.wallet <- wallet - 1
  return(c(final.wallet, num.bets))
}


D.labouchere.system <- function(){
  wallet <- 1
  numbers <- c(1,2,3,4)
  bet.money <- sum(head(numbers, n=1), tail(numbers,n=1))
  num.bets <- 0

  while (length(numbers) >= 1 & bet.money <= 100 ){
    num.bets <- num.bets + 1
    if (length(numbers) >= 2) {
      bet.money <- sum(head(numbers, n=1), tail(numbers,n=1))
      #result <- as.integer(runif(1, min=0, max=36))
      result <- sample(0:36, 1, replace=TRUE)
      if (1 <= result & result <= 18){
        wallet <- wallet + bet.money
        numbers <- numbers[c(-1, -length(numbers))]
      }else{
        wallet <- wallet - bet.money
        numbers <- c(numbers, bet.money)
      }
    }else if (length(numbers) == 1){
      bet.money <- numbers
      result <- runif(1, min=0, max=36)
      if (1 <= result & result <= 18){
        wallet <- wallet + bet.money
        numbers <- NULL
      }else{
        wallet <- wallet - bet.money
        numbers <- c(numbers, bet.money)
      }
    }
  }
  final.wallet <- wallet - 1
  return(c(final.wallet, num.bets))
```

```
}
```

The function `a.night.to.remember` was written to iterate over each of the above games 100,000 times. The results were analysed using four more functions, `expected.winnings`, `proportion.won`, `expectd.play.time` to estimate the expected winnings per game, the proportion of games won and the expected playing time and `analyse.night` to combine these three into one.

```
a.night.to.remember <- function(itn=100000, type){
  ### iterate by itn over each type of game
  switch(type,
         play.on.red = replicate(itn, betting.on.red()),
         play.on.number = replicate(itn, betting.on.number()),
         play.martingale = replicate(itn, martingale.system()),
         play.labouchere = replicate(itn, D.labouchere.system())
  )
}
```

```
expected.winnings <- function(night.log){
  final.wallet <-  night.log[1, ]
  mean.final.wallet <- mean(final.wallet)
  sd.final.wallet <- sd(final.wallet)
  return(c(mean.final.wallet, sd.final.wallet))
}

proportion.won <- function(night.log){
  mean.prop.won <- mean(night.log[1,]>0)*100
  sd.prop.won <- sd(night.log[1,]>0)
  return(c(mean.prop.won, sd.prop.won))
}

expected.play.time <- function(night.log){
  games.played <- night.log[2,]
  mean.games.played <- mean(games.played)
  sd.games.played <- sd(games.played)
  return(c(mean.games.played, sd.games.played))
}

analyse.night <- function(night.log){
  exp.winnings <- expected.winnings(night.log)
  prop.games.won <- proportion.won(night.log)
  exp.playing.time <- expected.play.time(night.log)
  return(c(exp.winnings, prop.games.won,exp.playing.time))
}
```

2.2 Verification

The exact answers for expected winnings and proportion of games won can be calculated for game A (betting on red) and game B (betting on a number) according to the following intuitions:

Expected winnings = P(win) × amount won + P(loss) × amount lost

Proportion won = P(win) × 100

For game A this becomes:

$$\frac{18}{37} \times 1 - \frac{19}{37} \times 1 = -\frac{1}{37} \tag{1}$$

$$\frac{18}{37} \times 100 \sim 48.65\% \tag{2}$$

And for game B this becomes:

$$\frac{1}{37} \times 35 - \frac{36}{37} \times 1 = -\frac{1}{37} \tag{3}$$

$$\frac{1}{37} \times 100 \sim 2.702\% \tag{4}$$

The percentage error in the estimates are shown in table 1.

```
x1 <- xtable(df, caption="Verification of games A and B", label="tab:verification1" )
print(x1, scalebox = 1, include.rownames = FALSE)
```

Games	% Error expected winnings	% Error proportion won
A	8.61	0.00
B	81.65	0.02

Table 1: Verification of games A and B

The maximum amount that can be won and the maximum amount that can be lost can both be calculated exactly for each game.

- Game A: Betting on red
  - Max win: \$2. As a single iteration game, the max won will be \$1 + \$1.
  - Max loss: \$1, the original bet.
- Game B: Betting on a number
  - Max win: \$36, the \$1 + the \$35 won if the number is matched.
  - Max loss: \$1, the original bet.
- Game C: Martingale System
  - Max win: \$10. If the player always wins, the bet remains at \$1, which is gradually added until the limit of \$10 is reached.
  - Max loss: \$127. Consecutive losses leads to doubling of the bet until it reaches \$100. This will occur exactly at $\log_2 100 \sim 6.64$ losses, which however, has to be rounded to 6. Moreover, when loosing, the bet is always subtracted from the current amount of money. Therefore, the max lost will be $1 - 2^1 - 2^2 - 2^3 - 2^4 - 2^5 - 2^6 = -125$.
- Game D: D. Labouchere System
  - Max win: \$10 in this case, or, more generally, the sum of the numbers in the original list. A simple proof of this can be found at https://math.stackexchange.com/q/2470522.
  - Max loss: \$4940. A continuous loss from the beginning will lead to the bet size increasing gradually by 1 from its starting point, 5 in this case, until it reaches 100. This gradual loss of money till the limit can be denoted as: $\sum_{n=5}^{99} n = 4940$.

2.3 Variation

The simulation experiment of the first section was repeated five times and the maximum and minumum values of the expected winings, proportion of wins and expected play time are shown in table 2 below.

```
x2 <- xtable(full.five.results.df, caption="Min and Max across 5x100,000 iterations",
       label="tab:five.min.max")
print(x2, scalebox = 1, include.rownames = FALSE)
```

Games	Exp. winnigs min-max	Prop. wins min-max	Exp. playtime min-max
A	-0.029 , -0.0221	48.552 , 48.896	1 , 1
B	-0.0338 , 1e-04	2.684 , 2.778	1 , 1
C	-1.8299 , -1.6643	91.797 , 91.924	17.6294 , 17.676
D	-4.1389 , -3.4259	95.608 , 95.766	8.7942 , 8.879

Table 2: Min and Max across 5x100,000 iterations

Finally, the mean and standard deviation for 100,000 iterations of each game can be measured. These results are shown in table 3.

```
x3 <- xtable(final.table.df, caption="Mean and standard deviation for each
       game of 100,000 iterations", label="tab:finaltable" )
print( x3, scalebox = 1, include.rownames = FALSE )
```

Games	Winnings mean, std dev	Prop. wins mean, std dev	Play time mean, std dev
A	-0.0247 , 0.9997	48.765 , 0.4998	1 , 0
B	-0.005 , 5.9018	2.764 , 0.1639	1 , 0
C	-1.7309 , 36.0897	91.874 , 0.2732	17.6537 , 4.1607
D	-3.7512 , 73.8996	95.701 , 0.2028	8.8287 , 7.7178

Table 3: Mean and standard deviation for each game of 100,000 iterations