200206758-MAS6024

Part 1

Create a 8*8 matrix to store the letters in the letter grid.

```
to_print <- F
lgrid <- matrix(NA, nrow = 8, ncol = 8)
lgrid[1,] <- c("r", "l", "q", "s", "t", "z", "c", "a")
lgrid[2,] <- c("i", "v", "d", "z", "h", "l", "t", "p")
lgrid[3,] <- c("u", "r", "o", "y", "w", "c", "a", "c")
lgrid[4,] <- c("x", "r", "f", "n", "d", "p", "g", "v")
lgrid[5,] <- c("h", "j", "f", "f", "k", "h", "g", "m")
lgrid[6,] <- c("k", "y", "e", "x", "x", "g", "k", "i")
lgrid[7,] <- c("l", "q", "e", "q", "f", "u", "e", "b")
lgrid[8,] <- c("l", "s", "d", "h", "i", "k", "y", "n")
```

Calculates the frequency for each letter in the grid and stores it in letter_frequency

```
library("Rfast")
```

```
## Loading required package: Rcpp
```

```
## Loading required package: RcppZiggurat
```

```
centre_elements <- as.vector(t(lgrid))
letter_frequency <- c()
for(ind in 1:length(centre_elements)) {
  letter_frequency <- c(letter_frequency, count_value(centre_elements, centre_elements[ind]))
}
```

Neighbor grid

Creates a neighborgrid matrix which stores all the neighbors to the North(N), NorthEast(NE), East(E), SouthEast(SE), South(S), SouthWest(SW), West(W), NorthWest(NW) of the CENTRE square. It also stores the frequency for each letters on the grid. It also adds columns to the neighborgrid matrix i.e. A1, A2, A3 .... H7, H8 In case of an edge element, the adjacent squares to it will save the value as 'NA'

```
n = 8
lgrid.pad = rbind(NA, cbind(NA, lgrid, NA), NA)
ind = 2: (n+1)
neighborgrid = rbind(
```

	A1	A2	A3	A4	A5	A6	A7	A8	B1	B2	B3	B4	B5	B6	B7
CENTRE	r	l	q	s	t	z	c	a	i	v	d	z	h	l	t
N	NA	r	i	u	x	h	k	l	NA	l	v	r	r	j	y
NE	NA	l	v	r	r	j	y	q	NA	q	d	o	f	f	e
E	l	v	r	r	j	y	q	s	q	d	o	f	f	e	e
SE	v	r	r	j	y	q	s	NA	d	o	f	f	e	e	d
S	i	u	x	h	k	l	l	NA	v	r	r	j	y	q	s
SW	NA	NA	NA	NA	NA	NA	NA	NA	i	u	x	h	k	l	l
W	NA	NA	NA	NA	NA	NA	NA	NA	r	i	u	x	h	k	l
NW	NA	NA	NA	NA	NA	NA	NA	NA	NA	r	i	u	x	h	k
FREQUENCY	3	4	3	2	2	2	3	2	3	2	3	2	4	4	2

Figure 1: Neighbor grid

```
  CENTRE = centre_elements,
  N  = as.vector(lgrid.pad[ind - 1, ind    ]),
  NE = as.vector(lgrid.pad[ind - 1, ind + 1]),
  E  = as.vector(lgrid.pad[ind    , ind + 1]),
  SE = as.vector(lgrid.pad[ind + 1, ind + 1]),
  S  = as.vector(lgrid.pad[ind + 1, ind    ]),
  SW = as.vector(lgrid.pad[ind + 1, ind - 1]),
  W  = as.vector(lgrid.pad[ind    , ind - 1]),
  NW = as.vector(lgrid.pad[ind - 1, ind - 1]),
  FREQUENCY = letter_frequency
)

colnames(neighborgrid) <- paste0(rep(LETTERS[1:8], each=8), rep(1:8, 8))
```

Random position

Samples a random position on the grid

```
fetch_current_square <- function() {
  return(paste0(sample(LETTERS[1:8], size = 1), sample(c(1:8), size = 1)))

}
```

Edge element check

Returns true if the current square is on edge of the board else returns false if on white square

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter()     masks stats::filter()
## x purrr::is_integer() masks Rfast::is_integer()
## x dplyr::lag()        masks stats::lag()
## x dplyr::nth()        masks Rfast::nth()
## x purrr::transpose()  masks Rfast::transpose()
```

```
check_if_border_element <- function(current_square) {
  if(anyNA(neighborgrid[, current_square])) {
    return (TRUE);
  } else {
    return (FALSE);
  }
}
```

Fetch next move on grid

Checks if players token is on the edge of the board, if yes, then move at random to one of the 64 squares. If player is not on the edge of the board, it derives a random direction adjacent to the current square by using sample method on the 8 directions. Switch case is used to fetch the new position on the grid. For example, the method return the new position as string "D5"

```
next_move <- function(current_square) {

  if(to_print == T) cat("Current square: ", current_square, "\n")
  if(check_if_border_element(current_square)) {
    current_square <- fetch_current_square()
    next_move(current_square)

  } else {

    direction <- sample(c("N", "NE", "E", "SE", "W", "NW", "SW", "S"), size = 1)
    x_cord <- str_sub(current_square, 1, 1)
    y_cord <- str_sub(current_square, 2, 2)

    new_current_square = switch(
      direction,
      "N"= paste0(LETTERS[match(x_cord, LETTERS) - 1] , as.numeric(y_cord)),
      "NE"= paste0(LETTERS[match(x_cord, LETTERS) - 1] , as.numeric(y_cord) + 1),
      "E"= paste0(x_cord, as.numeric(y_cord) + 1),
      "SE"= paste0(LETTERS[match(x_cord, LETTERS) + 1] , as.numeric(y_cord) + 1),
      "W"= paste0(x_cord, as.numeric(y_cord) - 1),
      "NW"= paste0(LETTERS[match(x_cord, LETTERS) - 1] , as.numeric(y_cord) - 1),
      "SW"= paste0(LETTERS[match(x_cord, LETTERS) + 1] , as.numeric(y_cord) - 1),
      "S"= paste0(LETTERS[match(x_cord, LETTERS) + 1] , as.numeric(y_cord))
    )
    if(to_print == T) cat("Direction: ", direction, "\t",
                          "New current square: ", new_current_square, "\n")
    return(new_current_square)
  }

}
```

Part 2

The strategy is as follows:

- Rule 1: The neighborgrid stores the frequency for each characters on the board. Initial 3 characters chosen to be added to the collection should have the frequency 3 or 4.
- Rule 2: The fourth character added to the collection should always be from the existing letters present in the collection.
- Rule 3: While adding the fifth character to the collection, if collection_copy *(explained below)* is empty then any character is added to the collection, else only the values present in collection copy are added to the collection to form a palindrome.
- A *collection_copy* is used to identify the pairs in a palindrome. Whenever a character is added in the collection, it gets added to the collection_copy as well, but in case a duplicate character is being added to the copy, instead of adding it to the copy, it removes it. Hence, while adding the fifth character to the palindrome collection, it should always be from the collection_copy. It's use is to basically identify the pairs because it only contains the character which would be required to complete the palindrome, in case collection_copy is empty any character can be chosen.
- For example, if characters are added in following sequence, the collection and collection_copy are updated as follows:

Sequence	Collection	Collection_copy
1. a	a	a
2. b	a b	a b
3. a	a b a	b
4. c	a b a c	**b c**
5. c	a b a c c	

- While adding the 5th element to collection it should always be from collection_copy(highlighted above) at sequence 4 i.e. "b" or "c"

Green square check

Initialize the 4 green squares on the board to a vector

1. **is_green_square()**: Checks if the current token is on green square, if yes, returns true else returns false
2. **execute_event_if_green_square()**: This function handles the two events in case if the current token is on green square.

- *Event A*: If event A occurs, the collection is replaced with letters f,f,h and k
- *Event B*: If event B occurs, the letter is removed from the collection. The collection_copy is also updated

```
green_square <- c("B6", "C7", "F2", "G3")

is_green_square <- function(current_square) {
```

```
  if(current_square  %in% green_square) {
    return (TRUE)
  } else {
    return (FALSE)
  }
}

execute_event_if_green_square <- function(current_square_value, p,
                                          collection, collection_copy) {
  event = sample(LETTERS[1:2], size= 1, prob= c(p,1-p))

  switch(
    event,
    "A" = {
      collection <- c("f", "f", "h", "k")
      collection_copy <- c("h", "k")
    },
    "B" = {
      #removes the existing current square values from the collection
      collection <- collection[!collection %in% current_square_value]

      # if the value if already present in copy remove it, else add the element
      if(current_square_value %in% collection_copy) {
        collection_copy <- collection_copy[collection_copy != current_square_value]
      } else {
        collection_copy <- c(collection_copy, current_square_value)
      }
    }
  )
  if(to_print == T) cat("Green square Event: ", event, "\t", "Collection: ",
                        collection, "\t", "Collection copy: ", collection_copy)

}
```

Part 3

Count number of moves required to form the palindrome

The function returns the number of moves required to form a palindrome

```
#fetch non edge element if not provided by user
count_num_moves <- function(current_square, prob, to_print) {
  move_counter <- 0
  is_palindrome <- FALSE
  collection <- c()
  collection_copy <- c()

  while(is_palindrome == FALSE) {

    if(to_print == T) cat("Current square: ", current_square, "\t",
                          "Value: ", neighborgrid["CENTRE", current_square],"\n")
```

```
freq <- neighborgrid["FREQUENCY", current_square]
value <- neighborgrid["CENTRE", current_square]

if(is_green_square(current_square)) {

  execute_event_if_green_square(neighborgrid["CENTRE", current_square],
                                prob, collection, collection_copy)

} else if(length(collection) < 3){

  if(freq == 4 || freq == 3) {
    collection <- c(collection, value)

    # if the value if already present in copy remove it, else add the element
    if(value %in% collection_copy) {
      collection_copy <- collection_copy[collection_copy != value]
    } else {
      collection_copy <- c(collection_copy, value)
    }

  }
} else if(length(collection) >= 3 && length(collection) <5 ) {

  #while adding the 4th element the it should add only those elements
  #present in the collection
  if(length(collection) == 3 && value %in% collection) {
    collection <- c(collection, value)

    if(value %in% collection_copy) {
      collection_copy <- collection_copy[collection_copy != value]
    } else {
      collection_copy <- c(collection_copy, value)
    }

  }

  #while adding the fifth character
  else if(length(collection) == 4) {

    #if copy is null add any character to the collection to form a palindrome
    if(length(collection_copy) == 0) {
      collection <- c(collection, value)

    } else if(value %in% collection_copy) {
      collection <- c(collection, value)
    }
  }
}
if(to_print == T) cat("Collection: ", collection, "\t",
                      "Collection copy: ", collection_copy, "\n")
move_counter = move_counter + 1
is_palindrome <- if(length(collection) == 5) ? break else FALSE
```

```
    current_square <- next_move(current_square)
  }

  if(to_print == T) cat("Total moves: ", move_counter, "\n")
  if(to_print == T) cat("Palindrome: ", collection, "\n")
  return(move_counter)
}
```

Part 4

As seen the average number of moves remains fairly constant for varying probabilities. This could be that the code is optimized in a way that it is not creating any bias for the conditions that depend on probabilities.

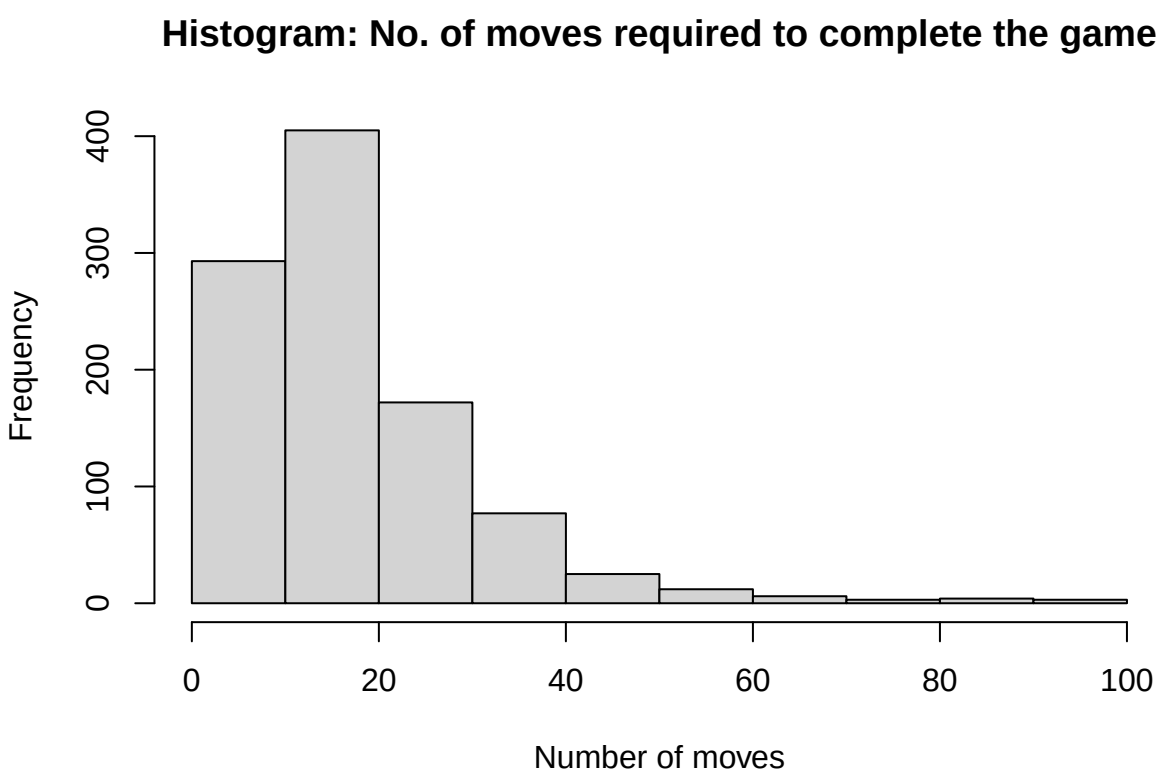
```
probabilities <- c(0.10, 0.30, 0.50, 0.75, 0.90)
average_moves <- c()
for(ind in 1:length(probabilities)) {
  move_count <- replicate(1000, count_num_moves("D4", probabilities[ind], to_print = F))
  average_moves <- c(average_moves, mean(move_count))
  cat("Average moves required for probability ", probabilities[ind], " :",
      mean(move_count), "\n")

}
```

```
## Average moves required for probability  0.1  : 19.327
## Average moves required for probability  0.3  : 18.541
## Average moves required for probability  0.5  : 18.426
## Average moves required for probability  0.75  : 18.821
## Average moves required for probability  0.9  : 18.567
```

Part 5

The average number of moves required for both the cases is fairly similar

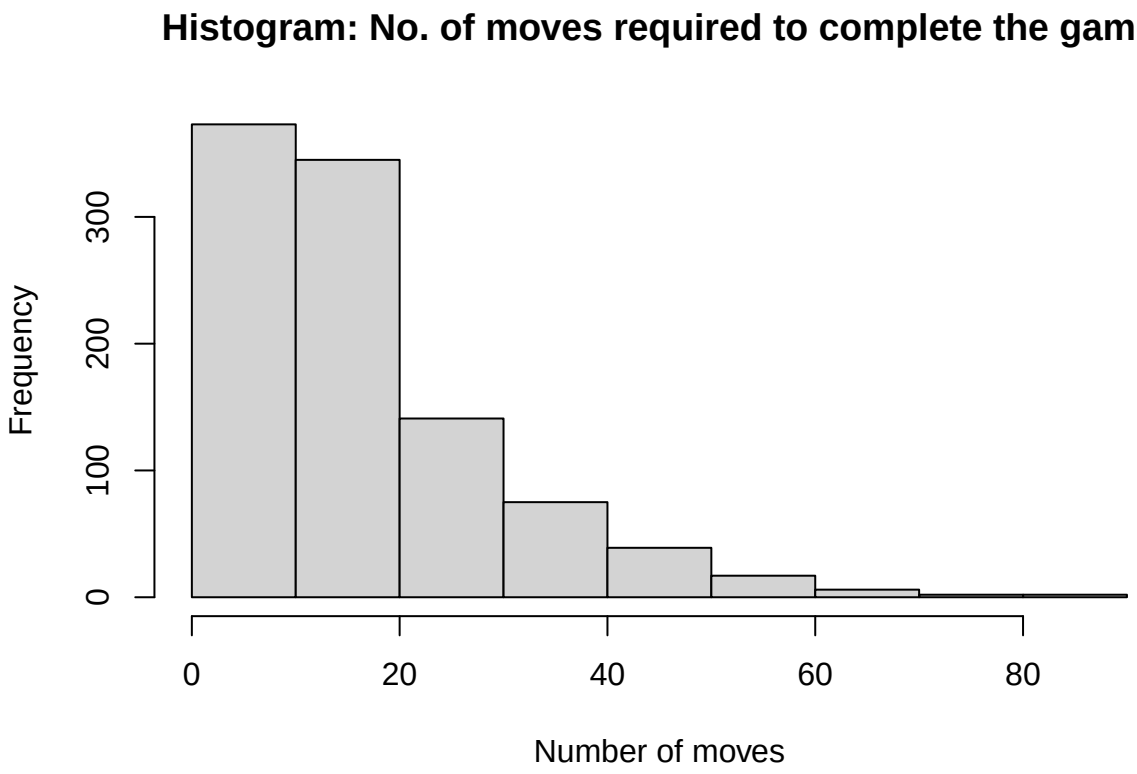
```
move_count_D4 <- replicate(1000, count_num_moves("D4", 0.95, to_print = F))
hist(move_count_D4, main ="Histogram: No. of moves required to complete the game",
     xlab = "Number of moves")
```




```
cat("Average moves required for probability 0.95 starting on D4: ",
    mean(move_count_D4), "\n")
```

```
## Average moves required for probability 0.95 starting on D4:  18.356
```

```
move_count_F6 <- replicate(1000, count_num_moves("F6", 0.05, to_print = F))
hist(move_count_F6, main ="Histogram: No. of moves required to complete the game",
     xlab = "Number of moves")
```




```
cat("Average moves required for probability 0.05 starting on F6:: ",
    mean(move_count_F6), "\n")
```

```
## Average moves required for probability 0.05 starting on F6::  17.294
```

Part 6

```
moves_A <- c(25,13,16,24,11,12,24,26,15,19,34)
moves_B <- c(35,41,23,26,18,15,33,42,18,47,21,26)
t.test(x = moves_A, y = moves_B, alternative = "two.sided", paired = F)
```

```
##
##  Welch Two Sample t-test
##
## data:  moves_A and moves_B
## t = -2.3463, df = 19.468, p-value = 0.02968
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -16.7146664  -0.9671518
## sample estimates:
## mean of x mean of y
##  19.90909  28.75000
```