Registration Number: 200206312

In the beginning, I use knitr options to customize chunk output. Ensure all the codes used is visible in the R Markdown report and prevent all warnings and messages that are generated by code from appearing in the finished file.

```
knitr::opts_chunk$set(echo = TRUE, warning=FALSE, message=FALSE)
```

Part 1. Write an R function that takes the current square as an argument and returns the square that the token moves to following the rules of the game.

First, I store the letters in a letter grid.

```
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")
lgrid
```

Then, I write an R function that takes the current square as an argument and returns the square that the token moves to following the rules of the game.

```
# Argument 'start' is current square's coordinate.
make_move <- function(start){
  x <- start[1]
  y <- start[2]
  # If the player’s token is on a square that is not on the edge of the board,
  # then the token moves at random to one of the adjacent squares.
  # If the token is on a square that is on the edge of the board,
  # then the token moves at random to one of the 64 squares on the board.
  if ((x > 1 & x < 8) & (y > 1 & y < 8)){
    adjacent_square <- list(c(x-1, y-1), c(x-1, y), c(x-1, y+1), c(x, y-1),
                            c(x, y+1), c(x+1, y-1), c(x+1, y), c(x+1, y+1))
    finish <- as.vector(unlist(sample(adjacent_square, 1)))
    finish_x <- finish[1]
    finish_y <- finish[2]
  } else{
    finish_x <- sample(1:8, 1)
    finish_y <- sample(1:8, 1)
  }
  return(c(finish_x, finish_y))
}
```

Part 2. Suggest suitable rules for deciding whether to add the letter to the player's collection if the token lands on a white square. Briefly discuss why your strategy will meet this aim and how it could be improved at the cost of increasing complexity.

In general, I formulate rules based on the number of letters and the combination of existing letters in the collection(i.e., how many repeated letters in the collection). The specific rules are as follows:

1. When the number of letters in the collection is **less than 3**, directly copy the letter on the current white square into the collection.

2. When the number of letters in the collection is **equal to 3**, operate according to the following rules:

- If **two or three letters** in the collection are the **same**, whatever letter is on the current square is copied directly to the fourth position of the collection.
- If the **three letters** are **different** from each other, only if the letter on the current square is the same as any one of the previous three letters, it will be copied to the collection.

3. When the number of letters in the collection is **equal to 4**, operate according to the following rules:

- If **four letters** are the **same**, whatever letter is on the current square is copied directly to the fifth position of the collection.
- If **any three letters** in the collection are the **same**, only if the letter on the current square is the same as any one of the first four letters, it will be copied to the fifth position of the collection.
- If the **first three letters** in the collection are all **different** (that is, the letters in the current collection are in the **forms** of "ABCC" or "ABCB" or "ABCA", etc.), only if the letter on the current square is same as any of the first four letters and is not equal to the fourth letter, it will be copied to the fifth position of the collection.
- If there are **two same letters in the first three letters** and the **fourth letter is not same** as any of the first three letters(that is, the letters in the current collection are in the **forms** of "AABC" or "ABBC" or "ABAC", etc.), then the letter on the current square will be copied to the fifth position of the collection only if it is same as any of the first four letters and is not equal to the most frequent letter in the current collection.
- If there are **two same letters in the first three letters**, the **fourth letter is same** as any of the first three letters and the current collection are in the **forms** like "AABB"(or "ABBA","ABAB", etc.), then whatever letter is on the current square is copied directly to the fifth position of the collection.
- If there are **two same letters in the first three letters**, the **fourth letter is same** as any of the first three letters and the current collection are in the **forms** like "AABA"(or "ABAA","BAAA", etc.), then the letter on the current square will be copied to the fifth position of the collection only if it is same as any of the first four letters in the current collection.

For the reason of player can rearrange the letters in their collection in any order to check whether they can form a five-letter palindrome, it is certain that **no matter what the first three letters of the collection are**, a five-letter palindrome can be formed through appropriate rules. Therefore, as long as the token does not land on the green square, it is guaranteed that every letters in the **first three possible moves** of the game **will be collected** in the collection. I think this can reduce the number of moves required to complete the game as much as possible, meanwhile, since only the last two positions of the collection need to be judged whether to place letters in it, this procedure is easy to implement.

One possible way to improve: Put the letters on the squares of the **first four moves into the collection** one by one (including the initial square). If the four letters in the collection are different from each other at this time, remove the fourth letter and add a letter that is the same as one of the first three letters to the collection. If there are repeated letters in the collection, the number of repeated letters determines what the last letter should be. For example: if the first four letters are all the same, you can take any letter as the last one and put it in the collection. If three of the first four letters are the same, the last letter only needs to be the same as one of the first four letters, etc. Through the above rules, the number of moves to complete the game may be less, but doing so will result in more judgment conditions of the program and increase the complexity.

Part 3. Write some R code to play the game with the strategy you specified in part 2.

```r
# Determine whether the current square's color is white or green.
on_green_square <- function(square_coor){
  if(isTRUE(square_coor[1] == 6 & square_coor[2] == 2) |
     isTRUE(square_coor[1] == 7 & square_coor[2] == 3) |
     isTRUE(square_coor[1] == 2 & square_coor[2] == 6) |
     isTRUE(square_coor[1] == 3 & square_coor[2] == 7)) return(TRUE) else return(FALSE)
}
# Determine whether there are two identical letters in the first three letters of the collection.
Two_same_letters <- function(c){
  if(isTRUE(c[1] == c[2] & c[1] != c[3]) |
     isTRUE(c[1] == c[3] & c[1] != c[2]) |
     isTRUE(c[2] == c[3] & c[2] != c[1])) return(TRUE) else return(FALSE)
}
# Determine whether the first three letters of the collection are the same.
Three_same_letters <- function(c){
  if(isTRUE(c[1] == c[2] & c[1] == c[3])) return(TRUE) else return(FALSE)
}
# Determine whether the first four letters of the collection are the same.
Four_same_letters <- function(c){
  if(isTRUE(c[1] == c[2] & c[1] == c[3] & c[1] == c[4])) return(TRUE) else return(FALSE)
}
# Determine whether the first four letters of the collection are in the
# form of "AAAB" or "AABB", etc.
AAAB_or_AABB <- function(c){
  if(sum(c[1] == c) == 1 | sum(c[1] == c) == 3) return(TRUE)
  if(sum(c[1] == c) == 2) return(FALSE)
}
# Get the most frequent letter in the current collection.
# "match" function returns a vector of the positions of (first) matches of
# its first argument in its second.
# "tabulate" function takes the integer-valued vector and counts the number of
# times each integer occurs in it.
getmode <- function(c) {
  uniqc <- unique(c)
  uniqc[which.max(tabulate(match(c, uniqc)))]
}
# Suggest suitable rules for deciding whether to add the letter to the player’s
# collection if the token lands on a white square.
# The function returns a updated collection.
white_square <- function(square_coor, collect){
  square_x <- square_coor[1]
  square_y <- square_coor[2]
  len_col <- length(collect)
  if(isTRUE(len_col == 0)) collect[len_col+1] <- lgrid[square_x, square_y]
  if(isTRUE(len_col == 1)) collect[len_col+1] <- lgrid[square_x, square_y]
  if(isTRUE(len_col == 2)) collect[len_col+1] <- lgrid[square_x, square_y]
  if(isTRUE(len_col == 3)){
    if(isTRUE(Two_same_letters(collect)) |
       isTRUE(Three_same_letters(collect))) collect[len_col+1] <- lgrid[square_x,
                                                                        square_y]
    if(isFALSE(Two_same_letters(collect)) &
       isFALSE(Three_same_letters(collect))){
      if(any(collect == lgrid[square_x, square_y])){
         collect[len_col+1] <- lgrid[square_x, square_y]
      }
    }
  }
  if(isTRUE(len_col == 4)){
    if(isTRUE(Four_same_letters(collect))) collect[len_col+1] <- lgrid[square_x,
                                                                       square_y]
    if(isTRUE(Three_same_letters(collect))){
      if(isTRUE(any(collect == lgrid[square_x, square_y]))){
        collect[len_col+1] <- lgrid[square_x, square_y]
      }
    }
    if(isFALSE(Two_same_letters(collect)) & isFALSE(Three_same_letters(collect))){
      if(isTRUE(any(collect == lgrid[square_x, square_y])) &
         isTRUE(collect[4] != lgrid[square_x, square_y])){
         collect[len_col+1] <- lgrid[square_x, square_y]
         }
    }
    if(isTRUE(Two_same_letters(collect))){
      if(isFALSE(any(collect[1:3] == collect[4]))){
        if(isTRUE(any(collect == lgrid[square_x, square_y])) &
           isFALSE(getmode(collect) == lgrid[square_x, square_y])){
          collect[len_col+1] <- lgrid[square_x, square_y]
           }
      }
      if(isTRUE(any(collect[1:3] == collect[4]))){
        if(isTRUE(AAAB_or_AABB(collect))){
          if(isTRUE(any(collect == lgrid[square_x, square_y]))){
            collect[len_col+1] <- lgrid[square_x, square_y]
          }
        }
        if(isFALSE(AAAB_or_AABB(collect))) collect[len_col+1] <- lgrid[square_x,
                                                                       square_y]
      }
    }
  }
  return(collect)
}
# Suggest suitable rule for collection if the token lands on a green square.
# The function returns a updated collection.
green_square <- function(square_coor_g, collect_g, prob){
  square_x_g <- square_coor_g[1]
  square_y_g <- square_coor_g[2]
  elements_remove <- lgrid[square_x_g, square_y_g]
  if(isTRUE(rbinom(1, 1, prob) == 1)){
    collect_g <- c("f", "f", "h", "k")
  } else{
    if(isTRUE(any(collect_g == lgrid[square_x_g, square_y_g]))){
      collect_g <- collect_g[!(collect_g %in% elements_remove)]
    }
  }
  return(collect_g)
}
# Function to play the game until there are five letters in the collection.
# The function returns the number of moves taken.
# If `to_print=T` then the function prints the squares visited and collection in each move.
count_num_moves <- function(start_sq, start_col, p, to_print){
  num_moves <- 0
  if(isTRUE(to_print)) cat("Start at square", start_sq, "\t")
  if(isTRUE(to_print)) cat("Collection is empty\n")
  while(isFALSE(length(start_col) == 5)){
    if(isTRUE(on_green_square(start_sq))){
      next_col <- green_square(start_sq, start_col, p)
      if(isTRUE(to_print)) cat("Collection is [", next_col, "]\n")
    }
    if(isFALSE(on_green_square(start_sq))){
      next_col <- white_square(start_sq, start_col)
      if(isTRUE(to_print)) cat("Collection is [", next_col, "]\n")
    }
    num_moves <- num_moves + 1
    next_sq <- make_move(start_sq)
    if(isTRUE(to_print & length(next_col) != 5)) cat("move to square", next_sq, "\n")
    start_sq <- next_sq
    start_col <- next_col
  }
  return(num_moves)
}
```

Due to the length limitation of the report, I will not demonstrate the results of the "count_num_moves" function. If you want to view the results, you can run the following codes.

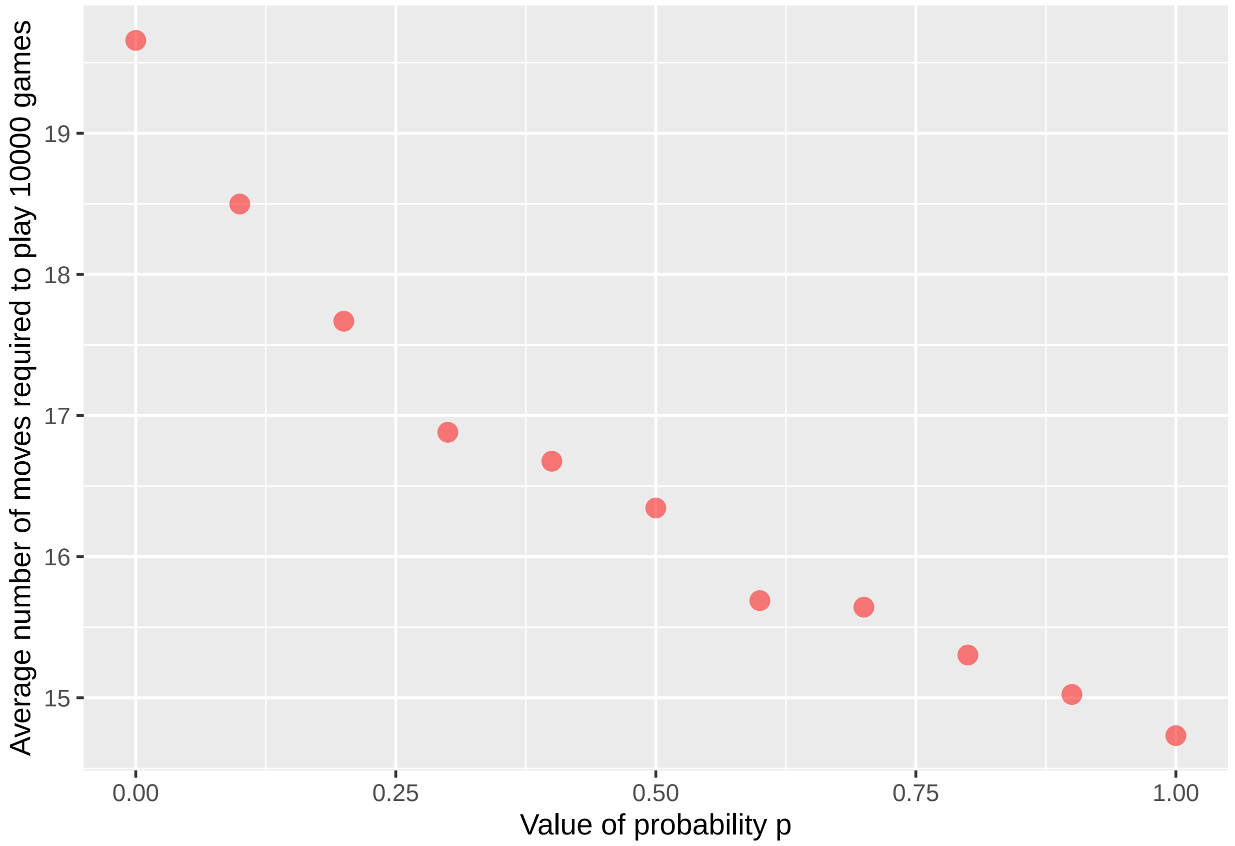
```
start <- c(4,4) # The token is initially placed on square D4
collection <- c()
count_num_moves(start, collection, 0.8, T) # p = 0.8
```

Part 4. The token is initially placed on square D4. Describe how the time to finish the game depends on p.

In this game, we can assume that the time required for each move of the game is equal, so that the average number of moves required to complete the game can be used to estimate the time required to complete the game. Therefore, set p equal to 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 and 1 successively, calculate the average number of moves required to play 10000 games under different p values respectively. Then, use ggplot to draw a graph to show the relationship between p and the number of moves.

```
# When token is initially placed on D4 "start", play game "num" times with the probability p.
record_move_num <- function(num, coor, p){
  dist_of_num_moves <- replicate(num, count_num_moves(coor, collection, p, F))
  m = mean(dist_of_num_moves)
```

```r
  return(m)
}
mean <- c()
start <- c(4,4) # D4
prob_values <- seq(0, 1, by = 0.1)
for (j in prob_values) {
  mean[j * 10 +1] <- record_move_num(10000, start, j)
}
library(tidyverse)
df <- as.data.frame(mean)
ggplot(data = df, aes(x = prob_values, y = mean)) +
  geom_point(size = 3, alpha = 0.5, colour = "red") +
  labs(x = "Value of probability p",
       y = "Average number of moves required to play 10000 games")
```

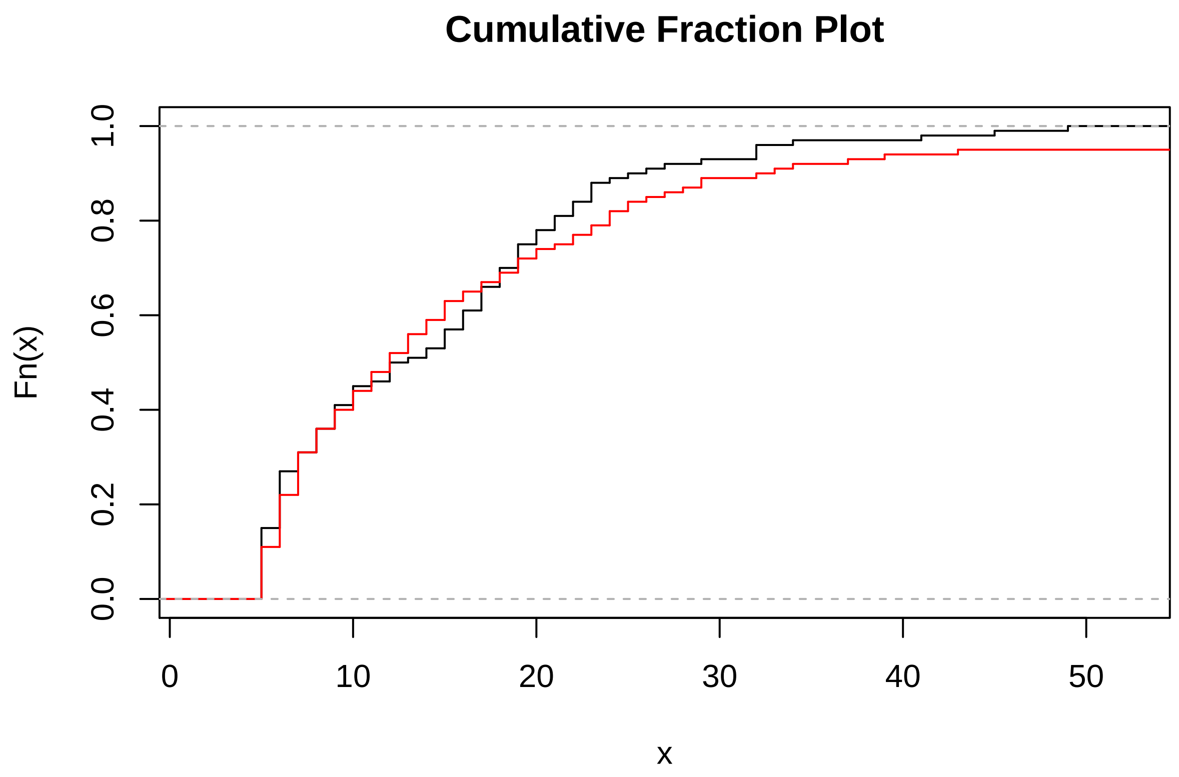


According to the figure above, the time required to complete the game decreases as the P value increases(**inverse relationship**).

Part 5. By creating plots and calculating some simple summary statistics, assess the evidence that the probability distributions of the number of moves required to complete the game are identical for the starting points D4 with p=0.95 and F6 with p=0.05.

First, I respectively play game 100 times under two different starting conditions((D4,0.95) AND (F6, 0.05)) and record the number of moves needed to complete the game. Then, I use ecdf() function to draw cumulative fraction plot to observer two sets of data. Finally, I use the two sets of data to perform Kolmogorov-Smirnoff test(KS test) to assess whether the two probability distributions are identical.

```
startA <- c(4,4) #D4
startB <- c(6,6) #F6
dist_of_num_movesA <- replicate(100, count_num_moves(startA, collection, 0.95, F))
dist_of_num_movesB <- replicate(100, count_num_moves(startB, collection, 0.05, F))
plot(ecdf(dist_of_num_movesA), verticals = TRUE, main = "Cumulative Fraction Plot" ,cex=0)
lines(ecdf(dist_of_num_movesB), verticals = TRUE, col="red", cex=0)
```



```
ks.test(dist_of_num_movesA, dist_of_num_movesB)
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  dist_of_num_movesA and dist_of_num_movesB
## D = 0.09, p-value = 0.8127
## alternative hypothesis: two-sided
```

From the figure above, the black and red lines represent the results with the starting point is D4 and F6 respectively. We can see that the maximum difference between the two distributions is very small, that is, the two distributions are very close and "approximately" identical. In the KS test, the null hypothesis is "the two distributions to be tested have the same distribution", we can support the previous inference through analyzing the test results

- "D", that is, the maximum distance between the distributions, is close to 0, so the two distributions are very close.
- "p-value" is greater than 0.05(test at the 5% level), that is, the null hypothesis cannot be rejected.

That indicates the evidence that the two probability distributions are identical.

Part 6. Assess the evidence that $E(X_A)=E(X_B)$.

In this part, since we don't know whether the overall distribution obeys the normal distribution, we can't directly use the T test to draw conclusions. I use unpaired(Two Sample) T test to assess the evidence. The null hypothesis is H_0: mean number of moves needed to complete the game are equal.

If H_0 is true, then any difference in the two sample means would be solely due to how the results of the required number of moves obtained by playing games 23 times were assigned to the two groups(One holds 11 results and the other holds 12). It must be equal to the probability of assigning the number of moves to the two groups in such a way that the imbalance occurs, as long as the number of moves were assigned to the two groups at random. This is the principle idea behind randomization tests. The randomization test is performed as follows:

1. Label the 23 results from 1 to 23.
2. Randomly re-assign the 23 results to the two groups.
3. Re-calculate the test-statistic for this permuted data.
4. Repeat steps 2 and 3 to obtain n sampled test-statistics.
5. Calculate the estimated p-value of the observed test statistic.

I implement the above procedure using 9999 random permutations:

```r
data_A <- c(25, 13, 16, 24, 11, 12, 24, 26, 15, 19, 34)
data_B <- c(35, 41, 23, 26, 18, 15, 33, 42, 18, 47, 21, 26)
meanA <- mean(data_A)
meanB <- mean(data_B)
s <- sqrt((var(data_A) * 10 +  var(data_B) * 11 ) / 21)
all_data <- c(25, 13, 16, 24, 11, 12, 24, 26, 15, 19, 34,
              35, 41, 23, 26, 18, 15, 33, 42, 18, 47, 21, 26)
calc_T <- function(){
  perm <- sample(1:23,23, replace = F)
  grA <- all_data[perm[1:11]]
  grB <- all_data[perm[12:23]]
  meanA_new <- mean(grA)
  meanB_new <- mean(grB)
# s_new is the new square root of pooled sample variance
  s_new <- sqrt((var(grA) * 10 +  var(grB) * 11 ) / 21)
# Finally Obtain t, which is a Student t quantile with n1(11)+n2(12)-2 degrees of freedom.
  (meanA_new - meanB_new) / (s_new * sqrt(1/11 + 1/12))
}
n <- 10000
sim_T <- replicate(n - 1, calc_T())
# T_obs: The observed test statistic.
T_obs <- (meanA - meanB) / (s * sqrt(1/11 + 1/12))
(p_val <- (sum(abs(sim_T) >= abs(T_obs)) + 1) / n) # Calculate the estimated p-value.
```

```
## [1] 0.0321
```

According to the output, "p-value" is smaller than 0.05(test at the 5% level), that is, the null hypothesis is rejected. That indicates the evidence that $E(X_A)\neq E(X_B)$.