200206688-MAS6024

Forming the letter grid of the board

```r
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")
```

This function checks if the token coordinates are on a green square

```r
green_sqrs <- matrix(NA, nrow = 4, ncol = 2)
green_sqrs[1,] <- c(2, 6)
green_sqrs[2,] <- c(3, 7)
green_sqrs[3,] <- c(6, 2)
green_sqrs[4,] <- c(7, 3)

is.green <- function(g_coords){
  g <- FALSE
  if (g_coords[1] == green_sqrs[1, 1] && g_coords[2] == green_sqrs[1, 2] ||
      g_coords[1] == green_sqrs[2, 1] && g_coords[2] == green_sqrs[2, 2] ||
      g_coords[1] == green_sqrs[3, 1] && g_coords[2] == green_sqrs[3, 2] ||
      g_coords[1] == green_sqrs[4, 1] && g_coords[2] == green_sqrs[4, 2]) {
    g <- TRUE
  }
  return(g)
}
```

1 Part 1:

Function to move the counter using the specified rules Making sure if the token is on the edge it moves to a random square

```r
move_counter <- function(start_coords){
  x <- c(start_coords[1])
  y <- c(start_coords[2])
  if (x == 1 || x == 8 || y == 1 || y == 8) {
    finish_coords <- sample(x = 1:8, size = 2, replace = T)
  }
  else {
    (adj_coords <- sample(x = -1:1, size = 2, replace = T))
    while (adj_coords == c(0, 0)) adj_coords <- sample(x = -1:1, size = 2, replace = T)
    finish_coords <- start_coords + adj_coords
  }

  return(finish_coords)
}
```

Forming function to decide whether to take a letter or not

```r
take_letter <- function(letters, next_letter) {
  take.letter <- sample(100, size = 1)
    if (take.letter > 50) {
      letters <- c(letters, next_letter)
    }
return(letters)
}
```

A function which checks if the letter collection forms a palindrome

```r
is.palindrome <- function(letters) {
    word <- paste(letters, sep = "", collapse = "")
    rawWord <- charToRaw(tolower(word)) ## converts to lower case
    palindrome <- identical(rawWord, rev(rawWord))
    return(palindrome)
}
```

2 Part 2:

The player has a random 50% chance of adding the letter to the players collection, until the fourth letter which is only picked up if it leads to less than four unique letters in the players collection. Otherwise, it would be impossible to form a palindrome when picking up a fifth letter. The fifth letter is only added if it causes the players letter collection to form a palindrome.

An alternative would be to implement user input. For example,if the token lands on a white square the user will have a choice of whether to collect that letter or not. Once they reach the maximum number of letters (5) when landing on a white square they will be able to decide whether they want that letter. If the player does they can then decide which letter from their collection to replace.

Forming function to check if adding a fifth letter forms a palindrome

```r
fifth_letter <- function(letters, next_letter){
  five_letters <- (c(letters, next_letter))
  palin <- sort_palindrome(five_letters)
  palindrome <- is.palindrome(palin)
  return(palindrome)
}
```

Function to ensure that a fourth letter is only picked up if it leads to less than four unique letters in the players collection.

```
fourth_letter <- function(letters, next_letter){
  four_letters <- (c(letters, next_letter))
  uniques <- unique(four_letters)
  if (length(uniques) < 4) {
    return(four_letters)
  }
  return(letters)
}
```

Function to sort the five letters into a palindrome if possible This function is combined with the is.palindrome function to check if a palindrome can be formed.

```
sort_palindrome <- function(p_letters){
  uniques <- unique(p_letters)
  palin <- list()
  x <- 1
  for (u in uniques) {
    counts <- 0
    for (i in p_letters){
      if (i == u) counts <- counts + 1
    }
    while (counts > 0){
      if (counts %%2 == 1) {
        palin[3] <- u
        counts <- counts - 1
      }
      else {
        palin[x] <- u
        palin[6-x] <- u
        x <- x + 1
        counts <- counts - 2

      }
    }
  }
return(palin)
}
```

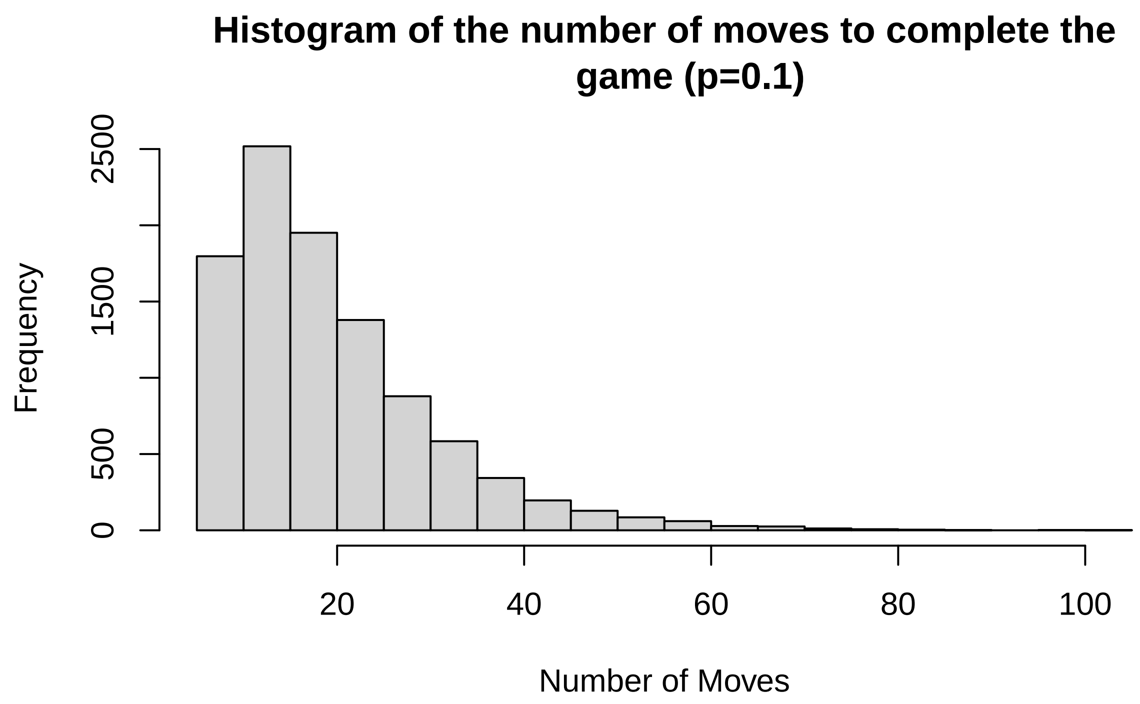
3 Part 3:

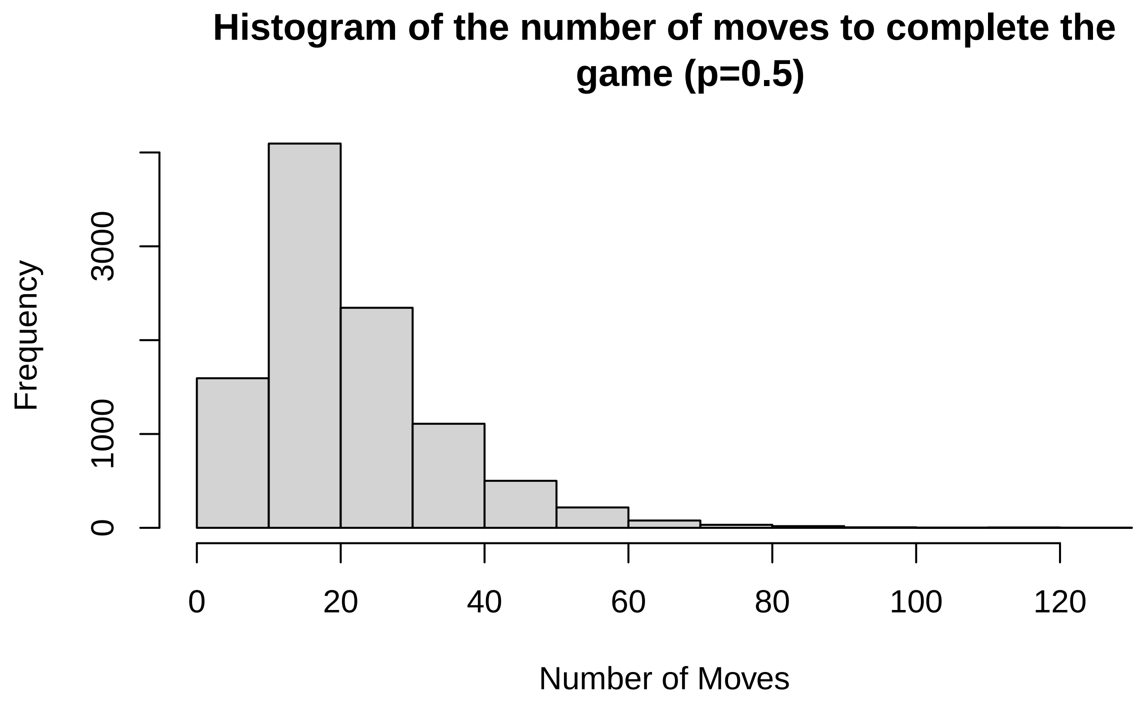
Function to simulate the game, returning the number of moves required.

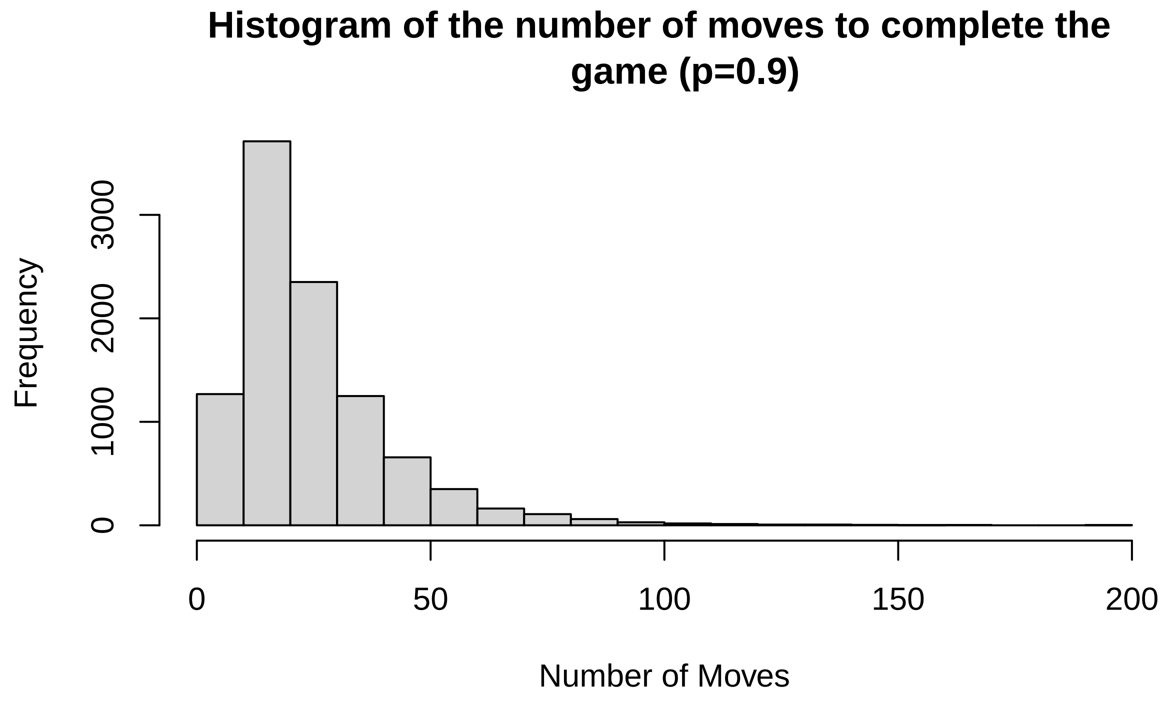
```
play_game <- function(lgrid, green_coords, start_coords, p){
  letters <- list()
  palindrome <- FALSE
  num_moves <- 1
  start_letter <- lgrid[start_coords[1], start_coords[2]]
  take_letter(letters, start_letter)

  while (palindrome == FALSE){
    next_coords <- c(move_counter(start_coords))
    start_coords <- next_coords
    num_moves <- num_moves + 1
    next_letter <- lgrid[next_coords[1], next_coords[2]]


    if (is.green(next_coords) == TRUE) {
      prob <- sample(100, size = 1) / 100
      if (prob >= p) {
        letters <- list("f", "f", "h", "k")
      }
      else {
          letters <- letters[(letters) %in% next_letter == FALSE]
      }
    }

    else if (length(letters) == 3) {
      letters <- fourth_letter(letters, next_letter)
    }

    else if (length(letters) == 4) {
      palindrome <- fifth_letter(letters, next_letter)


      if (palindrome == TRUE) {
        letters <- (c(letters, next_letter))
        num_moves <- num_moves + 1
        # print(paste("Well done, you won in", num_moves, "moves!"))
        return(num_moves)
      }

    }
    else {
      letters <- take_letter(letters, next_letter)
    }
  }
}
```

4 Part 4:

To find out how the time to finish depends on p a large number of realisations can be simulated for different values of p. These realisations can be plotted & compared to see how the mean & variance differ to show how the number of moves required to form a palindrome is dependent on p. 10000 realisations will be assessed for 3 different values of p (0.9, 0.1, 0.5).


Histogram of the number of moves to complete the game (p=0.1)
Frequency
0
500
1500
2500
20
40
60
80
100
Number of Moves


Histogram of the number of moves to complete the game (p=0.5)
Frequency
0
1000
3000
0
20
40
60
80
100
120
Number of Moves



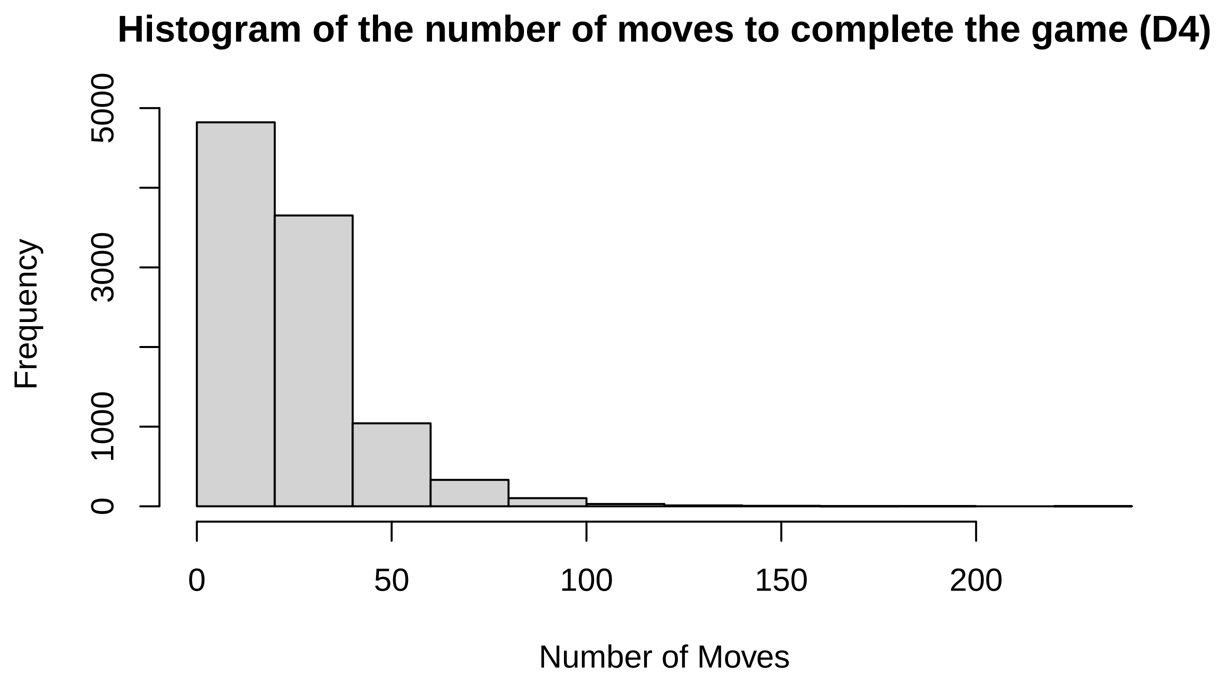
```
## [1] "Mean for 0.1p: 19.8901  Mean for 0.5p: 21.7177 Mean for 0.9p: 25.2377"
```

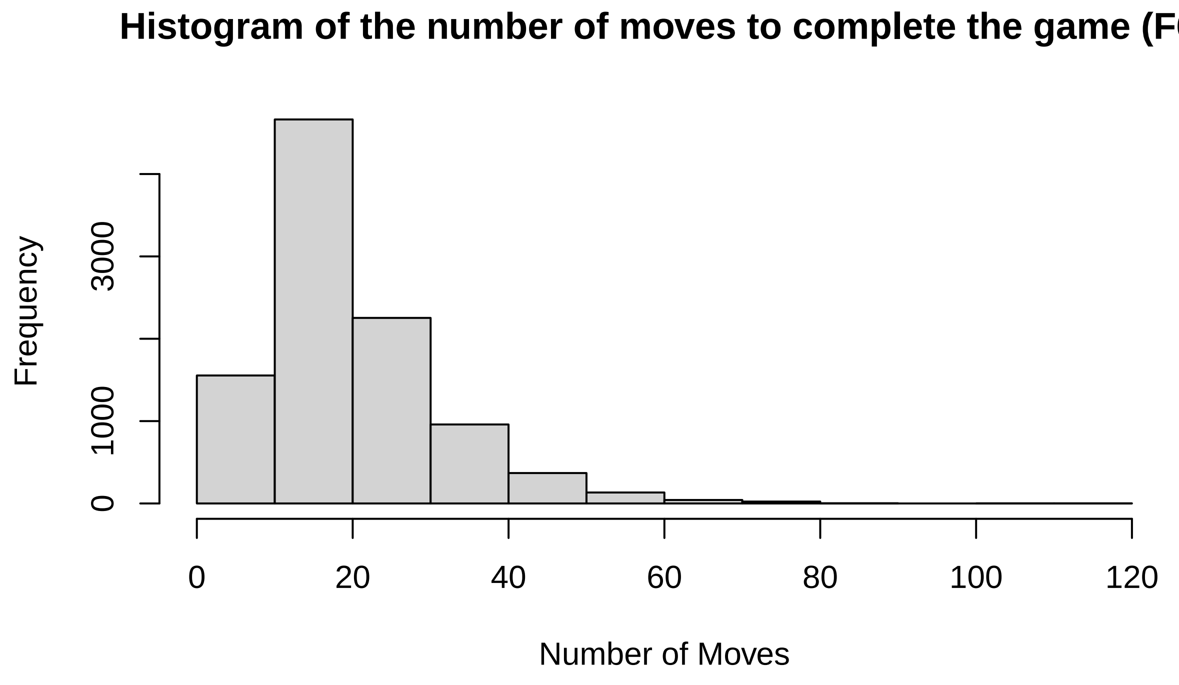
```
## [1] "Variance for 0.1p: 124.472  Variance for 0.5p: 162.951 Variance for 0.9p: 292.204"
```

These three values of p are compared over a large number of realisations and as shown by their means there appears to be a degree of dependence on the value of p for the expected number of moves. With an increase in the value of p accompanied by an increase in the average number of moves required. This is shown by the sample mean rising at each increment in the value of p. Furthermore, the variance increases alongside an increasing value of p indicating a much greater variation in the number of moves required for larger p values. Both these summary statistics illustrate that there is a dependence on the value of p, with a larger value leading to a higher number of moves required and a greater spread in the distribution.

5 Part 5:

To investigate this hypothesis the game can be simulated 10000 times for each scenario to determine whether the two scenarios can be assumed to be identical.





```
## [1] "Mean for D4 start position: 25.9502  Mean for F6 start position: 20.197"

## [1] "Variance for D4 start position: 307.139  Variance for F6 start position: 120.250"

##  2.5% 97.5%
##     8    73
```

```
##   2.5%  97.5%
##  7.975 48.025
```

The evidence suggests that the probability distributions of the number of moves are not identical. This is due to a variety of factors, firstly, there is a notable deviation in the sample means for the two scenarios with the D4 start position having a larger average of 26.0 compared to 20.2 for the F6 start position. Furthermore, the D4 start position displays a much larger variance which relates to the larger p value. In part 4 it was shown that larger p values resulted in a larger variance for the number of moves required. Finally, the stark contrast in the 95% confidence intervals for the two scenarios reinforces the difference in probability distributions. The D4 data shows a confidence interval of 8-73 moves whilst the F6 data has a confidence interval of 8 to 48 moves. The difference between the two distributions is found in the upper interval with the D4 scenario more skewed towards higher values.

6 Part 6:

The Monte Carlo approximation of the expectation of a random variable is the sample mean. As a result, the expectation of the two squares can be assumed to be the sample mean of the two samples.

```r
sqr_A <- c(25,13,16,24,11,12,24,26,15,19,34)
sqr_B <- c(35,41,23,26,18,15,33,42,18,47,21,26)
mean1 <- mean(sqr_A)
mean2 <- mean(sqr_B)
```

```
## [1] "The mean number of moves for square A is: 19.91"
```

```
## [1] "The mean number of moves for square B is: 28.75"
```

As shown above the two samples display a large difference in expectation if using the Monte Carlo approximation. However, the null hypothesis that the expectation values are equal for the two scenarios can be tested using randomization tests.

The p-value can be calculated using the function t.test as the hypothesis involves a two_sided test.

```r
t.test(x = sqr_A, y = sqr_B, alternative = "two.sided", paired = F)
```

```
##
##  Welch Two Sample t-test
##
## data:  sqr_A and sqr_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
```

```r
T_obs <- -2.3463
p_value <- 0.02968
```

The test results can be randomly assigned to the two different groups with the test statistic (p-value) recalculated. A similar p-value infers that the hypothesis is correct and the expectation values can be assumed to be equal.

```r
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)
  sqr_A <- all_data[perm[1:11]]
  sqr_B <- all_data[perm[12:23]]
  mean1 <- mean(sqr_A)
  mean2 <- mean(sqr_B)
  s <- sqrt((var(sqr_A) * 7 +  var(sqr_B) * 7 ) / 14)
  (mean1 - mean2) / (s / 2)
}
n <- 10000
sim_T <- replicate(n - 1, calc_T())
p_val <- (sum(abs(sim_T) >= abs(T_obs)) + 1) / n
```

```
## [1] "The original p value is 0.02968"
```

```
## [1] "The new p value is: 0.0105"
```

The two p-values are quite dissimilar, with the null hypothesis p-value much lower than the original p-value. So as a result, using randomization tests the null hypothesis is rejected and the two expectation values are not assumed to equal.