Jared Yu

914640019

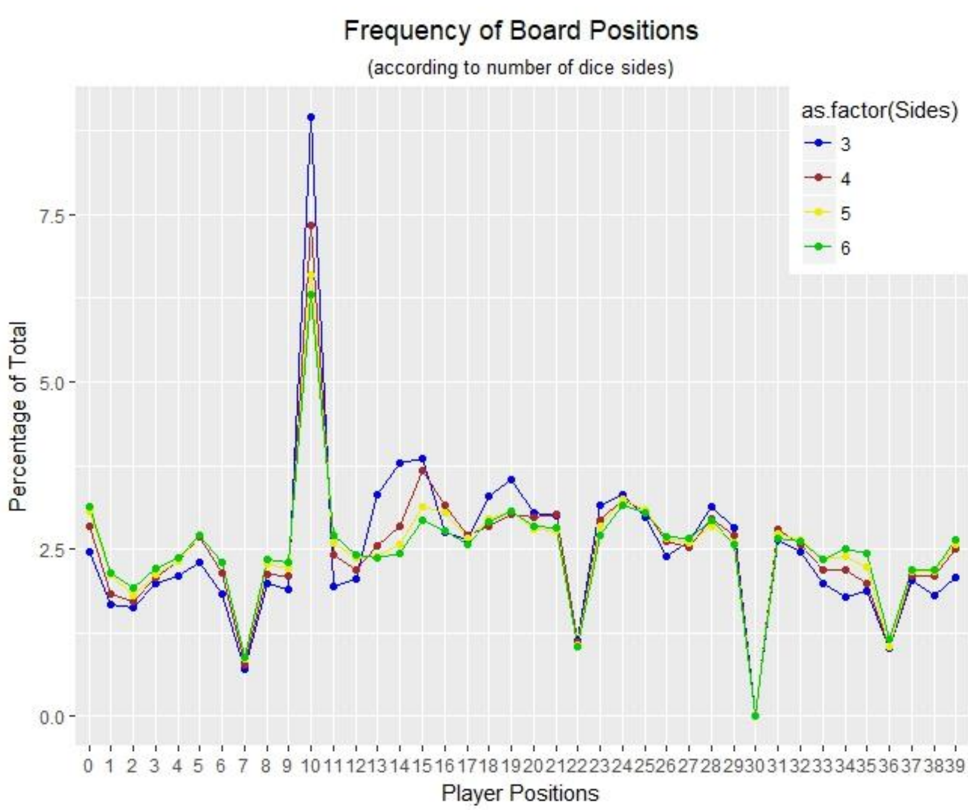
STA 141A

HW 3

HW 3 Writeup

Part 1: Working with the Board

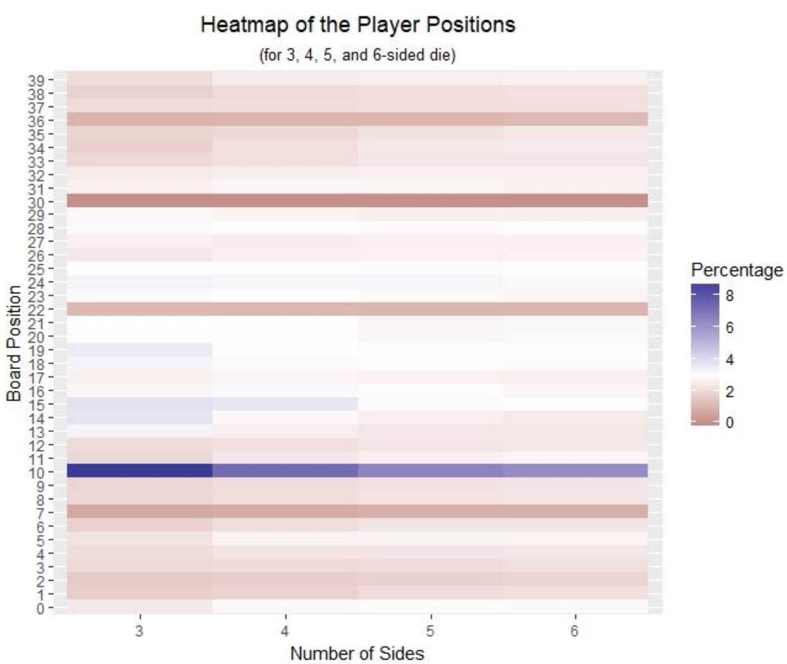
1. For the first problem of part 1, my strategy was to use separate functions that are to be called from within the main simulate_monopoly() function. In this manner I'm able to make changes in an efficient manner that is simple to check, and easy to change in the future. The separate functions include: loop_39() to check if the character position has gone past 39, triple() which checks if the player has rolled 3 consecutive pairs, chance_deck() which checks if the player has landed on a chance card, community_chest() which checks if the player lands on a community card, and jail_30() which checks if the player lands on position 30, sending them to jail.
The result from this function is that a vector of *n+1* positions, where *n* is the number of times the dice are rolled. The output also includes the factor levels to account for the 30th position which is skipped.
2. In the estimate_monopoly() function, my goal was to make use of the lapply() function so that I could repeat the simulation for a total of *k* iterations. A proportion table is used on the simulation to obtain the proportion of each of the 40 positions listed from 0-39. Each of the 4 different number of sides, from 3 – 6 are simulated for *k* = 1,000 iterations, and *n* = 10,000 turns.



The result is a line plot where the percentage of each position index is plotted according to their percentage of the total rolls. From the plot, it's clear that all die sides follow a rather similar trend. They seem to peak around the same time along with dip around the same time. It'd make sense to believe that the peaks and dips are associated with events that may cause the player to move to such positions. For instance, the first position, the 0th position, occurs more frequently then it's neighboring positions. The next peak is position 5, where a player can be sent to the Railroad 1 due to them drawing a Chance Card. Next, there's a dip at position 7 which is a Chance Card. The reason is likely that after landing here, the player has a likelihood of moving to another position within the same move (due to a Chance Card placing them elsewhere). The largest peak is at position 10, this is due to this spot being the Jail location. There are more ways to be sent to jail than any other event. That'd make this the most frequent position out of all the others on the board. At position 15 there's another Railroad which would explain the spike in data. Like position 7, position 22 is another Chance Card where the player has a greater likelihood of being moved to another location within the same turn, lowering the frequency of a player staying there. An interesting observation is that position 30 has a 0% occurrence, this is due to that location requiring the player to immediately be sent to jail. That would explain how the player never lands here, and partially also why position 10 is so large. Like position 7 and 22, 36 is a Chance Card where the player can be moved elsewhere.

After noticing the trend of the simulations with 3-sided die, it seems to make sense why there are peaks after position 10. The reason is that since 10 is the most frequent position due to events sending the player there, the simulation with 3-sided die must 'restart' from this location quite often. Also, since the player can only roll from 2-6, the 3-sided die simulation would stay within this range for a longer period on average than in comparison to the other die.

Additionally, I've included a heatmap of the same data after being told that it would be fine to provide both. Here it is below:



From this heatmap it becomes easier to notice that the higher-sided dice seem to be able to balance more the spread of the board positions. In the darker 3-sided die, it's possible to see that there is a larger spike in the data in the 10th jail position, so many of the other positions become much darker, meaning that there is a lower frequency of them occurring. As the number of sides increase, the player seems to be able to spend time in other board positions indicated by the lighter colors of the various board positions. It's again possible to see that the low frequency positions are from Chance or Community cards possibly moving the player to another position, or G2J which has 0% of occurring due to the event it causes.

From the plots it is not easy to determine exactly which 3 positions occurred most frequently among the different sided die. Using dplyr, it was easy to find the following table:

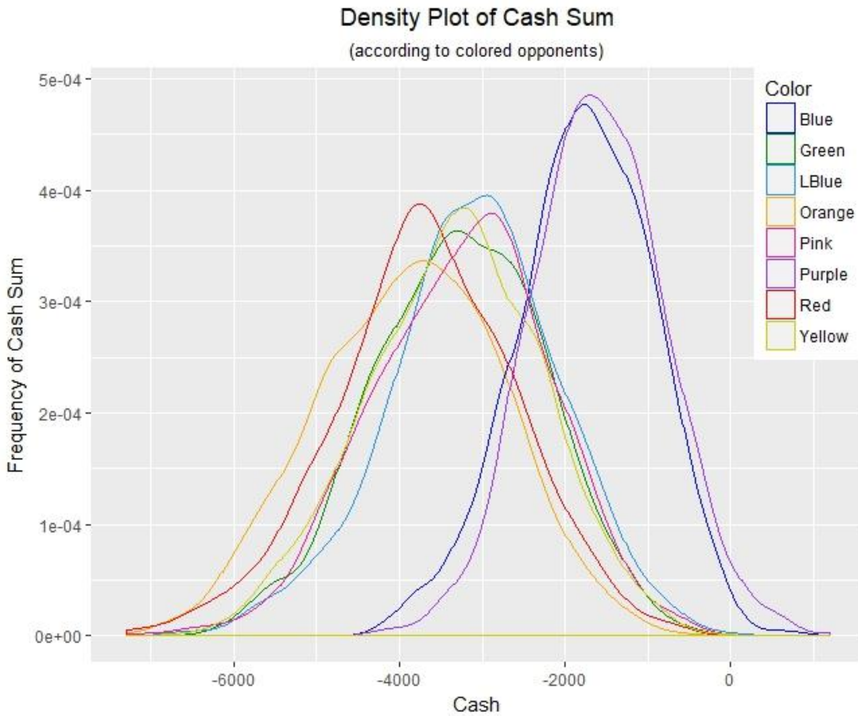
Board Position	Percentage (%)	Number of Sides
10	8.96	3
15	3.86	3
14	3.79	3
10	7.34	4
15	3.67	4
24	3.26	4
10	6.59	5
24	3.26	5
15	3.13	5
10	6.30	6
24	3.15	6
0	3.13	6

3. To determine the *standard error* of the long-term probability of ending a turn in jail, we were tasked to use the parameters of k = 1,000 simulations, n = 10,000 rolls, and d = 6-sided die. For this problem, a different estimate_monopolySE() was created. This function returns the frequency that's associated with each position on the board. Then this function was repeated using replicate() 1,000 times to generate enough data. Then the position of 10 was extracted from the result, and the result is a vector of all the different frequencies that the 10th position occurred. After obtaining this vector, the function sd() was used to determine the standard deviation of the sample. This leads to the *standard error* of the player going to jail being 0.2078264%.

Part 2: Working with Money

1. Since the problem requires us to include the money-related aspects of the game of Monopoly, I created several additional functions outside of simulate_monopoly2() which are called inside to help account for the different cases where money is involved. I included a function called charge_rent() which determines if a player lands on the property of a particular color, and proceed with determine what amount would be subtracted from the player's wallet. This amount is also used to determine how much money is owed to the color at the end of the simulation. Additionally, there's a tax_or_go() function which checks whether or not the current position of the player is a Tax or if they're on GO where they could either lose money or gain money. An alternate function was created called pass_go(), which checks whether or not the player has passed the GO position rather than landed on it. A last function called pay_day() works by summing all the cash whether it is negative or positive to determine how much money was earned or lost during a specific turn. Another function called bail() would check whether the player had ended the turn in jail, and would subsequently turn that turn's cash to 0.

   To test the new simulation, subsets of the property data was taken to determine which positions each color is associated with. This helps to ensure that the function charges the correct amount according to which color the opponent is. The 'revenue' column from the properties.csv was also used to determine the rent paid to the opponent for landing on their property.



2. The colors for each of the players were independently tested using estimate_monopoly2(). This second estimate_monopoly2() differs from the first one by including the property and cost information needed to determine the results for having a specific colored opponent. Another function named color_simulations() was created to create a matrix of 1000 observations for each of the 8 colors. This was the sum of money for each of the games where n = 100, d = 6. The results were then mixed into a data frame. The amount of cash from playing against a color is shown on the x-axis. The frequency of the cash sums are noted on the y-axis. After running k = 1,000 iterations of n = 100 turns on d = 6-sided die, the effect is that playing Blue and Purple can prove to be effective since they can lead to the other player's cash being a positive amount, despite this probability being rather low. Instead it may be helpful to think of Purple and Blue to be the 'least-difficult' opponents. However, other colors, especially Red and Orange are dangerous opponents since on average the player will have to pay these colors high rent for landing on their properties. Other colors are somewhere in between these two frequencies. It is interesting that Yellow is a color that seems to have two-peaks in the density, something not seen in the other colors. A pattern that is noticeable, is that Purple and Blue only own 2 properties each, and after many iterations of games where n is sufficiently large, it seems that this factor takes a big role. Other colors which have 3 properties end up charging the player more often than Purple or Blue do. For this reason, they are easier opponents in comparison to other colors.
3. For the simulations in problem 3, I had to edit the previous simulate_monopoly2() and estimate_monopoly2() with the versions simulate_monopoly2b() and estimate_monopoly2b(). The difference now is that the output allows for a 3rd matrix in the list that is different from the position matrix and the wallet matrix which accounts for money the player may have gained or lost. This 3rd matrix details the amount of money that the color is owed for having had the opponent land on their property. A new function called color_cash_balance() that simulates one game for all of the pairwise color combinations, and outputs a list showing the money gained or lost by each color from all of the matchups. A last function was created called simulation_100() which repeats the pairwise matches from color_cash_balance(), and determines the winner for

each while repeating the process 100 times. The argument allows for the number of sides to be changed, and the result are 3 matrices for n = 25, 50, and 100 games.

Color	Wins	Losses	Win %	# of games
Purple	653	47	93.29	25
Light Blue	627	73	89.57	25
Orange	422	278	60.29	25
Pink	393	307	56.14	25
Blue	364	336	52	25
Red	179	521	25.57	25
Yellow	142	558	20.29	25
Green	10	690	1.43	25

Color	Wins	Losses	Win %	# of games
Purple	613	87	87.57	50
Light Blue	611	89	87.29	50
Orange	446	254	63.71	50
Pink	402	298	57.43	50
Blue	300	400	42.86	50
Red	217	483	31	50
Yellow	170	530	24.29	50
Green	28	672	4	50

Color	Wins	Losses	Win %	# of games
Light Blue	593	107	84.71	100
Orange	523	177	74.71	100
Purple	452	248	64.57	100
Pink	427	273	61	100
Red	291	401	41.57	100
Yellow	244	456	34.86	100
Blue	196	504	28	100
Green	59	641	8.43	100

It is interesting to note that the tables for n = 25 and 50 are identical in terms of the ordering. Purple which was the most difficult opponent from part 2.2 is the leader in the standings. However, the second most difficult opponent Blue is somewhat further behind. This is possibly partially due to the cost of construction of properties being so great for Blue. For n = 100 games, the ordering starts to change, and this is possibly because of other money-related factors beginning to become more noticeable as the players move more around the board. An example would be that playing more Community or Chance cards can allow the player to make more money by moving further to pass GO more often, and therefore giving them the possibility of making more money in ways that aren't related to rent.

Credit: Huong Vu, Bailey Wang, Tiffany Chen, @312, @386, @382, @381, @397, @395

Resources:

https://www.rdocumentation.org/packages/dplyr/versions/0.7.3/topics/arrange

https://stackoverflow.com/questions/5208679/order-bars-in-ggplot2-bar-graph

http://www.cookbook-r.com/Graphs/Bar_and_line_graphs_(ggplot2)/#line-graphs

https://stackoverflow.com/questions/6574188/r-ggplot2-how-can-i-independently-adjust-the-x-axis-limits-on-a-facet-grid

https://stackoverflow.com/questions/23420961/plotting-multiple-lines-from-a-data-frame-with-ggplot2

https://stackoverflow.com/questions/43359050/error-continuous-value-supplied-to-discrete-scale-in-default-data-set-example/43359104

http://sape.inf.usi.ch/quick-reference/ggplot2/colour

https://generalassemb.ly/design/visual-design/color-theory

http://www.cookbook-r.com/Graphs/Legends_(ggplot2)/

https://siguniang.wordpress.com/2011/01/08/rplay-with-ggplot2-part-03_positive-negative-barplot/

https://stackoverflow.com/questions/5620885/how-does-one-reorder-columns-in-a-data-frame

https://stackoverflow.com/questions/16074440/r-ggplot2-center-align-a-multi-line-title

http://www.sthda.com/english/wiki/ggplot2-density-plot-quick-start-guide-r-software-and-data-visualization

https://www.teachucomp.com/sort-a-table-in-word-instructions/

http://stat.ethz.ch/R-manual/R-devel/library/base/html/levels.html

Code Appendix:

```
library(ggplot2)
library(beepr)
library(tidyverse)
library(dplyr)

properties = read.csv("properties.csv")
colors = read.csv("color_combos.csv")

### Part 1

# 1

community_chest = function(current_position, current_deck) { # Community Ches
t
  if (current_position == 2 || current_position == 17 || current_position ==
33) {
    current_deck_draw = current_deck[1] # Draw card
    if (current_deck_draw == 1) {
      current_position = 0 # sends player to GO
    } else if (current_deck_draw == 2) {
      current_position = 10 # sends player to Jail
    }

    current_deck = c(current_deck[-1], current_deck[1]) # Place card on the b
ottom
  }
  community_result = list(current_position, current_deck) # Return both the d
eck and position
  return(community_result)
} # Community Chest cards

chance_deck = function(current_position, current_deck) {
  if (current_position == 7 || current_position == 22 || current_position ==
36) { # Chance
    current_deck_draw = current_deck[1] # Draw card
    if (current_deck_draw == 1) { # check if card drawn is GO or JAIL
      current_position = 0 # sends player to GO
    } else if (current_deck_draw == 2) {
      current_position = 10 # sends player to JAIL
    } else if (current_deck_draw == 3) {
      current_position = 11 # sends player to C1
    } else if (current_deck_draw == 4) {
      current_position = 24 # sends player to E3
    } else if (current_deck_draw == 5) {
      current_position = 39 # sends player to H2
    } else if (current_deck_draw == 6) {
      current_position = 5 # sends player to R1
    } else if (current_deck_draw == 7 | current_deck_draw == 8) { # Check Rai
```

```
lroad
      index = current_position # Determine which Railroad to send the player
to
      if (index < 5) {
        current_position = 5
      } else if (index < 15) {
        current_position = 15
      } else if (index < 25) {
        current_position = 25
      } else if (index < 35) {
        current_position = 35
      }
    } else if (current_deck_draw == 9) { # Check Utility
      index = current_position # Determine which Utility to send the player t
o
      if (index < 12) {
        current_position = 12
      } else {
        current_position = 28
      }
    } else if (current_deck_draw == 10) { # Check if go back 3 steps
      index = current_position # Determine whether the player needs to go bac
k over GO
      if (index >= 3) {
        index = index - 3
        current_position = index
      } else if (index == 2) {
        current_position = 39
      } else if (index == 1) {
        current_position = 38
      } else {
        current_position = 37
      }
    }
    current_deck = c(current_deck[-1], current_deck[1]) # Place card on the b
ottom
  }
  chance_result = list(current_position, current_deck) # Return both the deck
 and position
  return(chance_result)
} # Chance Deck cards

triple = function(current_position, turn, dice_vector) {
  if (turn >= 3) { # check the dice_vector, a T/F vector
    if (dice_vector[turn] & dice_vector[turn-1] & dice_vector[turn-2]) {
      current_position = 10
    }
  }
  return(current_position)
} # 3 Consecutive pairs
```

```
loop_39 = function(current_location) {
  return(current_location %% 40)
} # Loops at 39

jail_30 = function(current_position) {
  if (current_position == 30) {
    current_position = 10
  }
  return(current_position)
} # Sends player to jail if they land on 30

set.seed(141)
simulate_monopoly = function(n, d) {
  chance_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16) #I
nitialize chance deck
  chance <- sample(chance_cards) # Shuffle cards
  community_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)
 #Initialize community chest deck
  community <- sample(community_cards) # Shuffle cards

  die1 = sample(1:d, n, replace = T) # Roll die 1
  die2 = sample(1:d, n, replace = T) # Roll die 2
  dice = die1 + die2 # Combine dice into vector
  dice_vec = die1 == die2 # Create T/F vector for triples
  player_positions = rep(0, n)

  current_position = 0 # Initialize current position of player
  for (i in 1:n) {
    current_position = dice[i] + current_position # Update current position
    current_position = loop_39(current_position) # Loops at position 39

    current_position = triple(current_position = current_position, # Check fo
r three consecutive pairs
                              turn = i, dice_vector = dice_vec)

    chance_list = chance_deck(current_position = current_position, # Check fo
r chance deck
                              current_deck = chance)
    current_position = chance_list[[1]]
    chance = chance_list[[2]]

    cc_list = community_chest(current_position = current_position, # Check fo
r community chest
                              current_deck = community)
    current_position = cc_list[[1]]
    community = cc_list[[2]]

    current_position = jail_30(current_position = current_position) # Check i
f player is at 30
```

```
    player_positions[i] = current_position # Update player position tracker
  }
  player_positions = c(0, player_positions) # Include initial location 0
  return(factor(player_positions, 0:39)) # Return factorized version of posit
ions
} # Simulate game of monopoly

# 2

estimate_monopoly = function(n, d) {
  k = 1000 # Number of simulations

  position_list = lapply(1:k, function(x) simulate_monopoly(n, d)) # Generate
 k simulations of monopoly
  position_ratio = table(unlist(position_list)) # Create a table of the vecto
r
  position_prop = as.data.frame(prop.table(position_ratio)*100) # Generate pr
oportion table of the results
  beep(2)
  return(position_prop)
} # Perform estimations of the simulation

# Simulate monopoly games for 3, 4, 5, and 6 sided die
die6 = estimate_monopoly(10000, 6); die5 = estimate_monopoly(10000, 5)
die4 = estimate_monopoly(10000, 4); die3 = estimate_monopoly(10000, 3)

# Add number of sides column
die6$Sides = 6; die5$Sides = 5; die4$Sides = 4; die3$Sides = 3

dice_sides = rbind(die3, die4, die5, die6) # Combine die into single data fra
me

colnames(dice_sides) = c("Die", "Percentage", "Sides") # Give appropriate lab
els

dice_sides = dice_sides %>% # Arrange data frame
  group_by(Sides) %>%
  arrange(-Percentage,.by_group = TRUE)

top3_positions = dice_sides %>% # Determine the top 3 positions per side
  group_by(Sides) %>%
  top_n(n = 3, wt = Percentage)

line_plot = ggplot(data = dice_sides, aes(x = Die, y = Percentage, # Create l
ine plot of data
  group = Sides, col = as.factor(Sides))) + labs(x = "Player Positions", y =
"Percentage of Total",
      title = "Frequency of Board Positions", subtitle = "(according to numb
er of dice sides)") +
```

```
  theme(legend.justification=c(1,1), legend.position=c(1,1)) +
  theme(plot.title = element_text(hjust = 0.5), plot.subtitle = element_text
(hjust = 0.5))
line_plot + geom_line() + geom_point() + scale_color_manual(values=c("blue1",
 "brown", "yellow2", "green3"))

heat_map = ggplot(data = dice_sides, aes(x = factor(Sides), y = Die)) # Creat
e heatmap of data
heat_map + geom_tile(aes(fill = Percentage)) + labs(x = "Number of Sides", y
= "Board Position",
  title = "Heatmap of the Player Positions", subtitle = "(for 3, 4, 5, and 6-
sided die)") +
  scale_x_discrete(labels = c(3,4,5,6)) + scale_fill_gradient2(midpoint = 3)
+
  theme(plot.title = element_text(hjust = 0.5), plot.subtitle = element_text
(hjust = 0.5))

# 3

set.seed(141)
estimate_monopolySE = function(d) { # Estimate monopoly for the standard erro
r
  simulation = as.data.frame(table(simulate_monopoly(10000, d))) # 10,000 tur
ns
  simulation$Freq = simulation$Freq/10000 # Make into decimal frequency

  return(simulation)
}

jail_se = replicate(1000, estimate_monopolySE(d = 6)[11,2]) # Save the 10th p
osition for 1,000 iterations
sd(jail_se) # Obtain the standard error, 0.002078264

hist((jail_se-mean(jail_se))/sd(jail_se),
     main = "Z-scores", xlab = "Standard Deviation") # Observe the standard d
eviations and look for outliers
hist(jail_se)

### Part 2

# 1

rent = properties$Revenue # Payment for each color

charge_rent = function(current_position, rent, property_location) {
  if (current_position %in%  property_location) { # Check if player lands on
property
    bill = (-rent[which(property_location == current_position)]) # Less the c
orresponding bill
  } else {
```

```
    bill = 0
  }
  return(bill)
} # Charge rent for color properties

community_chest2 = function(current_position, current_deck) { # Community Che
st
  if (current_position == 2 || current_position == 17 || current_position ==
33) {
    current_deck_draw = current_deck[1] # Draw card
    if (current_deck_draw == 1) {
      current_position = 0 # sends player to GO
    } else if (current_deck_draw == 2) {
      current_position = 10 # sends player to Jail
    }

    current_deck = c(current_deck[-1], current_deck[1]) # Place card on the b
ottom
  }
  community_result = list(current_position, current_deck) # Return the deck,
position, and money
  return(community_result)
} # Community Chest cards

chance_deck2 = function(current_position, current_deck) {
  if (current_position == 7 || current_position == 22 || current_position ==
36) { # Chance
    current_deck_draw = current_deck[1] # Draw card
    if (current_deck_draw == 1) { # check if card drawn is GO or JAIL
      current_position = 0 # sends player to GO
    } else if (current_deck_draw == 2) {
      current_position = 10 # sends player to JAIL
    } else if (current_deck_draw == 3) {
      current_position = 11 # sends player to C1
    } else if (current_deck_draw == 4) {
      current_position = 24 # sends player to E3
    } else if (current_deck_draw == 5) {
      current_position = 39 # sends player to H2
    } else if (current_deck_draw == 6) {
      current_position = 5 # sends player to R1
    } else if (current_deck_draw == 7 | current_deck_draw == 8) { # Check Rai
lroad
      index = current_position
      if (index < 5) {
        current_position = 5
      } else if (index < 15) {
        current_position = 15
      } else if (index < 25) {
        current_position = 25
      } else if (index < 35) {
```

```
        current_position = 35
      }
    } else if (current_deck_draw == 9) { # Check Utility
      index = current_position
      if (index < 12) {
        current_position = 12
      } else {
        current_position = 28
      }
    } else if (current_deck_draw == 10) { # Check if go back 3 steps
      index = current_position
      if (index >= 3) {
        index = index - 3
        current_position = index
      } else if (index == 2) {
        current_position = 39
      } else if (index == 1) {
        current_position = 38
      } else {
        current_position = 37
      }
    }
    current_deck = c(current_deck[-1], current_deck[1]) # Place card on the b
ottom
  }
  chance_result = list(current_position, current_deck) # Return the deck, pos
ition, and money
  return(chance_result)
} # Chance Deck cards

tax_or_go = function(current_position) {
  cash = 0 # Allocate memory for cash
  if (current_position == 4) { # Check if current position is T1 (4, -$200) o
r T2 (38, -$100)
    cash = -200
  } else if (current_position == 38) {
    cash = -100
  } else if (current_position == 0) { # Check if current position is GO (0, +
$200)
    cash = 200
  }
  return(cash)
} # Tax or GO charges

pass_go = function(player_positions, turn) {
  if (turn > 2) {
    if ((player_positions[turn - 1] <= 39 & # Check if previous position was
below 0
         player_positions[turn - 1] >= 29)
        & # Check if current position is above 0
```

```
        (player_positions[turn] >= 1 & player_positions[turn] <= 11)
        & (player_positions[turn - 1] != 36)) {
      pay = 200 # Awards 200
    } else {
      pay = 0
    }
  } else {
    pay = 0
  }
  return(pay)
} # Award 200 for passing GO

bail = function(cash1, cash2, cash3, current_position) {
  if (current_position == 10) {
    cash1 = cash2 = cash3 = 0
  }
  return(list(cash1, cash2, cash3))
} # Award no money if player ends in jail

payday = function(cash1, cash2, cash3) {
  cash_sum = cash1 + cash2 + cash3 # Sum cash per roll
  if (cash_sum != 0) {
    pay = cash_sum
  } else {
    pay = 0
  }
  return(list(pay, 0))
} # Collect cash per roll

simulate_monopoly2 = function(n, d, property, cost) {
  # Return n+1 vector of positions, along with money gained or lost each turn
  chance_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16) #I
nitialize chance deck
  chance <- sample(chance_cards) # Shuffle cards
  community_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)
 #Initialize community chest deck
  community <- sample(community_cards) # Shuffle cards

  die1 = sample(1:d, n, replace = T) # Roll die 1
  die2 = sample(1:d, n, replace = T) # Roll die 2
  dice = die1 + die2 # Combine dice into vector
  dice_vec = die1 == die2 # Check for pairs
  player_positions = rep(0, n) # Allocate memory for player vector
  cash_flow = rep(0, n) # Allocate memory for cash vector

  current_position = 0 # Initialize current position
  for (i in 1:n) {
    current_position = dice[i] + current_position # Update position
    current_position = loop_39(current_position) # Check for loop
```

```
    current_position = triple(current_position = current_position, # Check for three consecutive pairs
                              turn = i, dice_vector = dice_vec)

    chance_list = chance_deck2(current_position = current_position, # Check for chance deck
                               current_deck = chance)
    current_position = chance_list[[1]] # Update current position
    chance = chance_list[[2]] # Update chance deck

    cc_list = community_chest2(current_position = current_position, # Check for community chest
                               current_deck = community)
    current_position = cc_list[[1]] # Update current position
    community = cc_list[[2]] # Update community chest

    cash1 = tax_or_go(current_position = current_position) # Check if Tax or GO

    cash2 = pass_go(player_positions = player_positions,
                    turn = i) # Award 200 for passing GO

    cash3 = charge_rent(current_position = current_position,
                        property_location = property,rent = cost) # Charge rent for a color

    current_position = jail_30(current_position = current_position) # Check if player is on G2J

    # Award no money if player ends in jail
    cash1 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_position = current_position)[[1]]
    cash2 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_position = current_position)[[2]]
    cash3 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_position = current_position)[[3]]

    player_positions[i] = current_position # Update position vector

    payment = payday(cash1 = cash1, cash2 = cash2, cash3 = cash3) # Sum cash per roll
    cash_flow[i] = payment[[1]] # Update cash vector
    cash1 = cash2 = cash3 = payment[[2]] # Reset cash values to 0 after each roll
  }

  player_positions = c(0, player_positions) # Include initial position
  cash_flow = c(0, cash_flow) # Include initial position
  simulation = list(player_positions, cash_flow) # Output list of position and cash
```

```r
  return(simulation)
} # Simulate monopoly 2

# 2

estimate_monopoly2 = function(n, d, k, property, cost) {
  #k = 1000 # Number of simulations
  property_bill = rep(0, k)
  position_list = lapply(1:k,
    function(x) simulate_monopoly2(n, d, property, cost)) # Generate k simula
tions of monopoly

  for (i in 1:k) {
    property_bill[i] = sum(position_list[[i]][[2]]) # Return the cash vector
from each simulation
  }

  return(property_bill)
} # Perform estimations of the simulation

color_simulations = function(n, d, k, property, cost) {
  color_levels = levels(property$Color) # Extract different levels of color f
rom data
  n_color = length(levels(property$Color)) # Use as variable for number of ma
trix columns
  color_cash = matrix(0, k, n_color) # Create matrix for the color results
  colnames(color_cash) = color_levels # Change matric column names

  for (i in 1:n_color) { # Obtain the k simulations for each color
    result = as.data.frame(estimate_monopoly2(n, d, k,
                property = property$Index[property$Color == color_levels[i]], c
ost = cost))
    result = as.numeric(result[,1])
    color_cash[,i] = result # Save results by color to matrix
  }

  return(color_cash)
} # Obtain the k simulations for each color

color_matrix = color_simulations(n = 100, d = 6, k = 1000, # Create color mat
rix
                                 property = properties, cost = rent)

# Create columns to mold into data frame
blue_vec = as.data.frame(color_matrix[,1]); green_vec = as.data.frame(color_m
atrix[,2])
lblue_vec = as.data.frame(color_matrix[,3]); orange_vec = as.data.frame(color
_matrix[,4])
pink_vec = as.data.frame(color_matrix[,5]); purple_vec = as.data.frame(color_
matrix[,6])
```

```r
red_vec = as.data.frame(color_matrix[,7]); yellow_vec = as.data.frame(color_m
atrix[,8])

# Create column for color
purple_vec$Color = "Purple"; lblue_vec$Color = "LBlue"; pink_vec$Color = "Pin
k"; orange_vec$Color = "Orange"
red_vec$Color = "Red"; yellow_vec$Color = "Yellow"; green_vec$Color = "Green
"; blue_vec$Color = "Blue"

change_names = function(color_data) {
  colnames(color_data) = c("Cash", "Color")
  return(color_data)
} # Add column names for Cash

purple = change_names(purple_vec); light_blue = change_names(lblue_vec) # Cre
ate appropriate column names
pink = change_names(pink_vec); orange = change_names(orange_vec); red = chang
e_names(red_vec)
yellow = change_names(yellow_vec); green = change_names(green_vec); blue = ch
ange_names(blue_vec)

color_frequency = rbind(purple, light_blue, pink, orange, red, yellow, green,
 blue) # Combine data
density_plot = ggplot(data = color_frequency, aes(x = Cash, color = Color)) #
 Create density plot
density_plot + geom_density() + theme(plot.title = element_text(hjust = 0.5),
 plot.subtitle = element_text(hjust = 0.5)) +
  labs(y = "Frequency of Cash Sum", x = "Cash", title = "Density Plot of Cash
 Sum", subtitle = "(according to colored opponents)") +
  scale_color_manual(values = c("blue", "green4", "dodgerblue1", "orange1", "
deeppink1", "purple1", "red1", "yellow3")) +
  theme(legend.justification=c(1,1), legend.position=c(1,1))

# 3

set.seed(141)

simulate_monopoly2b = function(n, d, property, cost) {
  # Return n+1 vector of positions, along with money gained or lost each turn
  chance_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16) #I
nitialize chance deck
  chance <- sample(chance_cards) # Shuffle cards
  community_cards <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)
 #Initialize community chest deck
  community <- sample(community_cards) # Shuffle cards

  die1 = sample(1:d, n, replace = T) # Roll die 1
  die2 = sample(1:d, n, replace = T) # Roll die 2
  dice = die1 + die2 # Combine dice into vector
  dice_vec = die1 == die2 # Check for pairs
```

```
  player_positions = rep(0, n) # Allocate memory for player vector
  cash_flow = rep(0, n) # Allocate memory for cash vector
  opponent_bill = rep(0, n) # Allocate memory for bill to opponent

  current_position = 0 # Initialize current position

  for (i in 1:n) {
    current_position = dice[i] + current_position # Update position
    current_position = loop_39(current_position) # Check for loop

    current_position = triple(current_position = current_position, # Check fo
r three consecutive pairs
                              turn = i, dice_vector = dice_vec)

    chance_list = chance_deck2(current_position = current_position, # Check f
or chance deck
                               current_deck = chance)
    current_position = chance_list[[1]] # Update current position
    chance = chance_list[[2]] # Update chance deck

    cc_list = community_chest2(current_position = current_position, # Check f
or community chest
                               current_deck = community)
    current_position = cc_list[[1]] # Update current position
    community = cc_list[[2]] # Update community chest

    cash1 = tax_or_go(current_position = current_position) # Check if Tax or
GO

    cash2 = pass_go(player_positions = player_positions,
                    turn = i) # Award 200 for passing GO

    cash3 = charge_rent(current_position = current_position,
                        property_location = property, rent = cost) # Charge r
ent for a color
    opponent_bill[i] = -cash3 # Money owed to opponent (changed to positive v
alue)

    current_position = jail_30(current_position = current_position) # Check i
f player is on G2J

    # Award no money if player ends in jail
    cash1 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_positio
n = current_position)[[1]]
    cash2 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_positio
n = current_position)[[2]]
    cash3 = bail(cash1 = cash1, cash2 = cash2, cash3 = cash3, current_positio
n = current_position)[[3]]

    player_positions[i] = current_position # Update position vector
```

```
    payment = payday(cash1 = cash1, cash2 = cash2, cash3 = cash3) # Sum cash
per roll
    cash_flow[i] = payment[[1]] # Update cash vector
    cash1 = cash2 = cash3 = payment[[2]] # Reset cash values to 0 after each
roll
  }

  player_positions = c(0, player_positions) # Include initial position
  cash_flow = c(5000, cash_flow) # Include initial position
  opponent_bill = c(0, opponent_bill) # Include initial position
  simulation = list(player_positions, cash_flow, # Output list of position, c
ash, opponent bill, and construction
                    opponent_bill)
  return(simulation)
} # Simulate monopoly 2b (also pays opponent)

estimate_monopoly2b = function(n, d, property, cost) {
  k = 1
  property_bill = opponent_bill = 0

  position_list = lapply(1:k, function(x) simulate_monopoly2b(n, d, property,
 cost)) # Generate k simulations of monopoly

  for (i in 1:k) {
    property_bill = property_bill + sum(position_list[[k]][[2]])
    opponent_bill = opponent_bill + sum(position_list[[k]][[3]])
  }

  return(list(property_bill, opponent_bill))
} # Performs one simulation

color_cash_balance = function(n, d, color_pairs, property_data, revenue) {
  primary = paste(color_pairs$color1) # Get colors for pairwise matchups
  secondary = paste(color_pairs$color2) # Second set of colors

  color1_PL_vector = rep(0, length(color_pairs)) # Allocate memory for color
1 vector
  color2_PL_vector = rep(0, length(color_pairs)) # Allocate memory for color
2 vector

  for (i in 1:length(primary)) { # Cycle through pairwise combinations and de
termine Profit and Loss for each
    primary_color = property_data$Index[property_data$Color == primary[i]] #
Retrieve the indices of color 1
    secondary_color = property_data$Index[property_data$Color == secondary
[i]] # Retrieve the indices of color 2

    # Inverse colors to determine Profit / Loss
    color1 = estimate_monopoly2b(n, d, property = secondary_color, cost = rev
```

```
enue) # Retrieve color 1's list
    color2 = estimate_monopoly2b(n, d, property = primary_color, cost = reven
ue) # Retrieve color 2's list

    color1_construction = sum(property_data$Cost[property_data$Color == prima
ry[i]]) # Color 1 construction cost
    color2_construction = sum(property_data$Cost[property_data$Color == secon
dary[i]]) # Color 2 construction cost

    color1_PL = sum(color1[[1]]) + sum(color2[[2]]) - color1_construction # D
etermine P/L for color 1
    color2_PL = sum(color2[[1]]) + sum(color1[[2]]) - color2_construction # D
etermine P/L for color 2

    color1_PL_vector[i] = color1_PL # Save color 1 results to vector
    color2_PL_vector[i] = color2_PL # Save color 2 results to vector
  }
  return(list(color1_PL_vector, color2_PL_vector))
} # Simulate all pairwise games

simulation_100 = function(n) {
  simulation_matrix = matrix("", ncol=B, nrow=28)
  B = 100
  for (i in 1:B) {
    result = color_cash_balance(n,6,colors,properties,rent)
    simulation_matrix[,i] = sapply(1:28, function(x) ifelse(result[[1]][x] ==
 result[[2]][x], NA,
                            colors_character[x,(result[[1]][x] < result[[2]]
[x])+1]))
  }
  return(simulation_matrix)
} # Create 100 simulation of matchups for given color pairs

games_25 = simulation_100(25) # Simulate n = 25, 50, and 100 turns for k = 10
0 times
games_50 = simulation_100(50)
games_100 = simulation_100(100)

table(games_25)/(7*B)*100 # Set up a table of the results, and determine the
win rate %
table(games_50)/(7*B)*100
table(games_100)/(7*B)*100
```