Project1 Statistical Computation

Eyal Grinberg & Yam Rozen

2023-04-01

Q1 - Ladders & Snakes

An auxiliary function for summing digits.

```
# the function receives a number and returns the final sum of it's digits.

calc_digits_sum <- function(num) {
  digits <- as.numeric(strsplit(as.character(num), "")[[1]]) # from chatGPT: the function splits the number into its individual digits using the strsplit() function. It then converts each digit to a numeric type using the as.numeric() function.
  curr_sum <- sum(digits)
  if (curr_sum > 9) { # there's need to sum digits again
    return(calc_digits_sum(curr_sum)) #recursion
  }
  return(curr_sum)
}
```

data structures:

```
# for checking ladders
base_of_a_ladder_vec <- c(2,7,8,15,21,28,36,51,71,78,87)
top_of_a_ladder_vec <- c(38,14,31,26,42,84,44,67,91,98,94)
# for checking snakes
head_of_a_snake_vec <- c(16,46,49,62,64,74,89,92,95,99)
tail_of_a_snake_vec <- c(6,25,11,19,60,53,68,88,75,80)
```

An auxiliary function for checking if a square is a head of a snake or a base of a ladder.

```
# the function receives a vector and a number and returns the index of the number in the vector if it exists, -1 otherwise.

find_index_in_vec <- function(vec, num) {
  index <- which(vec == num)
  if(length(index) == 0) {
    return(-1)
  }
  return(index)
}
```

A function for one turn in the game.

```
# receives the starting square as an argument.
# returns the end square.

perform_one_turn_Q1 <- function(start_square) {
  dice_roll <- sample(1:6, 1)
  mid_square <- dice_roll + start_square
  # check for exceeding 100
  if (mid_square > 100) {
    mid_square <- 100 - (mid_square - 100)
  }
  # check sum digits condition
  if (calc_digits_sum(mid_square) == dice_roll) {
    mid_square <- floor(mid_square / 2)
  }
  # check for a snake
  index <- find_index_in_vec(head_of_a_snake_vec, mid_square)
  if (index != -1) {
    mid_square <- tail_of_a_snake_vec[index]
  }
  # check for a ladder
  index <- find_index_in_vec(base_of_a_ladder_vec, mid_square)
  if (index != -1) {
    mid_square <- top_of_a_ladder_vec[index]
  }
  # at this point 'mid_square' is actually the end square of this turn.
  return(mid_square)
}
```

the main function for performing a full game and counting the number of dice rolls.

```
perform_one_game_Q1 <- function() {
  cnt <- 0
  start_square <- 1
  end_square <- -1
  while (end_square != 100) {
    end_square <- perform_one_turn_Q1(start_square)
    start_square <- end_square
    cnt <- cnt + 1
  }
  return (cnt)
}
```

auxiliary function for CI calculation:

```
calc_CI <- function(data, conf) {
  alpha <- 1 - conf
  q_z <- qnorm(1 - alpha/2)
  n <- length(data)
  avg <- mean(data)
  sd <- sd(data)
  upper <- avg + q_z * (sd / sqrt(n))
```

```
  lower <- avg - q_z * (sd / sqrt(n))
  CI <- c(lower, upper)
  names(CI) <- c("Lower", "Upper")
  return (CI)
}
```

5000 games simulation:

```
sim_results_Q1 <- replicate(5000, perform_one_game_Q1())
```

Mean and CI calculations:

```
round(mean(sim_results_Q1), 3) # 3 digits after the decimal point as
requested

## [1] 132.283

round(calc_CI(sim_results_Q1, 0.95), 3)

##   Lower   Upper
## 128.908 135.658
```

Q2 - Tic Tac Toe

```
# auxiliary function that checks if any of the players won after a move was
made.
# the input is the the last square that was played by one of the players as
well as the board and it's size and the player.
# the function returns 1 if X player has won.
# the function returns 2 if O player has won.

check_win <- function(player, board, size, square) {
  # extract the relevant row and col number
  row_index_of_square <- ceiling(square / size)
  col_index_of_square <- -1
  modulu_res <- square %% size
  if (modulu_res) {
    col_index_of_square <- modulu_res
  }
  else { # the column of square is a non-remainder multiple of size --->
should be the last column in the matrix.
    col_index_of_square <- size
  }

  col_match <- row_match <- primary_diag_match <- secondary_diag_match <- 0
  # check rows and columns
  for (i in 1:size) {
    if (row_match == -1 && col_match == -1) {break} # there's no chance for a
row/col match, stop searching.
    if (row_match != -1) { # a little optimization, if we found a square in
the row that doesn't match, we won't keep checking that row.
      if (board[row_index_of_square, i] == player) {row_match <- row_match +
```

```
1}
      else {row_match <- -1}
    }
    if (col_match != -1) { # same optimization for columns
      if (board[col_index_of_square, i] == player) {col_match <- col_match +
1}
      else {col_match <- -1}
    }
  }

  # if the square is not on any of the diagonals, we only need to check the
row and the col of the given square. otherwise we need to check also the
diagonals.
  if (row_index_of_square == col_index_of_square ||
      col_index_of_square == size - row_index_of_square + 1) { # check if the
square is on one of the two diagonals.
  # same optimization for diagonals too.
    for (i in 1:size) {
      if (primary_diag_match == -1 && secondary_diag_match == -1) {break}
      if (primary_diag_match != -1) {
        if (board[i, i] == player) {primary_diag_match <- primary_diag_match
+ 1}
        else {primary_diag_match <- -1}
      }
      if (secondary_diag_match != 1) {
        if (board[i, size - i + 1] == player) {secondary_diag_match <-
secondary_diag_match + 1}
        else {secondary_diag_match <- -1}
      }
    }
  }

  if (row_match == size || col_match == size ||
      primary_diag_match == size || secondary_diag_match == size) {
    return (player) # should return 1 or 2, depends on which player played
this turn
  }
  return(-1)
}
```

One game function

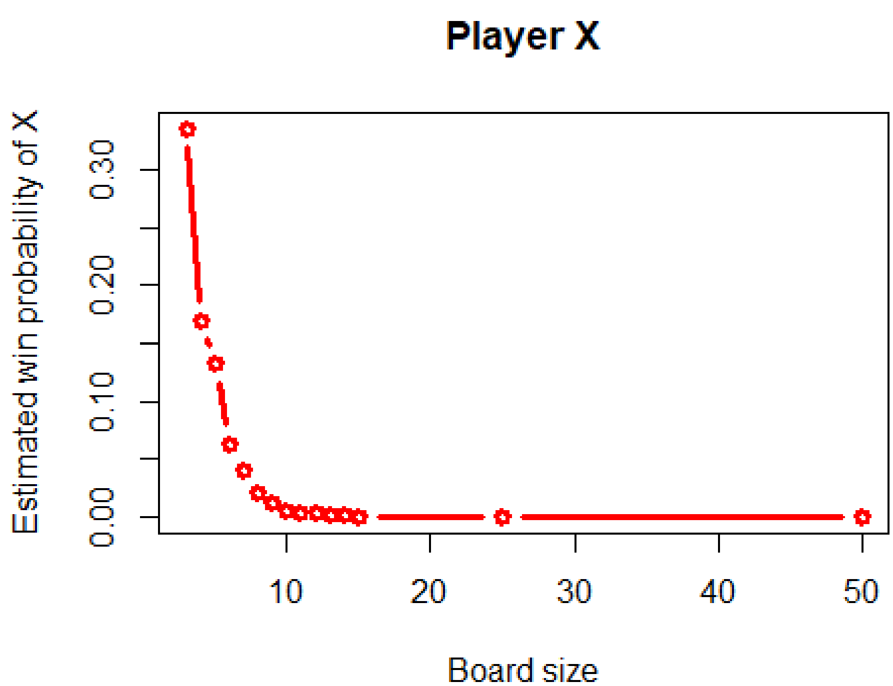
```
# the function returns 1 if X player has won, 2 if O player has won, and 0 if
it's a tie

perform_one_game_Q2 <- function(size) {

  board <- matrix(-1, size, size)
  permuted_squares <- sample(c(1:(size^2))) # here all the "magic" happens,
we created a permutation of the squares and each turn take a square
```

```
sequentially from that vector instead of sampling in each iteration.

  i <- 1
  while (i <= size^2) {
    # X player's turn
    board[ permuted_squares[i] ] <- 1
    if (i >= 2 * size) { # need to check win condition only after size turns
were played by each player
      win_check_res <- check_win(1, board, size, permuted_squares[i])
      if (win_check_res != -1) { # game over
        return(win_check_res)
      }
    }
    i <- i + 1
    if (i == size^2 + 1) {break} # if there's an odd number of squares the X
player will play one more turn.
    # O player's turn
    board[ permuted_squares[i] ] <- 2
    if (i >= 2 * size) { # need to check win condition only after 'size'
turns were played
      win_check_res <- check_win(2, board, size, permuted_squares[i])
      if (win_check_res != -1) { # game over
        return(win_check_res)
      }
    }
    i <- i + 1
  }
  # otherwise it's a tie
  return(0)
}
```

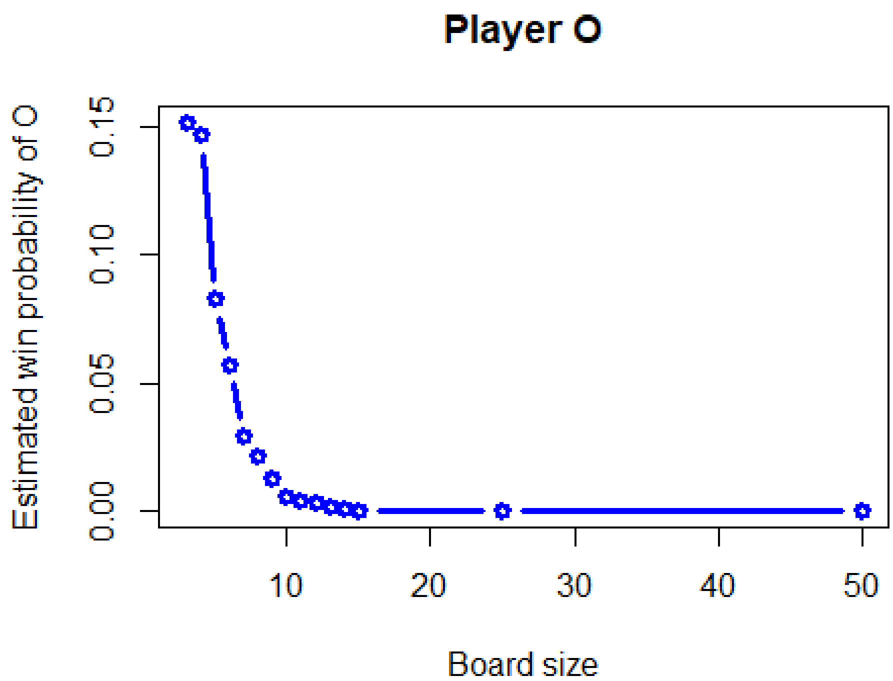
simulations:

```
sim_size <- 5000
# matrix initialization
prob_mat <- matrix(nrow = 3, ncol = 15)
row.names(prob_mat) <- c("Win probability X","Win probability O","Tie
probability")
colnames(prob_mat) <-
c('3x3','4x4','5x5','6x6','7x7','8x8','9x9','10x10','11x11','12x12','13x13','
14x14','15x15','25x25','50x50')
# performing simulations
for (i in 3:15) {
  results_game_size_i <- replicate(sim_size, perform_one_game_Q2(i))
  prob_mat[1,i-2] = mean(results_game_size_i == 1) # games that X has won
returned 1
  prob_mat[2,i-2] = mean(results_game_size_i == 2) # games that X has won
returned 2
  prob_mat[3,i-2] = mean(results_game_size_i == 0) # games that X has won
returned 0
```

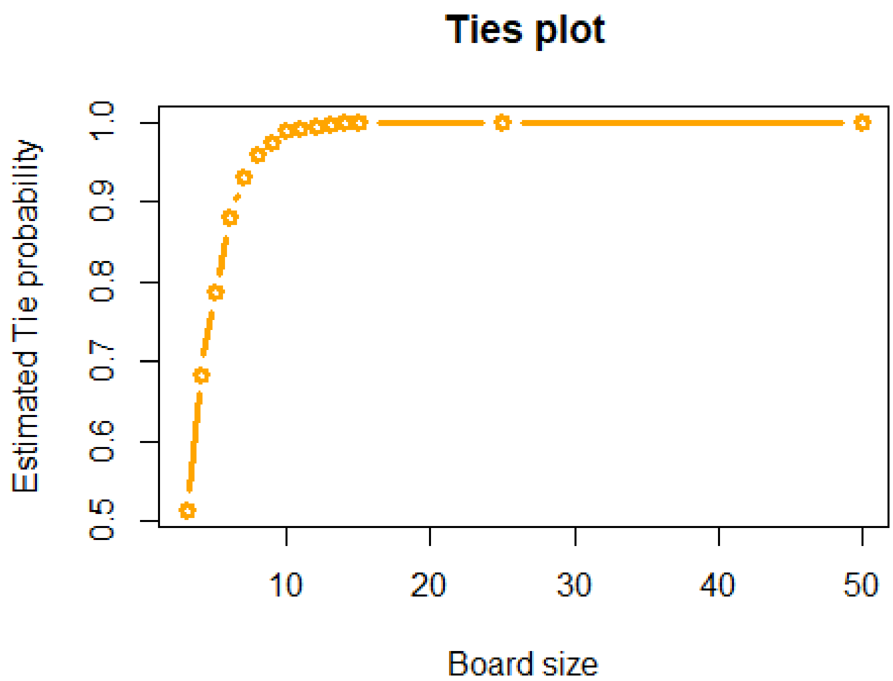
```
}
for (i in 1:2) {
  results_game_size_i <- replicate(sim_size, perform_one_game_Q2(i*25))
  prob_mat[1,i+13] = mean(results_game_size_i == 1) # games that X has won
returned 1
  prob_mat[2,i+13] = mean(results_game_size_i == 2) # games that X has won
returned 2
  prob_mat[3,i+13] = mean(results_game_size_i == 0) # games that X has won
returned 0
}
prob_mat

##                        3x3    4x4    5x5    6x6    7x7    8x8    9x9  10x10
## Win probability X 0.3350 0.1696 0.1316 0.0628 0.0394 0.0200 0.0128 0.0054
## Win probability O 0.1516 0.1468 0.0826 0.0568 0.0290 0.0212 0.0128 0.0056
## Tie probability   0.5134 0.6836 0.7858 0.8804 0.9316 0.9588 0.9744 0.9890
##                      11x11  12x12  13x13  14x14  15x15 25x25 50x50
## Win probability X 0.0038 0.0030 0.0012 0.0010 0.0002     0     0
## Win probability O 0.0042 0.0032 0.0012 0.0004 0.0000     0     0
## Tie probability   0.9920 0.9938 0.9976 0.9986 0.9998     1     1

board_size <- c(3:15,25,50)
plot(x = board_size, y = prob_mat[1,], xlab = "Board size",
     ylab = "Estimated win probability of X", col = "red", main = "Player
X",lwd = 3, type = "b")
```



```
plot(x = board_size, y = prob_mat[2,], xlab = "Board size",
     ylab = "Estimated win probability of O", col = "blue", main = "Player
O",lwd = 3, type = "b")
```




```
plot(x = board_size, y = prob_mat[3,], xlab = "Board size",
     ylab = "Estimated Tie probability", col = "orange", main = "Ties
plot",lwd = 3, type = "b")
```



As shown in the plots, it seems that as the board size increases the tie probability increases too, and the probabilities that any of the two players will win go down towards 0. the intuition for it is that on a larger board the majority of the square permutations will end in a tie.

Bonus Section - performing 5,000,000 simulations uncomment to run the simulations

```
# # Set the number of simulations
# n_sims <- 5000000
# sim_result_size_25 <- vector("numeric", n_sims)
# sim_result_size_50 <- vector("numeric", n_sims)
#
# # Iterate over the simulations
# for (i in 1:n_sims) {
#   # Run the function for each simulation
#   sim_result_size_25[i] <- perform_one_game_Q2(25)
#   sim_result_size_50[i] <- perform_one_game_Q2(50)
#
#   # Check if the current iteration is a multiple of 50,000
#   if (i %% 50000 == 0) {
#     # Print a message to indicate the progress of the simulation
#     cat(i / 50000,"% of the simulations completed. \n")
#   }
# }
# ```
# Probabilities
# ```{r}
```

```
# # matrix initialization
# prob_mat <- matrix(nrow = 3, ncol = 2)
# row.names(prob_mat) <- c("Win probability X","Win probability O","Tie
probability")
# colnames(prob_mat) <- c('25x25','50x50')
#
# # performing simulations
# prob_mat[1,1] = mean(sim_result_size_25 == 1) # games that X has won
returned 1
# prob_mat[2,1] = mean(sim_result_size_25 == 2) # games that X has won
returned 2
# prob_mat[3,1] = mean(sim_result_size_25 == 0) # games that X has won
returned 0
#
# prob_mat[1,2] = mean(sim_result_size_50 == 1) # games that X has won
returned 1
# prob_mat[2,2] = mean(sim_result_size_50 == 2) # games that X has won
returned 2
# prob_mat[3,2] = mean(sim_result_size_50 == 0) # games that X has won
returned 0
#
# prob_mat
```

Plots

```
# board_size <- c(25,50)
# plot(x = board_size, y = prob_mat[1,], xlab = "Board size",
#      ylab = "Estimated win probability of X", col = "red", main = "Player
X",lwd = 3, type = "b")
# plot(x = board_size, y = prob_mat[2,], xlab = "Board size",
#      ylab = "Estimated win probability of O", col = "blue", main = "Player
O",lwd = 3, type = "b")
# plot(x = board_size, y = prob_mat[3,], xlab = "Board size",
#      ylab = "Estimated Tie probability", col = "orange", main = "Ties
plot",lwd = 3, type = "b")
```

As expected, for 5,000,000 simulations the estimators for the win probabilities of any player is 0 and for the tie probability is 1.

When we worked on the project we succeeded performing the simulation, but when we tried to knit to PDF before submitting, we found out that it would take too much time and we would miss the due date. Therefore, we commented out only the bonus part and submit it that way. Obviously, just by looking at the code it's clear that it won't take more than a few hours to run.

compu_proj 3

2023-04-19

```
my_simu <- function(capacity, dogs_service_rate, cats_service_rate) {
dogs_arrive = c()
cats_arrive = c()
dogs_service = c()
cats_service = c()
while(length(cats_arrive) == 0 || cats_arrive[length(cats_arrive)] < 12 * 60)
{
cat_arriving <- rexp(1, 1.5)
if(length(cats_arrive) == 0) {
cats_arrive = append(cats_arrive, cat_arriving)
}
else {
cats_arrive = append(cats_arrive, cat_arriving + cats_arrive[length(cats_arrive)])
}
}
cats_service = rexp(length(cats_arrive),cats_service_rate)
while(length(dogs_arrive) == 0 || dogs_arrive[length(dogs_arrive)] < 12 * 60)
{
dog_arriving <- rexp(1, 3)
if(length(dogs_arrive) == 0) {
dogs_arrive = append(dogs_arrive, dog_arriving)
}
else {
dogs_arrive = append(dogs_arrive, dog_arriving + dogs_arrive[length(dogs_arrive)])
}
}
dogs_service = rexp(length(dogs_arrive),dogs_service_rate)
i_cats = 1
i_dogs = 1
cared_dogs = 0
cared_cats = 0
rejected_dogs = 0
rejected_cats = 0
total_time = 0
points = 0
que = c()
que_kinds = c()
que_nums = c(0)
que_times = c(0)
while(total_time < (12 * 60)) {
# Loop of creating queue and rejecting:
while (min(cats_arrive[i_cats],dogs_arrive[i_dogs]) < total_time) {
if (cats_arrive[i_cats] <= dogs_arrive[i_dogs]) {
if (length(que) == 0) {
```

```
que = append(que,cats_service[i_cats])
que_kinds = append(que_kinds,"cat")
que_nums = append(que_nums, length(que))
que_times = append(que_times, total_time)
}
else {
rejected_cats = rejected_cats + 1
}
i_cats = i_cats + 1
}
else {
if (length(que) < capacity) {
que = append(que,dogs_service[i_dogs])
que_kinds = append(que_kinds,"dog")
que_nums = append(que_nums, length(que))
que_times = append(que_times, total_time)
# loss_points = loss_points + 1
}
else {
rejected_dogs = rejected_dogs + 1
}
i_dogs = i_dogs + 1
}
}
# Dealing the queue:
if (length(que) > 0 && total_time + que[1] < 12 * 60) {
total_time = total_time + que[1]
if(que_kinds[1] == "cat") {
cared_cats = cared_cats + 1
}
else {
cared_dogs = cared_dogs + 1
}
que = que[-1]
que_kinds = que_kinds[-1]
que_nums = append(que_nums, length(que))
que_times = append(que_times, total_time)
}
else {
# Dealing the minimum:
if (cats_arrive[i_cats] < dogs_arrive[i_dogs]) {
total_time = cats_arrive[i_cats] + cats_service[i_cats]
cared_cats = cared_cats + 1
i_cats = i_cats + 1
}
else {
total_time = dogs_arrive[i_dogs] + dogs_service[i_dogs]
cared_dogs = cared_dogs + 1
i_dogs = i_dogs + 1
}
}
}
points = (cared_dogs) + (cared_cats * 3) - (rejected_dogs * 0.1)
```

```
result <-c(cared_dogs, cared_cats, rejected_dogs, rejected_cats,(sum(que_nums * que_times) / sum(que_ti

return(result)
}
```

A

```
m <- replicate(100,my_simu(10, 3, 5))
cat("Cared Dogs: ", mean(m[1,]),"\nCared Cats: ", mean(m[2,]),
"\nRejected Dogs: ", mean(m[3,]), "\nRejected Cats: ", mean(m[4,]),
"\nLine Length: ", mean(m[5,]), "\nProfit: ", mean(m[6,]))
```

```
## Cared Dogs:  1974.21
## Cared Cats:  129.6
## Rejected Dogs:  185.78
## Rejected Cats:  946.8
## Line Length:  4.857973
## Profit:  2344.432
```

B b1 # disadvantge is that thers more chance for cat to reject , bur cat pay more then dog.

```
m <- replicate(100,my_simu(20, 3, 5))
cat("Cared Dogs: ", mean(m[1,]),"\nCared Cats: ", mean(m[2,]),
"\nRejected Dogs: ", mean(m[3,]), "\nRejected Cats: ", mean(m[4,]),
"\nLine Length: ", mean(m[5,]), "\nProfit: ", mean(m[6,]))
```

```
## Cared Dogs:  2055.13
## Cared Cats:  73.66
## Rejected Dogs:  98.56
## Rejected Cats:  1008.08
## Line Length:  9.91432
## Profit:  2266.254
```

b2

```
m <- replicate(100,my_simu(10, 3.3, 5.5))
cat("Cared Dogs: ", mean(m[1,]),"\nCared Cats: ", mean(m[2,]),
"\nRejected Dogs: ", mean(m[3,]), "\nRejected Cats: ", mean(m[4,]),
"\nLine Length: ", mean(m[5,]), "\nProfit: ", mean(m[6,]))
```

```
## Cared Dogs:  2050.54
## Cared Cats:  205.24
## Rejected Dogs:  107
## Rejected Cats:  880.35
## Line Length:  4.062504
## Profit:  2655.56
```

b3

```
my_simu_with_turtles <- function(capacity, dogs_service_rate, cats_service_rate) {
dogs_arrive = c()
cats_arrive = c()
dogs_service = c()
cats_service = c()
while(length(cats_arrive) == 0 || cats_arrive[length(cats_arrive)] < 12 * 60)
{
cat_arriving <- rexp(1, 1.5)
if(length(cats_arrive) == 0) {
cats_arrive = append(cats_arrive, cat_arriving)
}
else {
cats_arrive = append(cats_arrive, cat_arriving + cats_arrive[length(cats_arrive)])
}
}
cats_service = rexp(length(cats_arrive),cats_service_rate)
while(length(dogs_arrive) == 0 || dogs_arrive[length(dogs_arrive)] < 12 * 60)
{
dog_arriving <- rexp(1, 3)
if(length(dogs_arrive) == 0) {
dogs_arrive = append(dogs_arrive, dog_arriving)
}
else {
dogs_arrive = append(dogs_arrive, dog_arriving + dogs_arrive[length(dogs_arrive)])
}
}
dogs_service = rexp(length(dogs_arrive),dogs_service_rate)
i_cats = 1
i_dogs = 1
cared_dogs = 0
cared_cats = 0
rejected_dogs = 0
rejected_cats = 0
total_time_cats = 0
total_time_dogs = 0
points = 0
que_dogs = c()
que_cats = c()
que_dogs_nums = c(0)
que_cats_nums = c(0)
que_dogs_times = c(0)
que_cats_times = c(0)
while(total_time_cats < (12 * 60)) {
# Loop of creating queue and rejecting:
while (cats_arrive[i_cats] < total_time_cats) {
if (length(que_cats) == 0) {
que_cats = append(que_cats,cats_service[i_cats])
que_cats_nums = append(que_cats_nums, length(que_cats))
que_cats_times = append(que_cats_times, total_time_cats)
}
  else {
rejected_cats = rejected_cats + 1
}
```

```
i_cats = i_cats + 1
}
# Dealing the queue:
if (length(que_cats) > 0 && total_time_cats + que_cats[1] < 12 * 60) {
total_time_cats = total_time_cats + que_cats[1]
cared_cats = cared_cats + 1
que_cats = que_cats[-1]
que_cats_nums = append(que_cats_nums, length(que_cats))
que_cats_times = append(que_cats_times, total_time_cats)
}
else {
# Dealing the minimum:
total_time_cats = cats_arrive[i_cats] + cats_service[i_cats]
cared_cats = cared_cats + 1
i_cats = i_cats + 1
}
}
while(total_time_dogs < (12 * 60)) {
# Loop of creating queue and rejecting:
while (dogs_arrive[i_dogs] < total_time_dogs) {
if (length(que_dogs) < capacity) {
que_dogs = append(que_dogs,dogs_service[i_dogs])
que_dogs_nums = append(que_dogs_nums, length(que_dogs))
que_dogs_times = append(que_dogs_times, total_time_dogs)
}
else {
rejected_dogs = rejected_dogs + 1
}
i_dogs = i_dogs + 1
}
# Dealing the queue:
if (length(que_dogs) > 0 && total_time_dogs + que_dogs[1] < 12 * 60) {
total_time_dogs = total_time_dogs + que_dogs[1]
cared_dogs = cared_dogs + 1
que_dogs = que_dogs[-1]
que_dogs_nums = append(que_dogs_nums, length(que_dogs))
que_dogs_times = append(que_dogs_times, total_time_dogs)
}
else {
# Dealing the minimum:
total_time_dogs = dogs_arrive[i_dogs] + dogs_service[i_dogs]
cared_dogs = cared_dogs + 1
i_dogs = i_dogs + 1
}
}
points = (cared_dogs) + (cared_cats * 3) - (rejected_dogs * 0.1)
result <-c(cared_dogs, cared_cats, rejected_dogs, rejected_cats,
(sum(que_dogs_nums * que_dogs_times) / sum(que_dogs_times)),(sum(que_cats_nums * que_cats_times) / sum(

return(result)
}
m <- replicate(100,my_simu_with_turtles(10, 1, 5/3))
cat("Cared Dogs: ", mean(m[1,]),"\nCared Cats: ", mean(m[2,]),
```

```r
"\nRejected Dogs: ", mean(m[3,]), "\nRejected Cats: ", mean(m[4,]),
"\nDogs Line Length: ", mean(m[5,]), "\nCats Line Length: ", mean(m[6,]),
"\nProfit: ", mean(m[7,]))
```

```
## Cared Dogs:  717.07
## Cared Cats:  754.82
## Rejected Dogs:  1427.89
## Rejected Cats:  320.15
## Dogs Line Length:  8.995961
## Cats Line Length:  0.4999447
## Profit:  2838.741
```

option 3 is the best option depend only the profit , if more elements like racism against dog is important so we will consider again.