Machine Problem 1

Bascug, Ethan Job
Dy, Alwyn
Fortaleza, Camelle Faye
Portuito, Rey Joseph

PROBLEM 1

(BIRTHDAYS) Ignoring leap days, the days of the year can be numbered 1 to 365. Assume that birthdays are equally likely to fall on any day of the year. Consider a group of n people, of which you are not a member. Suppose some two people in the group share a birthday. Estimate the smallest n for which the probability that two people in the group share a birthday is greater than 90%. For this problem, let the number of trials be 10000.

A Algorithm

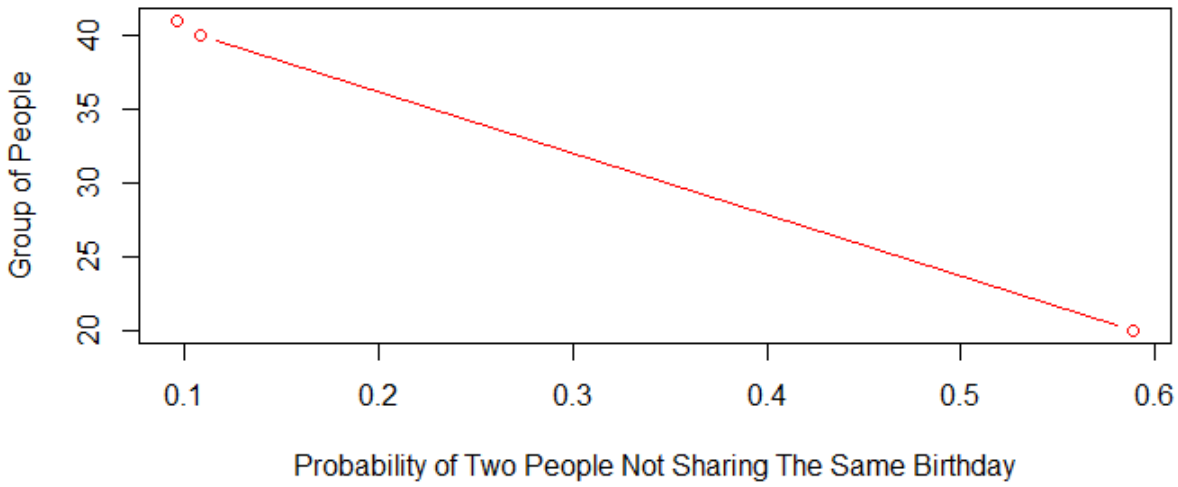
1. Set an estimation on the number of people and compute the probability that none of them share the same birthday.
2. Deduct the result to 1 to acquire the probability that two of the estimated number of people share the same birthday.
3. Check if the probability is greater than 90%.
4. If not, add more people and repeat 1 and 2, until the probability is greater than 90%.

B R Code

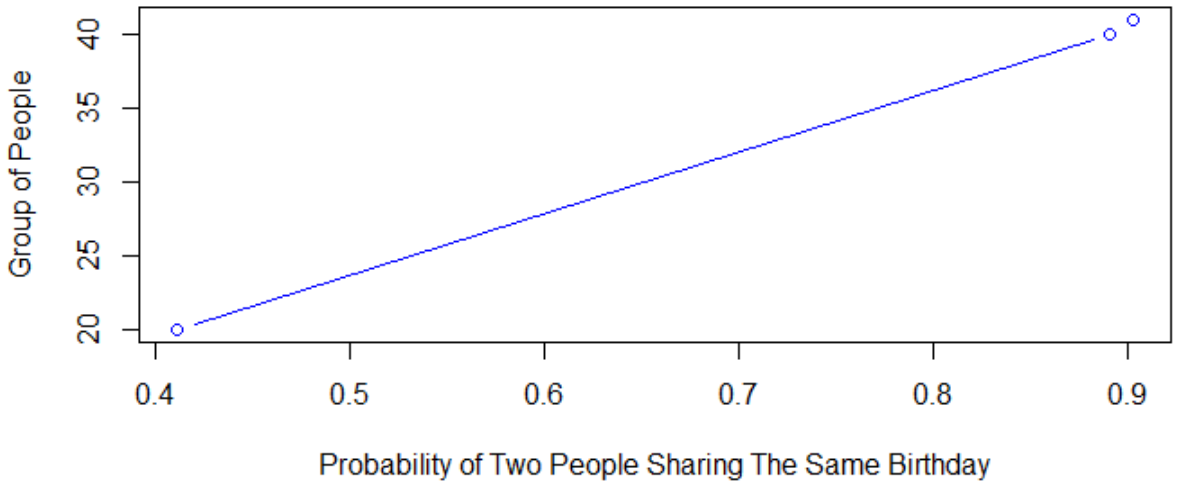
```
  #(1)Parameters: Smallest number of people (n) for which the probability that the
     two people in the group share a birthday is greater than 90%.

   #(2)to get the probability of two people with the same birthday in n people.
   #(3)find the probability of having no people (m) shared the same birthday
m1_20<-c(365/365*364/365*363/365*362/365*
         361/365*360/365*359/365*358/365*
         357/365*356/365*355/365*354/365*
         353/365*352/365*351/365*350/365*
         349/365*348/365*347/365*346/365)
                              #(4)A group of of 20 people has m1_20=0.5885616 to get
                                 "n1_20" we subtract "m1_20" to 1,
n1_20<-1-m1_20                #(5)Since "m" is defined in #(3) as no people that
    shares the same birthday
                                                        #(6)n1_20 < 90%

m1_40<-c(365/365*364/365*363/365*362/365* #(7) we add 20 more people, since n1_20 <
    90% and we need n > 90%
```

```
         361/365*360/365*359/365*358/365* #(8) Let m= m1_40 and n= n1_n40, Follow the
            process we did at (5)
         357/365*356/365*355/365*354/365*
         353/365*352/365*351/365*350/365*
         349/365*348/365*347/365*346/365*
         345/365*344/365*343/365*342/365*
         341/365*340/365*339/365*338/365*
         337/365*336/365*335/365*334/365*
         333/365*332/365*331/365*330/365*
         329/365*328/365*327/365*326/365)

n1_40<-1-m1_40 #(9)n1_20 < 90%, about 0.0087682 less
               #(10) we add 1 more person to m people, since we are relatively
                  getting closer to 90%
               #(11)Let m= m1_41 and n= n1_41, Follow the process we did at (5) and
                  (8)

m1_41<-c(365/365*364/365*363/365*362/365*
        361/365*360/365*359/365*358/365*
        357/365*356/365*355/365*354/365*
        353/365*352/365*351/365*350/365*
        349/365*348/365*347/365*346/365*
        345/365*344/365*343/365*342/365*
        341/365*340/365*339/365*338/365*
        337/365*336/365*335/365*334/365*
        333/365*332/365*331/365*330/365*
        329/365*328/365*327/365*326/365*
        325/365)

n1_41<-1-m1_41              #(12)n1_41 = 0.903151, then n > 90% about 0.003151

 m<-cbind(m1_20,m1_40,m1_41) #combine same vectors for plotting
 n<-cbind(n1_20,n1_40,n1_41)
 Gr_people<-cbind(20,40,50)

 plot(n,Gr_people ,
      xlab ="Probability of Two People Sharing The Same Birthday",
      ylab="Group of People",
      type="b",
      col="blue")
 plot(m,Gr_people ,
      xlab ="Probability of Two People Not Sharing The Same Birthday",
      ylab="Group of People",
      type="b",
      col="red")

 #The smallest n for which the probability that two people in the group share a
    birthday is greater than 90% is 41, since n41>90% about 0.003151.
```

C Plots



Plot 1.1 Examines how the probability of two people not sharing the same birthday is being influnced by the number of people in a group. See that as the number of people in a group rises the probability of two people not sharing the same birthday gradually decreases.



Plot 1.2 (above) Shows how the probability of two people sharing the same birthday is affected by the number of people in each group. As the number of people in a group increases the probability of two people sharing the same birthday also increases.

D **Mathematical Calculations**

Even though we input this equation to a calculator because the chance is that it would display "Math Error" or "Syntax Error", but to formality the equation we used to calculate the probability of two people sharing the same birthday is:

$$\frac{364!}{365-n!}365^{n-1}$$

Unfortunately, our calculator cannot perform such equation, so there's no other choice but go on the long way; by multiplying the probability of two people not sharing the same birthday in a group of people (m) and the result is deducted to 1 to acquire the probability of two people sharing the same birthday in a group of people (n).

Since, $p(m) + p(n) = 1$ then $p(n) = 1 - p(m)$
So, $p(m) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \ldots \frac{m}{365}$

PROBLEM 3

(COLOR GAME) The color game is a popular betting game here in the Philippines. Three colored dice are being tossed and the player will bet money to one of the 6 colors. A player wins the same amount, twice the amount, or three times the amount of money he bet if the color he chose matches the colors from the 3 dice that has been rolled. Suppose that a player only bets one color for every round. Simulate this game and show that a player will eventually lose his money.

A Algorithm

The game only allows the player to bet on a single color per round, this greatly reduces the probability of having a positive net money after playing the game. However, maximizing the chance of a player not losing all their money allows us to have a solid proof that they will indeed lose all of their money after enough rounds of playing the color game. To do this, we let them bet only a portion of their initial money for each 'round; this amount can be obtained by multiplying their initial money by a betting percentage. Furthermore, since the exact color is not important, we represent the 6 colors using integers 0 to 5.

1. Input the initial money and the betting percentage per round. This parameter accepts floating-point values from 0 to 1. Additionally, a player can also input -1, this means that their betting percentage will be randomized every round.
2. *Round start:* Add 1 to round counter.
3. Calculate betting money by multiplying their balance with the betting percentage. Subtract the betting money from the balance.
4. A color is randomly selected to be the bet color.
5. *Roll 3 dice:* A set of 3 colors are randomly chosen from a total of 18 (6*3) possible colors, each color is equally likely to be selected.
6. Count how many of the 3 rolled colors match the bet color.
7. *Compute winnings:* If there are matched colors, the bet money is doubled, tripled, or quadrupled, depending on the number of matched colors. The winnings will be added to the balance.
8. If the balance is non-zero, steps 2 to 8 are repeated. However, if the balance is 0, the game ends.
9. Return the game summary, where the trials and the player's betting money, chosen color, rolled colors, and match colors for each round are noted.

B R Code

```
#PARAMETERS: initial money of the player and the percentage of their money they are
    willing to bet. betting percentage can be -1, randomize betting percentage for
    every round
colorGame <- function(balance, betting_percentage){
  round <- 0 # store the number of rounds
  colors <- c(0:5) # represents the 6 colors per die
  chance <- c(rep(1/6, 6)) # probabilities of each color, future-proofed in case of
      biased die
  flag <- 0 # value of flag will be 1 if betting percentage is randomized, else 0
  if (betting_percentage==-1) flag <- 1
```

```
8
9   set.seed(34029) # seed allows for replication of the same output
10  gameSummary <- data.frame( # a blank data frame to tabulate the game summary
11    Trial = integer(),
12    Balance = integer(),
13    Bet = integer(),
14    Chosen_Color = integer(),
15    Rolled_Colors = character(),
16    Won = integer())
17
18  repeat{ # loops until money is 0
19    round <- round+1 # increment round counter
20    if (flag==1) betting_percentage <- runif(1,min = 0, max=1.00001) # randomize
        betting percentage if player decides to randomize
21
22    # calculate betting money, subtracts it from the balance
23    bet <- ceiling(balance*betting_percentage)
24    balance <- balance-bet
25
26    won <- 0 # variable to store the number of matched colors
27    betColor <- sample(colors, size = 1, prob = chance) # randomly choose a color
        among the 6 colors
28    rolledColor <- sample(colors, size = 3, prob = chance, replace=TRUE) # 1 color per
        die (3 dice in total) is randomly selected with replacement
29    for(i in rolledColor)  # counts the number of winnings
30      if (betColor == i) won <- won+1
31    gameSummary[nrow(gameSummary)+1,] <- c(round,balance,bet,betColor,
        toString(rolledColor),won) # update gameSummary with new data from this round
32    if (won>0) balance <- balance+(bet*(won+1)) # calculating the total money after
        the round
33    print(c(round,balance,bet,betColor, toString(rolledColor),won)) # for debugging
        purposes
34    if (balance==0) break  # game ends, player loses since there is no money left
35  }
36  write.table(gameSummary, "gameSummary.csv", append=TRUE, sep=",", row.names = FALSE)
      # write a csv file that records the information per round of the color game
37  return (gameSummary) # function returns a game summary where the necessary
      information about this game is tabulated
38  }
```

C Simulation and Analysis

Running the colorGame function with an arbitrary initial money of 10,000, a low betting percentage of 0.05, the color game ends after 1404 rounds, with a balance of 0 (Figure 2.1). If we run the game with the same initial money, but with a fairly high betting percentage of 0.90, the game ends after 7 rounds, and a balance of 0 (Figure 2.2). Additionally, if we run with the same betting percentages, but with a low initial money of 150, we get 626 rounds for the 0.05 (Figure 2.3), and only 4 rounds for the 0.90(Figure 2.4), both having 0 balance at the end. Due to the sheer number of rounds for these trials, a graph is used to visualize the balance per round.

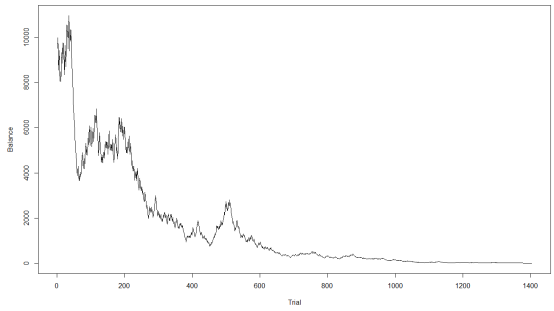


Figure 2.1 Graph of the simulation with initial money: 10,000, betting percentage: 0.05

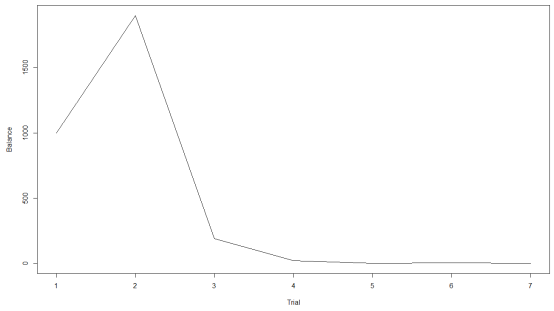


Figure 2.2 Graph of the simulation with initial money: 10,000, betting percentage: 0.90

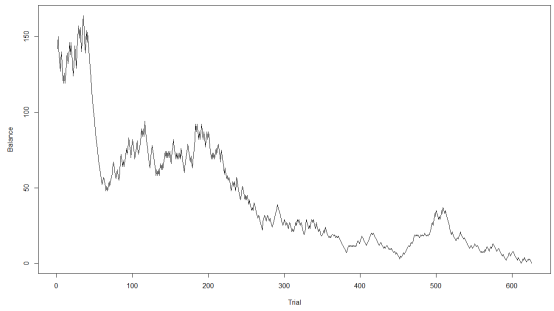


Figure 2.3 Graph of the simulation with initial money: 150, betting percentage: 0.05

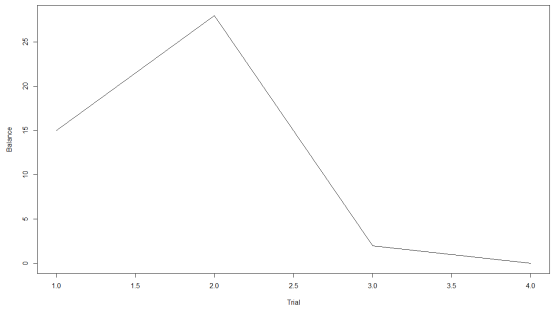


Figure 2.4 Graph of the simulation with initial money: 150, betting percentage: 0.90

Although these 4 iterations of the color game showed the eventual loss of money, we still don't have enough confidence to conclude that the player will always lose their money. Thus, a second function is needed to replicate the color game as many as possible, with randomized initial money and betting percentages. The number of rounds it took for every iteration is also noted, to signify that the simulation really halted once the balance is 0.

```
#PARAMETERS: number of trials and the percentage of their money they are willing to
    bet. betting percentage can be -1, randomize betting percentage for every round
testColorGame <- function(trials, betting_percentage){

  set.seed(299792458) # seed allows for replication of the same output

  # a blank data frame to tabulate the results of the color game from randomized
      initial money
  testSummary = data.frame(
    Case = integer(),
    Money = integer(),
    Rounds = integer())

  for (i in 1:trials){ # loops until number of trials is reached

    # randomize the initial money using uniform distribution from 1 to 1,000,000
    money <- ceiling(runif(1,min=1,max=1000001))

    # get the number of rounds of this trial of color game from the game summary
    rounds <- nrow(colorGame(money, betting_percentage))

```

```
    #updates the table for this test
    testSummary[nrow(testSummary)+1,] <- c(i,money,rounds)
  }

  # make a csv file to record results of the test
  write.table(testSummary, "testSummary.csv", append=FALSE, sep = ",", row.names =
      FALSE)

  # function returns a tabulated summary of this test, noting the initial money and
      the number of rounds it took to end the game
  return(testSummary)
}
```

Using the code above, the color game is repeated 100 times with randomized initial money (ranging from 1 to 1,000,000) and randomized betting percentages (from 0 to 1), using the code above. The results are presented in Table 2.1. Additionally, just to illustrate that having low betting percentages also lead to losing money, we simulate the game with a randomized betting percentage ranging from 0 to 0.25, see Table 2.2. With the 2 simulations, there is no doubt that the player will lose all their money after enough rounds of the color game, even though the second simulation took much longer than the first.

Table 2.1: Results of the simulated color game with 100 repetitions and randomized initial money and betting percentages

Case	Money	Rounds	Case	Money	Rounds	Case	Money	Rounds	Case	Money	Rounds
1	923928	23	26	385607	40	51	836726	34	76	105831	13
2	241872	43	27	777254	33	52	642445	72	77	95155	24
3	60606	11	28	607824	19	53	391855	55	78	306581	10
4	406655	43	29	251510	32	54	303307	24	79	371144	32
5	801737	15	30	554316	35	55	747028	34	80	196765	46
6	71303	21	31	17603	7	56	315055	31	81	563927	38
7	620532	35	32	970819	28	57	884012	31	82	916158	29
8	83513	19	33	371281	37	58	949059	27	83	703074	27
9	880118	28	34	251760	23	59	170030	8	84	69897	41
10	121789	22	35	193057	33	60	594862	71	85	323010	15
11	790102	27	36	486671	29	61	638897	36	86	57290	51
12	298273	77	37	420510	28	62	418969	20	87	866494	34
13	353870	36	38	153659	50	63	680496	33	88	70009	22
14	451987	16	39	248890	24	64	473629	22	89	513953	24
15	981965	43	40	94571	14	65	994609	33	90	382194	17
16	2535	9	41	197215	42	66	23830	9	91	976131	27
17	192284	13	42	20319	7	67	480501	29	92	489654	20
18	250368	19	43	703992	44	68	519852	58	93	884906	21
19	238110	40	44	457980	28	69	238781	16	94	120936	33
20	26733	34	45	496095	60	70	181612	45	95	67437	20
21	721801	23	46	15528	27	71	198497	11	96	262839	48
22	661240	40	47	644417	51	72	848812	23	97	483453	42
23	558805	25	48	352763	21	73	504576	37	98	782864	18
24	551379	25	49	58637	39	74	800525	33	99	87455	27
25	211887	8	50	173330	46	75	828281	19	100	329405	16

Table 2.2: Results of the simulated color game with 100 repetitions and randomized initial money and betting percentages ranging from 0 to 0.25

Case	Money	Rounds	Case	Money	Rounds	Case	Money	Rounds	Case	Money	Rounds
1	923928	514	26	522752	451	51	395112	451	76	974851	674
2	125089	334	27	528558	378	52	609889	445	77	478727	634
3	343760	418	28	721635	583	53	832978	852	78	123049	413
4	486798	553	29	158369	592	54	914738	258	79	509927	265
5	594862	510	30	352376	449	55	465677	388	80	776255	403
6	383033	758	31	558676	381	56	355137	505	81	481497	273
7	797354	750	32	369595	578	57	618638	373	82	233224	519
8	796045	543	33	783467	357	58	617078	447	83	859718	1041
9	124153	415	34	765129	577	59	177221	751	84	853086	431
10	517786	354	35	399492	889	60	12021	193	85	913027	670
11	765132	697	36	698573	642	61	797785	345	86	102917	600
12	678168	565	37	237723	983	62	722184	426	87	474702	433
13	402141	335	38	598792	818	63	795659	829	88	298441	307
14	805929	267	39	890889	900	64	553914	426	89	511404	508
15	456134	347	40	673812	420	65	130232	765	90	82636	488
16	381781	753	41	84257	452	66	858432	508	91	334914	486
17	575852	327	42	830163	427	67	708172	503	92	831917	533
18	225475	514	43	524062	904	68	24735	338	93	158683	290
19	598298	482	44	922062	541	69	191946	371	94	159404	520
20	191451	511	45	666351	367	70	665572	670	95	73561	376
21	660297	539	46	46121	404	71	726155	515	96	658434	415
22	495736	566	47	672704	560	72	148708	384	97	375971	724
23	624469	421	48	625999	328	73	209890	275	98	348515	304
24	293254	384	49	431603	450	74	139513	332	99	765029	446
25	998975	532	50	572271	656	75	37181	477	100	536263	435

PROBLEM 4

(THE POWER SET) A collection of all possible subsets $\{E_i\} \in \omega$ is called the *power set* of Ω, denoted by $\rho(\Omega)$. For example, if $\Omega = \{1, 2, 3\}$, then its power set is given by $\rho(\Omega) = \{\emptyset, \Omega, \{1\}, \{2\}, \{3\}\}, \{1, 2, \}, \{1, 3\}, \{2, 3\}$. Note that all power sets are valid event spaces.

(a) Create a function that generates the power set of any given vector.

(b) Create another function that validates whether the generated power set is an event space

A R Code and Analysis

(A) Create a function that generates the power set of any given vector

```
#A)Create a function that generates the power set of any given vector.
powerSet <-function (x, m, rev = FALSE)
{
  if (missing(m)) m = length(x)
  if (m == 0) return(list(x[c()]))

  out = list(x[c()])
  if (length(x) == 1)
    return(c(out, list(x)))
  for (i in seq_along(x)) {
    if (rev)
      out = c(lapply(out[lengths(out) < m], function(y) c(y, x[i])), out)
    else out = c(out, lapply(out[lengths(out) < m], function(y) c(y, x[i])))
  }
  out
}
```

RECALL,
Power Set Power set $\{P\}(S)$ of a set S is the set of all subsets of S. For example $S = \{a, b, c\}$ then $\mathcal{P}(s) = \{\{\emptyset\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. If S has n elements in it then $\mathcal{P}(s)$ will have 2^n elements.

The powerSet function produces the power set of a vector.
Creates a list containing every subset of the elements of the vector $\backslash code\{x\}$.
- parameter x, vector of elements (the set).
- parameter m, maximum cardinality of subsets.
- parameter rev, logical indicating whether to reverse the order of subsets.

It first checks the parameters defined or inputted. It is composed of if, if-else statements and a recursive function that passes each argument to a certain output for a condition that it satisfies. In the first if statement, if there is no maximum cardinality of subsets that is inputted, then it returns an empty/null set. In the next if statement, it checks the vector of elements. Last is the recursive function that outputs the order of the subsets of the power set.

```
> x <- c("red", "blue", "black")
> powerSet(x)
[[1]]
character(0)

[[2]]
[1] "red"

[[3]]
[1] "blue"

[[4]]
[1] "red"  "blue"

[[5]]
[1] "black"

[[6]]
[1] "red"   "black"

[[7]]
[1] "blue"  "black"

[[8]]
[1] "red"   "blue"  "black"
```

OUTPUT: power set of set $(x) = \{red, blue, black\}$

(B) Create another function that validates whether the generated power set is an event space.

```
B)Create another function that validates whether the generated power set is an event
    space.
validate <- function(powerSet){

  x <- c("1", "2", "3")

  #defining the possible outputs
  valid <- ("VALID. The power set is an event space.")
  invalid <- ("INVALID. The power set is not an event space.")

  n <- length(x)
  cardinality <- length(powerSet)

  #if/else function to identify whether the power set is an event space
  if(2^n != cardinality){
    return(valid)
  }else {
    return(invalid)
  }
}
```

RECALL,
Power Set Power set $\{P\}(S)$ of a set S is the set of all subsets of S. For example $S = \{a, b, c\}$ then $\mathcal{P}(s) = \{\{\emptyset\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. If S has n elements in it then $\mathcal{P}(s)$ will have 2^n elements.

Cardinality is the number of elements of the set.

The validate function validates if the generated power set is a valid event space or not. For the power set to be a valid event space, it must have the right cardinality of elements. This

function contains an if-else statement that checks the cardinality of elements within the power set and returns VALID and INVALID as its outcomes. If the defined set has 2^n elements, then it is a valid event space. Otherwise, it is invalid.

```
> validate(x)
[1] "VALID. The power set is an event space."
> 
```

OUTPUT: The defined set $(x) = \{red, blue, black\}$ *is a VALID event space since it has* 2^n *cardinality of elements.*

PROBLEM 5

(A KNIGHT'S TALE) Using brute force, find the minimum number of moves a horse can make to cover all the tiles in a chess board. The starting point of the horse should only be one of the 4 corner tiles

A Algorithn and R Code

The board is created as a 8x8 matrix with -1 as its initial value.

```
board = matrix(-1, nrow = 8, ncol = 8)
```

Then there are two functions to update and get information about the board, get_board and set_board.

```
set_board = function(x,y,move){board[x,y] <<- move}
get_board = function(x,y){board[x,y]}
```

Get_board returns a value of the board in the position, while set_board on the other hand sets a value in that position.

```
x_move = c(2,1,-1,-2,-2,-1,1,2)
y_move = c(1,2,2,1,-1,-2,-2,-1)
```

The x_move and the y_move are two vectors or arrays that dictates the movement behavior of the knight.

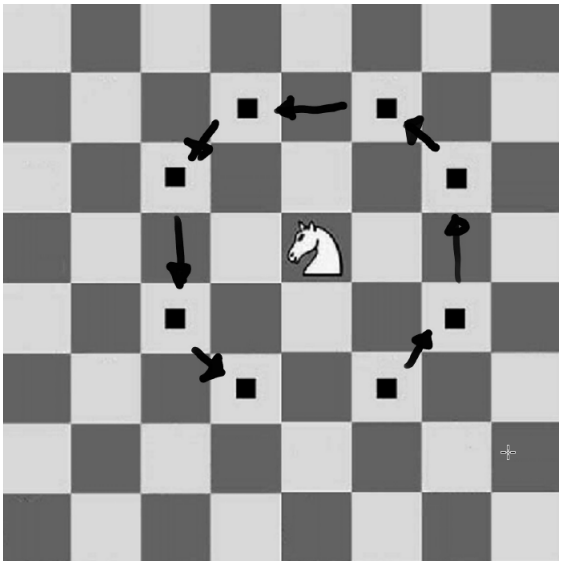
```
valid_move = function(x,y){
  if(x >= 1 & x <= 8 & y >= 1 & y <= 8){
    return (get_board(x,y) == -1)
  }
  return (FALSE)
}
```

The valid_move function checks if the position of the knight moves is valid or not and returns a boolean value in return.

```
knight_move = function(move, x, y){
  if(move == 64){
    return (TRUE)
  }

  for(i in 1:8){
    next_x = x + x_move[i]
    next_y = y + y_move[i]

    if(valid_move(next_x,next_y)){
      print(board)
      set_board(next_x,next_y,move)
      if(knight_move(move + 1, next_x, next_y)){
        return (TRUE)
      }
```

```
      else{
        set_board(next_x,next_y,-1)
      }
    }
  }
  return (FALSE)
}
```

The knight move is the function that records how many times the knight moves and as to what position the knight will move. It first checks if the total moves is 64 which is the maximum amount the knight can move which means covering all squares or positions in the board. Then the program goes to a for loop where it moves the knight downward then right in a counterclockwise movement. The program runs recursively which means it calls itself and checks all possible locations and if a route is to be found not the viable option then the program backtracks until it found another possible route to take on to. If the route ends and it did not cover the whole board then it returns FALSE and changes the value of that position to -1 then backtracks. If it is the solution then it returns TRUE.



The solve function is the one that calls the knight_move function and checks if the problem can be solved or not, and if not, then it prints the board and "No route is true".

```
solve = function(){
  set_board(1,1,0)
  if(knight_move(1,1,1)){
    print(board)
  }
  else{
    print(board)
    print("NO ROUTE IS TRUE")
  }
}
```

B **Simulation**

For the solution, the problem has a route and a solution. The problem can be solved having a total move of 64 moves, starting at 0 up to 63.

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]
[1,]	0	59	38	33	30	17	8	63
[2,]	37	34	31	60	9	62	29	16
[3,]	58	1	36	39	32	27	18	7
[4,]	35	48	41	26	61	10	15	28
[5,]	42	57	2	49	40	36	6	19
[6,]	47	50	45	54	25	20	11	14
[7,]	56	43	52	3	22	13	24	5
[8,]	51	46	55	44	53	4	21	12