

STATS 506 HW1

Minxuan Chen

2023-09-05

Table of contents

0.1	Problem 1 Wine data	1
0.1.1	a	1
0.1.2	b	2
0.1.3	c	3
0.1.4	d	3
0.1.5	e	4
0.2	Problem 2 Perfect Powers	5
0.2.1	a	5
0.2.2	b	6
0.3	Problem 3 ChatGPT	6
0.3.1	a	6
0.3.2	b	8
0.3.3	c	10
0.3.4	d	13
0.3.5	e: Remedy	14

0.1 Problem 1 Wine data

0.1.1 a

```
1 wine.data <- read.table("./wine.data", sep=",", header=FALSE,
2                           col.name=c(
3                             "class", "Alcohol", "Malic acid", "Ash",
4                             "Alcalinity_of_ash", "Magnesium", "Total_phenols",
5                             "Flavanoids", "Nonflavanoid_phenols", "Proanthocyanins",
6                             "Color_intensity", "Hue", "OD280_OD315_of_diluted_wines",
7                             "Proline" ))
```

```
8 head(wine.data)
```

	class	Alcohol	Malic.acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols
1	1	14.23	1.71	2.43	15.6	127	2.80
2	1	13.20	1.78	2.14	11.2	100	2.65
3	1	13.16	2.36	2.67	18.6	101	2.80
4	1	14.37	1.95	2.50	16.8	113	3.85
5	1	13.24	2.59	2.87	21.0	118	2.80
6	1	14.20	1.76	2.45	15.2	112	3.27

	Flavanoids	Nonflavanoid_phenols	Proanthocyanins	Color_intensity	Hue
1	3.06		0.28	2.29	5.64 1.04
2	2.76		0.26	1.28	4.38 1.05
3	3.24		0.30	2.81	5.68 1.03
4	3.49		0.24	2.18	7.80 0.86
5	2.69		0.39	1.82	4.32 1.04
6	3.39		0.34	1.97	6.75 1.05

	OD280_OD315_of_diluted_wines	Proline
1	3.92	1065
2	3.40	1050
3	3.17	1185
4	3.45	1480
5	2.93	735
6	2.85	1450

0.1.2 b

wine.names reports that

```
9. Class Distribution: number of instances per class
class 1 59
class 2 71
class 3 48
```

In our dataframe

```
1 wine.data$class <- as.factor(wine.data$class)
2 summary(wine.data$class)
```

```
1 2 3
59 71 48
```

0.1.3 c

```
1 # 1
2 wine.data[which.max(wine.data$Alcohol) ,1]
```

```
[1] 1
Levels: 1 2 3
```

```
1 # 2
2 wine.data[which.min(wine.data$Alcohol) ,1]
```

```
[1] 2
Levels: 1 2 3
```

```
1 # 3
2 sum((wine.data$Magnesium>114))
```

```
[1] 26
```

```
1 # 4
2 tapply((wine.data$Magnesium>114), wine.data$class, FUN=sum)
```

```
  1  2  3
15  6  5
```

0.1.4 d

```
1 all.tab <- c(class=NA, colMeans(wine.data[, -1]))
2 class.tab <- aggregate(.~class, wine.data, mean)
3 out.tab <- rbind(all.tab, class.tab)
4 rownames(out.tab) <- c("overall", "class 1", "class 2", "class 3")
5 out.tab[, -1]
```

	Alcohol	Malic.acid	Ash	Alcalinity_of_ash	Magnesium	Total_phenols
overall	13.00062	2.336348	2.366517	19.49494	99.74157	2.295112
class 1	13.74475	2.010678	2.455593	17.03729	106.33898	2.840169
class 2	12.27873	1.932676	2.244789	20.23803	94.54930	2.258873
class 3	13.15375	3.333750	2.437083	21.41667	99.31250	1.678750

	Flavanoids	Nonflavanoid_phenols	Proanthocyanins	Color_intensity
overall	2.0292697		0.3618539	1.590899
class 1	2.9823729		0.2900000	1.899322
class 2	2.0808451		0.3636620	1.630282
class 3	0.7814583		0.4475000	1.153542

	Hue	OD280_OD315_of_diluted_wines	Proline
overall	0.9574494		2.611685
class 1	1.0620339		3.157797
class 2	1.0562817		2.785352
class 3	0.6827083		1.683542

0.1.5 e

We'd like to treat this as a multiple comparisons problem. Use Bonferroni correction

```
1 pairwise.t.test(wine.data$Ash, wine.data$class, p.adjust.method="bonferroni")
```

Pairwise comparisons using t tests with pooled SD

data: wine.data\$Ash and wine.data\$class

```

1      2
2 1.9e-05 -
3 1.00000 0.00028
```

P value adjustment method: bonferroni

Based on the output, with an overall α level of 0.05, the level of Ash is not all equal across the three classes. Furthermore, we can observe significant differences between class 1 and 2, as well as between class 2 and 3. However, there is no significant difference between class 1 and 3.

Implement manually:

```

1 var.tab <- aggregate(Ash~class, wine.data, FUN=var)
2 idx <- list(c(1, 2), c(1, 3), c(2, 3))
```

```

3  cls.nb <- c(59, 71, 48)
4  sd.pool <- sqrt(sum((cls.nb-1)*var.tab$Ash)/(sum(cls.nb)-3))
5  p.values <- seq(3)
6  for (i in seq(3)){
7    i1 <- idx[[i]][1]
8    i2 <- idx[[i]][2]
9    p.values[i] <- 2*(1-pt(abs(out.tab[i1+1, 4]-out.tab[i2+1, 4])/
10                      (sqrt(1/cls.nb[i1]+1/cls.nb[i2])*sd.pool),
11                      df=sum(cls.nb)-3))
12  }
13  p.values

```

```
[1] 6.368278e-06 7.114826e-01 9.209777e-05
```

Please note that these p-values are unadjusted, and multiplying them by 3 will yield the Bonferroni-corrected result (R output). Thus, we obtain the same and correct result through manual calculation.

0.2 Problem 2 Perfect Powers

0.2.1 a

```

1  isPerfectPower <- function(number, p){
2    maxroot <- ceiling(number^(1/p))
3    roots <- abs(seq(0,maxroot)^p-number) < sqrt(.Machine$double.eps)
4    id <- which.max(roots)
5    if (sum(roots)==0)
6      return(list(isPerfect=FALSE, root=number^(1/p)))
7    return(list(isPerfect=TRUE, root=id-1))
8  }
9
10 isPerfectPower(27, p=2)

```

```
$isPerfect
```

```
[1] FALSE
```

```
$root
```

```
[1] 5.196152
```

0.2.2 b

```
1 findRootPower <- function(number){
2   maxpower <- ceiling(log2(number))
3   for (i in 2:maxpower){
4     res <- isPerfectPower(number, p=i)
5     if (res$isPerfect)
6       return (paste(number," = ", res$root, "^", i))
7   }
8   return ("Not a perfect power")
9 }
10 for (i in c(27, 13060694016, 7776, 170859375, 58247422, 94143178827)){
11   print(i)
12   print(findRootPower(i))
13 }
```

```
[1] 27
[1] "27 = 3 ^ 3"
[1] 13060694016
[1] "13060694016 = 6 ^ 13"
[1] 7776
[1] "7776 = 6 ^ 5"
[1] 170859375
[1] "170859375 = 15 ^ 7"
[1] 58247422
[1] "Not a perfect power"
[1] 94143178827
[1] "94143178827 = 3 ^ 23"
```

0.3 Problem 3 ChatGPT

0.3.1 a

GPT-3.5 output:

Here are two R functions for your requirements. The first function, `identify_poker_hand`, takes a vector of suits and a vector of ranks representing a 5-card hand and returns the name of the hand. The second function, `deal_poker_round`, simulates dealing a round of cards in a game of poker (5 card stud) for a specified number of players, displays all the hands, and uses the first function to display the name of each hand.

```

1  # Function to identify the poker hand
2  identify_poker_hand <- function(suits, ranks) {
3    # Ensure that suits and ranks are sorted
4    suits <- sort(suits)
5    ranks <- sort(ranks)
6
7    # Function to check if the hand is a flush
8    is_flush <- function(suits) {
9      all(suits == suits[1])
10   }
11
12   # Function to check if the hand is a straight
13   is_straight <- function(ranks) {
14     all(diff(ranks) == 1)
15   }
16
17   # Check for specific poker hands
18   if (is_flush(suits) && is_straight(ranks)) {
19     if (max(ranks) == 14) {
20       return("Royal Flush")
21     } else {
22       return("Straight Flush")
23     }
24   } else if (any(table(ranks) == 4)) {
25     return("Four of a Kind")
26   } else if (all(table(ranks) == c(2, 3))) {
27     return("Full House")
28   } else if (is_flush(suits)) {
29     return("Flush")
30   } else if (is_straight(ranks)) {
31     return("Straight")
32   } else if (any(table(ranks) == 3)) {
33     return("Three of a Kind")
34   } else if (sum(table(ranks) == 2) == 4) {
35     return("Two Pair")
36   } else if (sum(table(ranks) == 2) == 2) {
37     return("One Pair")
38   } else {
39     return("High Card")
40   }
41 }
42

```

```

43 # Function to simulate dealing a round of poker
44 deal_poker_round <- function(num_players) {
45   # Define the suits and ranks
46   suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
47   ranks <- c(2:10, "Jack", "Queen", "King", "Ace")
48
49   # Create a deck of cards
50   deck <- expand.grid(Rank = ranks, Suit = suits)
51
52   # Shuffle the deck
53   deck <- deck[sample(nrow(deck)), ]
54
55   # Deal cards to players
56   hands <- list()
57   for (i in 1:num_players) {
58     start <- (i - 1) * 5 + 1
59     end <- i * 5
60     player_hand <- deck[start:end, ]
61     hands[[paste("Player", i)]] <- player_hand
62     cat("Player", i, "Hand:", "\n")
63     print(player_hand)
64     cat("Hand Name:", identify_poker_hand(player_hand$Suit, player_hand$Rank), "\n")
65     cat("\n")
66   }
67
68   return(hands)
69 }

```

You can adjust the `num_players` variable to specify the number of players in the game. This code defines the poker hands and simulates dealing cards for each player, displaying their hands and identifying the name of each hand using the `identify_poker_hand` function.

0.3.2 b

We need to check the validity of two functions `identity_poker_hand()` and `deal_poker_round()`. We directly run the example given by GPT.

```

1 # These three lines are also generated by GPT
2 # Example usage:
3 set.seed(123) # Set a random seed for reproducibility

```



```

4 num_players <- 4
5 hands_dealt <- deal_poker_round(num_players)

```

Player 1 Hand:

	Rank	Suit
31	6	Clubs
15	3	Diamonds
14	2	Diamonds
3	4	Hearts
42	4	Spades

Warning in table(ranks) == c(2, 3): longer object length is not a multiple of shorter object length

Hand Name: Straight

Player 2 Hand:

	Rank	Suit
43	5	Spades
37	Queen	Clubs
50	Queen	Spades
25	King	Diamonds
26	Ace	Diamonds

Warning in table(ranks) == c(2, 3): longer object length is not a multiple of shorter object length

Hand Name: Straight

Player 3 Hand:

	Rank	Suit
27	2	Clubs
5	6	Hearts
48	10	Spades
28	3	Clubs
9	10	Hearts

Warning in table(ranks) == c(2, 3): longer object length is not a multiple of shorter object length

Hand Name: Straight

Player 4 Hand:

	Rank	Suit
29	4	Clubs
35	10	Clubs
8	9	Hearts
47	9	Spades
7	8	Hearts

Warning in table(ranks) == c(2, 3): longer object length is not a multiple of shorter object length

Hand Name: Straight

From the result, the code works, but throws **Warnings** when trying to return the hand name.

0.3.3 c

Commented codes.

Note that certain comments are automatically generated by GPT and are enclosed in parentheses. If no further comments are appended below these parentheses, it means that we agree with GPT.

```
1 # (Function to identify the poker hand)
2 identify_poker_hand <- function(suits, ranks) {
3   # (Ensure that suits and ranks are sorted)
4   suits <- sort(suits)
5   ranks <- sort(ranks)
6
7   # (Function to check if the hand is a flush)
8   is_flush <- function(suits) {
9     # suits == suits[1] returns a logical vector representing whether each
10    # suit is the same as the first one.
11    # all() can test if all of the values given are true.
12    # So the return represent whether all the same suit, i.e. a flush
13    all(suits == suits[1])
14  }
15
16  # (Function to check if the hand is a straight)
```

```

17 is_straight <- function(ranks) {
18   # From help(diff), "If x is a vector of length n and differences = 1,
19   # then the computed result is equal to the successive differences
20   # x[(1+lag):n] - x[1:(n-lag)]." Here lag=1(default).
21   # This code test whether all diffs are 1, i.e. a straight
22   all(diff(ranks) == 1)
23 }
24
25 # (Check for specific poker hands)
26 if (is_flush(suits) && is_straight(ranks)) {
27   # test whether contains ace (14)
28   if (max(ranks) == 14) {
29     return("Royal Flush")
30   } else {
31     return("Straight Flush")
32   }
33   # table() returns a contingency table of the counts at each levels
34   # in rank. any() check whether at least one value is true.
35   # So 4 means four cards of the same suit
36 } else if (any(table(ranks) == 4)) {
37   return("Four of a Kind")
38   # full house will yield a table (2, 3) or (3, 2), depending on the order of ranks
39   # The method may be conceptually correct, but not the implementation
40 } else if (all(table(ranks) == c(2, 3))) {
41   return("Full House")
42   # trivial
43 } else if (is_flush(suits)) {
44   return("Flush")
45   # trivial
46 } else if (is_straight(ranks)) {
47   return("Straight")
48   # similar to "Four of a kind"
49 } else if (any(table(ranks) == 3)) {
50   return("Three of a Kind")
51   # table(ranks) == 2 yield a permutation of (TRUE, TRUE, FALSE)
52   # the method may be correct but the implementation requires adjustment.
53 } else if (sum(table(ranks) == 2) == 4) {
54   return("Two Pair")
55   # table(ranks) == 2 yield a permutation of (TRUE, FALSE, FALSE, FALSE)
56   # the method may be correct but the implementation requires adjustment.
57 } else if (sum(table(ranks) == 2) == 2) {

```

```

58     return("One Pair")
59 } else {
60     # trivial
61     return("High Card")
62 }
63 }
64
65 # (Function to simulate dealing a round of poker
66 deal_poker_round <- function(num_players) {
67     # (Define the suits and ranks)
68     suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
69     ranks <- c(2:10, "Jack", "Queen", "King", "Ace")
70
71     # (Create a deck of cards)
72     # expand.grid() returns the Cartesian product of ranks and suits,
73     # which is the full representation of cards
74     deck <- expand.grid(Rank = ranks, Suit = suits)
75
76     # (Shuffle the deck)
77     deck <- deck[sample(nrow(deck)), ]
78
79     # (Deal cards to players)
80     hands <- list()
81     for (i in 1:num_players) {
82         # index is : 1st (1 2 3 4 5), 2nd (6,7,8,9,10)
83         # so we use the following two formulas
84         start <- (i - 1) * 5 + 1
85         end <- i * 5
86         # draw five cards from the deck
87         player_hand <- deck[start:end, ]
88         hands[[paste("Player", i)]] <- player_hand
89         cat("Player", i, "Hand:", "\n")
90         print(player_hand)
91         # print hand name
92         cat("Hand Name:", identify_poker_hand(player_hand$Suit, player_hand$Rank), "\n")
93         cat("\n")
94     }
95
96     return(hands)
97 }

```

0.3.4 d

- Are the inputs and outputs as described above?

Yes. As shown in the previous output.

- Are the hands valid (e.g. real cards, no duplicates, right number)?

Yes. From

```
1 suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
2 ranks <- c(2:10, "Jack", "Queen", "King", "Ace")
3 deck <- expand.grid(Rank = ranks, Suit = suits)
4
5 # (Shuffle the deck)
6 deck <- deck[sample(nrow(deck)), ]
```

- Are the names of the hands correct?

No.

```
1 #full house
2 ranks <- c(2,2,2,3,3)
3 suits <- c("Hearts", "Diamonds", "Clubs", "Spades", "Clubs")
4 cat("Should be full house but: ", identify_poker_hand(suits, ranks))
```

Should be full house but: Three of a Kind

```
1 #two pair
2 ranks <- c(2,2,3,3,4)
3 suits <- c("Hearts", "Diamonds", "Clubs", "Spades", "Clubs")
4 cat("Should be two pair but: ", identify_poker_hand(suits, ranks))
```

Should be two pair but: One Pair

```
1 #one pair
2 ranks <- c(2,2,3,6,4)
3 suits <- c("Hearts", "Diamonds", "Clubs", "Spades", "Clubs")
4 cat("Should be one pair but: ", identify_poker_hand(suits, ranks))
```

Should be one pair but: High Card

- Does it ensure no duplicates in cards across hands? What happens if you ask for more than 10 hands to be dealt (as there are only 52 cards in a standard deck)?

`expand.grid()` can generate non-duplicate 52 cards, and the default value of `replace` in `sample()` is `FALSE`, ensuring no duplicates in cards across hands.

Asking more than 10 hands will result in NA value in Player 11's hands.

```
1 capture.output(out <- deal_poker_round(11), file=nullfile())
2 out[["Player 11"]]
```

	Rank	Suit
29	4	Clubs
1	2	Hearts
NA	<NA>	<NA>
NA.1	<NA>	<NA>
NA.2	<NA>	<NA>

0.3.5 e: Remedy

We use the code in part c (0.3.3) to analyse the errors.

1. Line 4, 5, 68, 69 and 92. Sorting `rank`s which contains numbers and strings is problematic. Additionally, sorting `rank`s and `suits` separately can disrupt the `rank-suit` pairing.

We can use a named vector `2:14` to define the ranks. The sort of suits is unnecessary since we don't perform hand comparisons.

```
1 suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
2 ranks <- 2:14
3 names(ranks) <- c(2:10, "Jack", "Queen", "King", "Ace")
4
5 ranks.st <- sort(ranks, index.return=TRUE)
6 ranks <- ranks.st$x
7 suits <- suits[ranks.st$ix]
```

2. Line 40, full house. Full house will yield a table (2, 3) or (3, 2), depending on the order of ranks.

```
1 else if (identical(as.numeric((sort(table(ranks)))), c(2, 3)))
```

3. Line 53, two pair. Similar to full house.

```
1 else if (sum(table(ranks) == 2) == 2)
```

4. Line 57, one pair.

```
1 else if (sum(table(ranks) == 2) == 1)
```

5. Line 66. We need to limit the number of players.

```
1 deal_poker_round <- function(num_players) {
2   if(num_players>10)
3     return (paste("Number of players is", num_players, "Too many!"))
```

6. Line 17, 26-31. Note that “A,2,3,4,5” is also a straight.

```
1 is_straight <- function(ranks) {
2   return (all(diff(ranks) == 1) || identical(ranks, c(2,3,4,5,14)))
3 }
4
5 if (is_flush(suits) && is_straight(ranks)) {
6   # test whether contains ace (14)
7   if (min(ranks) == 10) {
8     return("Royal Flush")
9   } else {
10    return("Straight Flush")
11  }
```

Revised :

```
1 # (Function to identify the poker hand)
2 identify_poker_hand <- function(suits, ranks) {
3   # (Ensure that suits and ranks are sorted)
4   ranks.st <- sort(ranks, index.return=TRUE)
5   ranks <- ranks.st$x
6   suits <- suits[ranks.st$ix]
7
8   # (Function to check if the hand is a flush)
9   is_flush <- function(suits) {
10    #suits == suits[1] returns a logical vector representing whether each
11    #suit is the same as the first one.
12    #all() can test if all of the values given are true.
13    #So the return represent whether all the same suit, i.e. a flush
```

```

14     all(suits == suits[1])
15 }
16
17 # (Function to check if the hand is a straight)
18 is_straight <- function(ranks) {
19     #From help(diff), "If x is a vector of length n and differences = 1,
20     #then the computed result is equal to the successive differences
21     #x[(1+lag):n] - x[1:(n-lag)]." Here lag=1(default).
22     #This code test whether all diffs are 1, i.e. a straight
23     return (all(diff(ranks) == 1) || identical(ranks, c(2,3,4,5,14)))
24 }
25
26 # Check for specific poker hands
27 if (is_flush(suits) && is_straight(ranks)) {
28     # test whether contains ace (14)
29     if (min(ranks) == 10) {
30         return("Royal Flush")
31     } else {
32         return("Straight Flush")
33     }
34     # table() returns a contingency table of the counts at each levels
35     # in rank. any() checks whether at least one value is true.
36     # So 4 means four cards of the same suit
37 } else if (any(table(ranks) == 4)) {
38     return("Four of a Kind")
39     # full house will yield a table (2, 3) or (3, 2), depending on the order of ranks
40     # The method may be conceptually correct, but the implementation requires adjustment.
41 } else if (identical(as.numeric(sort(table(ranks))), c(2, 3))) {
42     return("Full House")
43     # trivial
44 } else if (is_flush(suits)) {
45     return("Flush")
46     # trivial
47 } else if (is_straight(ranks)) {
48     return("Straight")
49     # similar to "Four of a kind"
50 } else if (any(table(ranks) == 3)) {
51     return("Three of a Kind")
52     # table(ranks) == 2 yield a permutation of (TRUE, TRUE, FALSE)
53 } else if (sum(table(ranks) == 2) == 2) {
54     return("Two Pair")

```



```

55     # table(ranks) == 2 yield a permutation of (TRUE, FALSE, FALSE, FALSE)
56   } else if (sum(table(ranks) == 2) == 1) {
57     return("One Pair")
58   } else {
59     #trivial
60     return("High Card")
61   }
62 }
63
64 # (Function to simulate dealing a round of poker
65 deal_poker_round <- function(num_players) {
66
67   if(num_players>10)
68     return (paste("Number of players is", num_players, "Too many!"))
69
70   # (Define the suits and ranks)
71   suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
72   ranks <- 2:14
73   names(ranks) <- c(2:10, "Jack", "Queen", "King", "Ace")
74
75   # (Create a deck of cards)
76   # expand.grid() returns the Cartesian product of ranks and suits,
77   # which is the full representation of cards
78   deck <- expand.grid(Rank = ranks, Suit = suits)
79
80   # (Shuffle the deck)
81   deck <- deck[sample(nrow(deck)), ]
82   row.names(deck) <- NULL
83
84   # (Deal cards to players)
85   hands <- list()
86   for (i in 1:num_players) {
87     # index is : 1st (1 2 3 4 5), 2nd (6,7,8,9,10)
88     # so we use the following two formulas
89     start <- (i - 1) * 5 + 1
90     end <- i * 5
91     # draw five cards from the deck
92     player_hand <- deck[start:end, ]
93
94     hands[[paste("Player", i)]] <- player_hand
95     cat("Player", i, "Hand:", "\n")

```

```

96
97     print(data.frame(Rank=names(ranks[player_hand$Rank-1]),
98                     Suit=player_hand$Suit))
99
100     # print hand name
101     cat("Hand Name:",
102         identify_poker_hand(player_hand$Suit, player_hand$Rank), "\n")
103     cat("\n")
104 }
105
106 return(hands)
107 }

```

Test: We employ different seeds to simulate hands dealt to a player.

```

1 #Royal Flush
2 set.seed(436847)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	Ace	Diamonds
2	Queen	Diamonds
3	Jack	Diamonds
4	10	Diamonds
5	King	Diamonds

Hand Name: Royal Flush

```

1 #Straight Flush
2 set.seed(145340)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	5	Clubs
2	3	Clubs
3	7	Clubs
4	4	Clubs
5	6	Clubs

Hand Name: Straight Flush

```

1 #Four of a Kind
2 set.seed(146431)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	Ace	Spades
2	8	Clubs
3	8	Spades
4	8	Hearts
5	8	Diamonds

Hand Name: Four of a Kind

```

1 #Full House
2 set.seed(37784)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	Ace	Clubs
2	Queen	Diamonds
3	Queen	Clubs
4	Ace	Hearts
5	Ace	Diamonds

Hand Name: Full House

```

1 #Flush
2 set.seed(146907)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	Ace	Spades
2	King	Spades
3	5	Spades
4	Queen	Spades
5	Jack	Spades

Hand Name: Flush

```

1 #Straight
2 set.seed(147099)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	10	Spades
2	8	Hearts
3	9	Hearts
4	Jack	Clubs
5	7	Spades

Hand Name: Straight

```

1 #Three of a Kind
2 set.seed(147217)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	3	Spades
2	9	Hearts
3	King	Clubs
4	3	Hearts
5	3	Diamonds

Hand Name: Three of a Kind

```

1 #Two Pair
2 set.seed(147251)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	2	Diamonds
2	10	Spades
3	2	Spades
4	10	Clubs
5	Queen	Hearts

Hand Name: Two Pair

```

1 #One Pair
2 set.seed(147250)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	5	Spades
2	King	Diamonds
3	9	Diamonds
4	9	Spades
5	4	Clubs

Hand Name: One Pair

```

1 #High Card
2 set.seed(147230)
3 tp <- deal_poker_round(1)

```

Player 1 Hand:

	Rank	Suit
1	Queen	Spades
2	Ace	Hearts
3	2	Clubs
4	6	Spades
5	9	Clubs

Hand Name: High Card