Poker

Stefan Björnander

This report describes an application written in Go for comparing two hands in poker.

Table of Contents

Ranks

When comparing poker hands, we need to establish their ranks, which is one of the following:

- Nothing. No cards of the same value, at least two suits, and no consecutive order.
- One pair. Two cards with the same value
- Two pairs. Two pairs of cards with the same value.
- Three of a kind. Three cards with the same value.
- Straight. All card values in consecutive order.
- Flush. All cards of the same suit.
- Full House. Three of a kind and a pair.
- Four of a kind. Four cards with the same value.
- Straight Flush. A straight and a flush.
- Royal Flush. A straight flush with ace as its highest cards.

Technically, we do not need the royal flush since all straight flushes are compared with regards to their highest cards. Thereby follows that a straight flush with an ace outranks all straight flushes without aces.

Comparation

When comparing two hands, we look into their ranks, max values, min values, and rest values.

- Rank
    - One of the ten ranks described above, in ascending order. Royal flush holds the highest rank, and nothing holds the lowest rank.
- Max value
    - Royal flush, straight flush, and straight: the value of the highest cards.
    - Full house and three of a kind: the value of three of a kind.
    - Two pairs: the value of the highest pair.
    - One pair: the value of the pair.
- Min value
    - Two pairs: the value of the lowest pair.
- Rest values
    - Flush and nothing: the values of all the cards, in descending order.
    - Two pairs: the value of the remaining card, that is not a part of any of the pairs.
    - One pair: the values of the remaining three cards, that is not part of the pair, in descending order.

The comparation of two hands is performed as follows:

1. If the hands hold different ranks, the hand with highest rank wins.
2. If they hold the same rank, but different max values, the hand with the largest max value wins.
    - Always applies to four of a kind, full house, and three of a kind, since two hands cannot hold four or three cards with same value. Note that when comparing two full houses, we compare the values of the three of a kind, even though the values of the pairs may be higher.
    - May apply to two pairs and one pair. But the max values may be equal, since two hands may hold pairs of the same values.
    - Never applies to flush or nothing, the max value is always zero.

3. If they hold the same rank and max value, but different min values, the hand with the highest min value wins.
   - Only applies when comparing two hands of two pairs, where the highest pairs are equal. If the lowest pairs are different, the hand with the highest lowest pair wins.
   - Is always zero for all other hands.
4. If they hold the same rank, max value, and min value, the hand with the highest cards among the remaining cards wins.
   - Flush and nothing: the remaining cards are made by all the cards of the hands. The cards are compared in descending order, where the highest card wins.
   - Two pairs: the remaining cards are made up by the fifth card that is not a part of the pairs. The hand with the highest card wins.
   - One pair: the remaining cards are made by the three cards that are not part of the pair. The cards are compared in descending order, where the highest card wins.
   - Always empty for all other hands.

If the hands hold the same rank, max value, min value, and rest values, they are considered to be equal. The following table sums up the features when comparing two hands.

Rank	Max Value	Min Value	Rest Values
Royal flush	The value of the highest card	Always zero	Always empty
Stright flush	The value of the highest card	Always zero	Always empty
Four of a kind	The value of the four cards of a kind	Always zero	Always empty
Full house	The value of the three cards of a kind	Always zero	Always empty
Flush	Always zero	Always zero	The values of all the cards
Straight	The value of the highest card	Always zero	Always empty
Three of a kind	The value of the three cards of a kind	Always zero	Always empty
Two pairs	The value of the highest pair	The value of the lowest pair	The values of the card not in the two pairs
One pair	The value of the pair	Always zero	The values of the three cards not in the pair
Nothing	Always zero	Always zero	The values of all the cards

In case of flush or nothing we need to compare all the cards of the hands, since their four highest cards may hold the same values.

Code

The code is divided into the following source files:

Card	Holds code for a single card.
Poker	Holds code for a hand of five cards, and the comparation of two hands.
Result	Holds the result of the comparation: Win, Tie, or Lose.
Error	Holds error messages to be panicked.
Main	Creates and displays hands.
Main Test	Hold test cases for the comparation of two hands.

Card

The card file handles a card, which is made up by a value and a suit.

poker/card.go
```
package card
```

A card can hold the values between two and ace, inclusive.

```
const (
  Jack int = iota + 11
  Queen
  King
  Ace
)
```

The suit of a card is clubs, diamonds, hearts, and spades.

```
const (
  Clubs int = iota
  Diamonds
  Hearts
  Spades
)
```

A card holds a value and a suit, which are both represented by integer values.

```
type Card struct {
  Value int
  Suit int
}
```

The **New** function returns a new card with the given value and suit.

```
func New(value int, suit int) Card {
  return Card{value, suit}
}
```

The **LessThan** and **GreaterThan** methods are called when cards are inserted in the card list, in ascending or descending order. They return true if the value of left card is less than or greater than the right card, respectively.

```
func (leftCard Card) LessThan(rightCard Card) bool {
  return leftCard.Value < rightCard.Value
}

func (leftCard Card) GreaterThan(rightCard Card) bool {
  return leftCard.Value > rightCard.Value
}
```

Finally, the **String** method returns a string with the value and the suit of the card written in plain text.

```
func (card Card) String() string {
  valueArray := [...]string{"Two", "Three", "Four", "Five", "Six", "Seven",
                      "Eight", "Nine", "Ten", "Jack", "Queen", "King", "Ace"}
  suitArray := [...]string{"Clubs", "Diamonds", "Hearts", "Spades"}
  return valueArray[card.Value - 2] + " of " + suitArray[card.Suit]
}
```

Hand

The source file of the hand holds code for creating a hand, analyzing the card, ranking the hand, comparing two hands, and converting the hand into a string.

main.go
```go
package main

import (
  "strings"
  "strconv"
  "container/list"
  "fmt"
  "poker/card"
  "poker/error"
  "poker/result"
)
```

The rank of a hand may differ from nothing to royal flush.

```go
const (
  Nothing int = iota
  OnePair
  TwoPairs
  ThreeOfAKind
  Straight
  Flush
  FullHouse
  FourOfAKind
  StraightFlush
  RoyalFlush
)
```

The **ComparableHand** interface makes sure that all structures implementing it hold the **CompareWith** method.

```go
type ComparableHand interface {
  CompareWith(compareTo ComparableHand) (theResult int)
}
```

A hand holds a list of cards, a rank, a max and min value, a map of the cards, and a list of rest values.

```go
type hand struct {
  m_cardList *list.List
  m_rank, m_maxValue, m_minValue int
  m_map map[int]*list.List
  m_restValues *list.List
}
```

The **NewHand** function creates and returns a new hand by analyzing the text, analyzing the cards, and setting the rank.

```go
func NewHand(text string) ComparableHand {
  var newHand hand
  (&newHand).analyzeText(text)
  (&newHand).analyzeCards()
  (&newHand).setRankOfHand()
  return newHand
}
```

The **analyzeText** method extracts the value and suit of each card in the text, and creates and adds the cards to the card list of the hand.

```
func (handPtr *hand) analyzeText(text string) {
  handPtr.m_cardList = list.New()
  cardTextArray := strings.Split(text, " ")
```

If there are not five cards in the text, we report and error.

```
  if len(cardTextArray) != 5 {
    error.InvalidNumberOfCards(len(cardTextArray))
  }
```

We iterate thought the cards of the text.

```
  for _, cardText := range cardTextArray {
```

If a card is not represented by two character, we report an error.

```
    if len(cardText) != 2 {
      error.InvalidLengthOfCardText(cardText)
    }
```

We extract the character representing the value, and convert it to upper case.

```
    valueChar := string(cardText[0])
    valueCharUpper := strings.ToUpper(valueChar)

    const valueText string = "23456789TJQKA"
    var value int
```

If the character does not represent a card value, we report an error.

```
    if strings.Contains(valueText, valueCharUpper) {
      value = strings.Index(valueText, valueCharUpper) + 2
    } else {
      error.InvalidCardValue(valueChar)
    }
```

We extract the character representing the suit, and convert it to upper case.

```
    suitChar := string(cardText[1])
    suitCharUpper := strings.ToUpper(suitChar)

    suitMap := map[string]int{"C": card.Clubs, "D": card.Diamonds,
                              "H": card.Hearts, "S": card.Spades}
    suit, exists := suitMap[suitCharUpper]
```

If the character does not represent a card suit, we report an error.

```
    if !exists {
      error.InvalidCardSuit(suitChar)
    }
```

Now we are ready to create a new card, and insert it at its proper position in the card list.

```
    newCard := card.New(value, suit)
    inserted := false

    for iterator := handPtr.m_cardList.Front(); iterator != nil;
        iterator = iterator.Next() {
      card := iterator.Value.(card.Card)
```

If the card is already present in the hand, we report and error.

```
if card == newCard {
   error.CardOccoursTwice(card)
}
```

When we find a card in the card list that is smaller (its value is lower) than the new card, we insert the new card at that position.

```
if newCard.GreaterThan(card) {
   handPtr.m_cardList.InsertBefore(newCard, iterator)
   inserted = true
   break
}
   }
```

If we have iterated through the card list without inserting the new card, we add it at the end of the card list.

```
if !inserted {
   handPtr.m_cardList.PushBack(newCard)
}
   }
}
```

The **analyzeCards** method iterates through the cards and, for each card, count the number of times the card value occurs in the hand. In this way, we catch all four of a kind, three of a kind, and pairs.

```
func (handPtr *hand) analyzeCards() {
  handPtr.m_map = make(map[int]*list.List)
```

For instance, **{2: [12], 3: [10]}** is a full house with two queens and three tens, **{2: [10, 8], 1: [6]}** is two pairs of tens and eights, and one six. The value of the highest pair always comes first in the list, since the card lists was created in that order. Note that we do not look into the suits of the cards in this method. The only time the suits is of interest is when comparing two flushes.

```
for outerIterator := handPtr.m_cardList.Front(); outerIterator != nil;
    outerIterator = outerIterator.Next() {
  outerCard := outerIterator.Value.(card.Card)
  count := 0
```

For each card, we iterate the cards again and count how many times its value occurs in the hand.

```
for innerIterator := handPtr.m_cardList.Front(); innerIterator != nil;
    innerIterator = innerIterator.Next() {
  innerCard := innerIterator.Value.(card.Card)

  if outerCard.Value == innerCard.Value {
    count++
  }
}
```

When we have counted the number of cards with the same value, we look up its list in the map. If there is no list, we create a new list.

```
listPtr, exists := handPtr.m_map[count]
if !exists {
  listPtr = list.New()
}
```

7

We add the value of the cards to the list, if it is not already present in the list, and add the list to the map.

```
    if !containsValueInList(listPtr, outerCard.Value) {
      listPtr.PushBack(outerCard.Value)
    }

    handPtr.m_map[count] = listPtr
  }
}
```

The **containsValueInList** method returns true if the value is a member of the list.

```
func containsValueInList(listPtr *list.List, value interface{}) bool {
  for iterator := listPtr.Front(); iterator != nil;
      iterator = iterator.Next() {
    if iterator.Value == value {
      return true
    }
  }

  return false
}
```

The **setRankOfHand** method sets the rank, max and min value, and the rest values of the hand by calling a sequence of methods, beginning with the royal flush that holds the highest rank.

```
func (handPtr *hand) setRankOfHand() {
```

In case of a royal flush we do not have to do anything, except store the rank.

```
  if handPtr.isRoyalFlush() {
    handPtr.m_rank = RoyalFlush
```

In case of a straight flush we set the max value to the value of the highest card, which is stored at the beginning of the card list (the card list was been sorted that way when it was created).

```
  } else if handPtr.isStraightFlush() {
    handPtr.m_rank = StraightFlush
    handPtr.m_maxValue = handPtr.m_cardList.Front().Value.(card.Card).Value
```

In case of four of a kind, we set the max value to the value of the four cards, which is stored in the map with the key four.

```
  } else if handPtr.isFourOfAKind() {
    handPtr.m_rank = FourOfAKind
    handPtr.m_maxValue = handPtr.m_map[4].Front().Value.(int)
```

In case of full house, we set the max value to the value of the three cards.

```
  } else if handPtr.isFullHouse() {
    handPtr.m_rank = FullHouse
    handPtr.m_maxValue = handPtr.m_map[3].Front().Value.(int)
```

In case of flush, we set the rest values to the values of all the cards, since we may have to compare every card with another flush, in case the four highest cards hold the same values.

```
  } else if handPtr.isFlush() {
    handPtr.m_rank = Flush
    handPtr.loadCardValues(handPtr.m_cardList)
```

In case of straight, we set the max value to the value of the highest cards, which stored at the beginning of the card list.

```
  } else if handPtr.isStraight() {
    handPtr.m_rank = Straight
    handPtr.m_maxValue = handPtr.m_cardList.Front().Value.(card.Card).Value
```

In case of three of a kind, we set the max value to the value of the three cards, which is stored in the map with the key three.

```
  } else if handPtr.isThreeOfAKind() {
    handPtr.m_rank = ThreeOfAKind
    handPtr.m_maxValue = handPtr.m_map[3].Front().Value.(int)
```

In case of two pairs we set the max value to the value of the highest pair, and the min value to the value of the lowest pair. Moreover, and add the fifth remaining card to the rest values. The value of he highest pair always comes first in the list, since the cards was sorted in that order when it was created.

```
  } else if handPtr.isTwoPairs() {
    handPtr.m_rank = TwoPairs
    handPtr.m_maxValue = handPtr.m_map[2].Front().Value.(int)
    handPtr.m_minValue = handPtr.m_map[2].Front().Next().Value.(int)
    handPtr.m_restValues = handPtr.m_map[1]
```

In case of one pair we set the max value to the value of the pair, and add the values of the remaining three cards to the rest values.

```
  } else if handPtr.isOnePair() {
    handPtr.m_rank = OnePair
    handPtr.m_maxValue = handPtr.m_map[2].Front().Value.(int)
    handPtr.m_restValues = handPtr.m_map[1]
```

If case of nothing, we save the values of all the cards to the rest values.

```
  } else {
    handPtr.m_rank = Nothing
    handPtr.loadCardValues(handPtr.m_cardList)
  }
}
```

The **containsKeyInMap** method returns true if the map holds the key.

```
func containsKeyInMap(m map[int]*list.List, key int) bool {
  _, exists := m[key]
  return exists
}
```

The **loadCardValues** method loads the values of the list to the rest values.

```
func (handPtr *hand) loadCardValues(values *list.List) {
  handPtr.m_restValues = list.New()

  for iterator := values.Front(); iterator != nil;
      iterator = iterator.Next() {
    handPtr.m_restValues.PushBack(iterator.Value.(card.Card).Value)
  }
}
```

A hand is a royal flush if it is a straight flush where the highest card is an ace.

```
func (handPtr *hand) isRoyalFlush() bool {
  return handPtr.isStraightFlush() &&
        (handPtr.m_cardList.Front().Value.(card.Card).Value == card.Ace)
}
```

A hand is a straight flush if it is a straight and a flush.

```go
func (handPtr *hand) isStraightFlush() bool {
  return handPtr.isStraight() && handPtr.isFlush()
}
```

A hand is four of a kind if it holds four cards of the same value, which it does if the map holds a key of value four.

```go
func (handPtr *hand) isFourOfAKind() bool {
  return containsKeyInMap(handPtr.m_map, 4)
}
```

A hand is a full house if it is three of a kind and a pair.

```go
func (handPtr *hand) isFullHouse() bool {
  return handPtr.isThreeOfAKind() && handPtr.isOnePair();
}
```

A hand is a flush is all the cards hold the same suit. We iterate through the cards, beginning with the second card, and checking that they all have the same suit as the first card.

```go
func (handPtr *hand) isFlush() bool {
  firstSuit := handPtr.m_cardList.Front().Value.(card.Card).Suit

  for iterator := handPtr.m_cardList.Front().Next(); iterator != nil;
      iterator = iterator.Next() {
    if iterator.Value.(card.Card).Suit != firstSuit {
      return false
    }
  }

  return true
}
```

A hand is a straight if the difference between the first and last card is four (remember that the cards are sorted in descending order) and there is no set of cards larger than one; that is, there are no pairs and no three or four of a kind.

```go
func (handPtr *hand) isStraight() bool {
  firstCard := handPtr.m_cardList.Front().Value.(card.Card)
  lastCard := handPtr.m_cardList.Back().Value.(card.Card)

  return ((firstCard.Value - lastCard.Value) == 4) &&
         (len(handPtr.m_map) == 1) && containsKeyInMap(handPtr.m_map, 1)
}
```

A hand is three of a kind if it holds three cards of the same value, which it does if the map holds a key of value three.

```go
func (handPtr *hand) isThreeOfAKind() bool {
  return containsKeyInMap(handPtr.m_map, 3)
}
```

A hand holds two pairs if there are two occurrences of two cards with the same value.

```go
func (handPtr *hand) isTwoPairs() bool {
  return containsKeyInMap(handPtr.m_map, 2) && (handPtr.m_map[2].Len() == 2)
}
```

A hand holds one pair if there are one occurrence of two cards with the same value.

```
func (handPtr *hand) isOnePair() bool {
  return containsKeyInMap(handPtr.m_map, 2)
}
```

The **CompareWith** method compares the hand with another hand, and return **Win**, **Tie**, or **Lose** of the **Result** type, cast to integer.

```
func (leftHand hand) CompareWith(compareTo  ComparableHand) (theResult int){
```

The **compare** variable holds the difference between the two hands and will eventually be used to determine if the left hand wins, ties, or loses over the right hand.

```
  compare := 0
```

We try to cast the **compareTo** argument of the **ComparableHand** interface to the **hand** structure **rightHand**. If the conversion fails, nothing happens, and **Tie** is returned.

```
  rightHand, ok := compareTo.(hand)
  if ok {
```

If the conversion succeeded, we check that the hands do not overlaps; that is, the same cards does not occur in both the hands. However, I decided to comment the code block, since it is unclear whether the test cases allow overlapping hands.

```
    /*for iterator := leftHand.m_cards.Front(); iterator != nil;
        iterator = iterator.Next() {
      card := iterator.Value.(card.Card)
      if containsValueInList(rightHand.m_cards, card) {
        error.HandsOverlap(card)
      }
    }*/
```

Then we compare the ranks of the hands. If they differ, **compare** is assigned the value of the difference between the left and right hand.

```
    if leftHand.m_rank != rightHand.m_rank {
      compare = leftHand.m_rank - rightHand.m_rank
```

If the hands hold the same rank, we look into their max values. If they differ, **compare** is assigned the value of their difference.

```
    } else if leftHand.m_maxValue != rightHand.m_maxValue {
      compare = leftHand.m_maxValue - rightHand.m_maxValue
```

If the hands hold the same rank and the same max values, we look into their min values. If they differ, **compare** is assigned the value of their difference.

```
    } else if leftHand.m_minValue != rightHand.m_minValue {
      compare = leftHand.m_minValue - rightHand.m_minValue
```

Finally, if the hands hold the same rank as well as max and min values, we look into their rest values, and compare them from the highest to the lowest value. The **compare** variable is assigned the value of their difference, if there in fact is a difference. It is quite possible that there is no difference. For instance, two hands may have flushes with the same values, but different suits.

We iterate through the rest values of both the left and right hand, and compare their values. But before that we need to check whether the rest values are not nil, since we may have two royal flushes, in which case all values so far have been equal, but they do not have rest values.

```
    } else if leftHand.m_restValues != nil {
      leftIterator := leftHand.m_restValues.Front()
      rightIterator := rightHand.m_restValues.Front();

      for leftIterator != nil {
        leftValue := leftIterator.Value.(int)
        rightValue := rightIterator.Value.(int)
```

If we find two cards with different values, **compare** is assigned the difference and we break the loop.

```
        if leftValue != rightValue {
          compare = leftIterator.Value.(int) - rightIterator.Value.(int)
          break
        }

        leftIterator = leftIterator.Next()
        rightIterator = rightIterator.Next()
      }
    }
  }
```

When we have compared the ranks and values of the hands, we return the final result. If **compare** holds a positive value the left hand has won and we return **Win**, if it holds a negative value the right hand has won and we return **Lose**, and if it is zero we have a tie and we return **Tie**. Note that we cast the the result to int, because **CompareWith** returns int rather than **Result**.

```
  if compare > 0 {
    return int(result.Win)

  } else if compare < 0 {
    return int(result.Lose)

  } else {
    return int(result.Tie)
  }
}
```

The **String** method return a string representing the hand, which is divided into three parts:

- The cards, written in plain text.
- The rank and max and min values, unless they are zero.
- The map holding the values of the hand.

```
func (theHand hand) String() string {
```

We iterate through the cards and add their text to the cards buffer.

```
  cardsBuffer := "{"
  for iterator := theHand.m_cardList.Front(); iterator != nil;
      iterator = iterator.Next() {
    if iterator != theHand.m_cardList.Front() {
      cardsBuffer += ", "
    }

    card := iterator.Value.(card.Card)
    cardsBuffer += card.String()
  }
  cardsBuffer += "}"
```

We add the text of the rank to the rank buffer.

```
rankArray := [...] string {"Nothing", "One Pair", "Two Pairs",
                "Three Of A Kind", "Straight", "Flush", "Full House",
                "Four Of A Kind", "Straight Flush", "Royal Flush"};
rankBuffer := rankArray[theHand.m_rank]
```

We add the max and min values to the rank buffer, if they are not zero.

```
if theHand.m_maxValue != 0 {
  rankBuffer += ", max value " + strconv.Itoa(theHand.m_maxValue)
}

if theHand.m_minValue != 0 {
  rankBuffer += ", min value " + strconv.Itoa(theHand.m_minValue)
}
```

We add the map to the map buffer. We iterate through the map and, for each key, write the key and its associated value list.

```
mapBuffer := "{"
for size, list := range theHand.m_map {
  if len(mapBuffer) > 1 {
    mapBuffer += ", "
  }

  mapBuffer += strconv.Itoa(size) + ": ["
```

For each key, we iterate through its value list and add each value to the map buffer.

```
for iterator := list.Front(); iterator != nil;
    iterator = iterator.Next() {
  if iterator != list.Front() {
    mapBuffer += ", "
  }

  mapBuffer += strconv.Itoa(iterator.Value.(int))
}

  mapBuffer += "]"
}
mapBuffer += "}"
```

Finally, we return the card buffer, rank buffer, and map buffer.

```
  return cardsBuffer + "\n" + rankBuffer + ", " + mapBuffer
}
```

Errors

There are six possible errors:

- Invalid number of cards. There are less or more than five cards in the text describing the hand.
- Invalid length of card text. The text describing a card is less or more than two characters.
- Invalid card value. The card character is not a digit between two and nine, inclusive. Neither is it a character describing ten, jack, queen, king, or ace.
- Invalid card suit. The card suit is not a character representing clubs, diamonds, hearts, or spades.
- Card occurs twice. The same card occurs twice in the hand.
- Hands overlaps. The same card occurs in both hands when comparing hands.

Each of these errors result in a panic call, that can be caught by the calling function.

poker /error.go
```go
package error

import (
  "fmt"
  "poker/card"
)

type Error struct {
  m_message string
}

func (errorPtr *Error) Message() string {
  return errorPtr.m_message
}

func InvalidNumberOfCards(number int) {
  panic(Error{fmt.Sprintf("Invalid number of cards in hand: %d.", number)})
}

func InvalidLengthOfCardText(cardText string) {
  panic(Error{fmt.Sprintf("Invalid length of card text: %s.", cardText)})
}

func InvalidCardValue(valueChar string) {
  panic(Error{fmt.Sprintf("Invalid card value: %s.", valueChar)})
}

func InvalidCardSuit(suitChar string) {
  panic(Error{fmt.Sprintf("Invalid card suit: %s.", suitChar)})
}

func CardOccoursTwice(card card.Card) {
  panic(Error{fmt.Sprintf("Card occours twice in the hand: %s.",
                          card.String())})
}

func HandsOverlap(card card.Card) {
  panic(Error{fmt.Sprintf("Card occours in both hands: %s.",
                          card.String())})
}
```

Result

The Result type values represent tie, win, and lose between the hands given to the **CompareWith** method.

poker /result.go
```go
package result

type Result int

const (
  Tie Result = iota
  Win
  Lose
)
```

Main

The main program writes a set of hands to make sure the hands have been correctly read and interpreted.
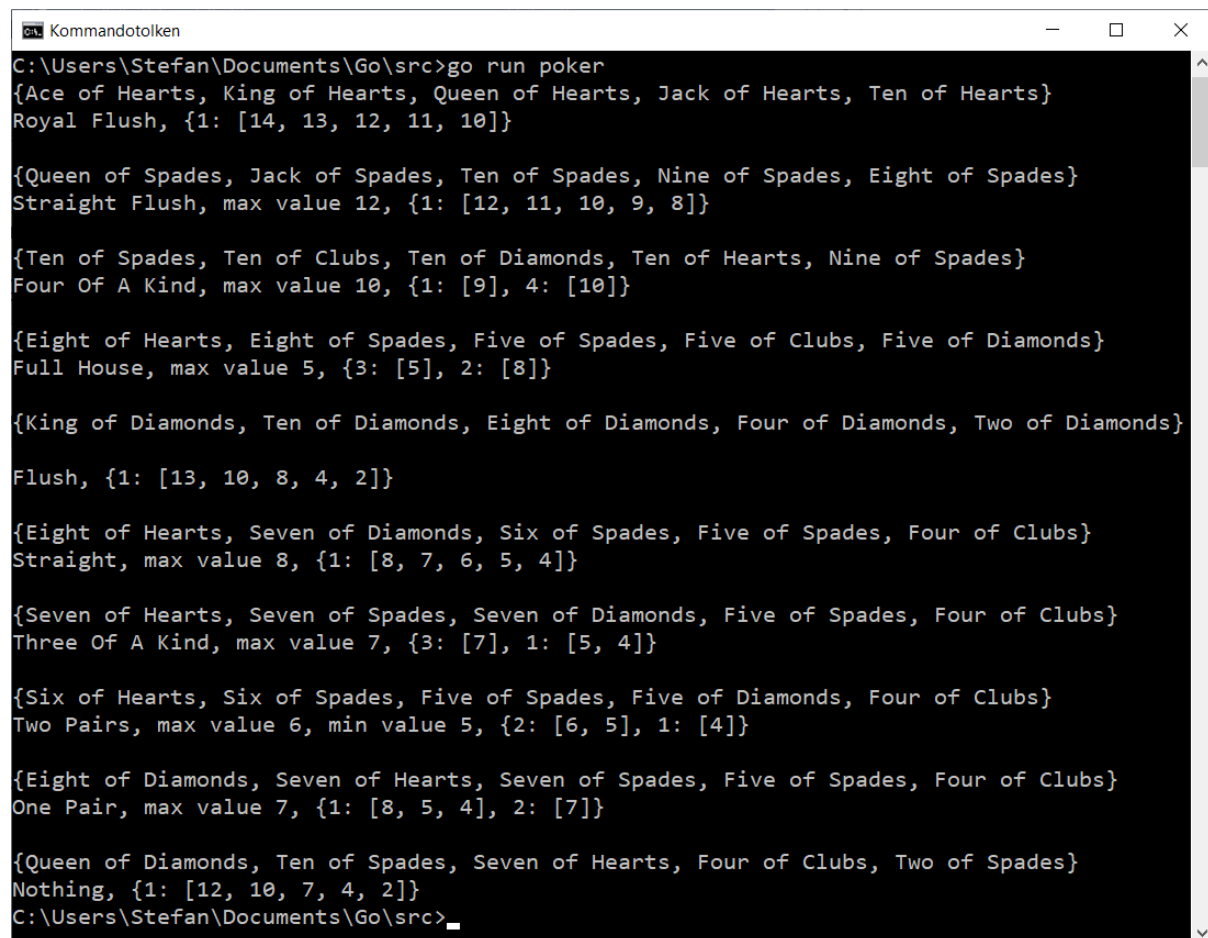
main.go
```go
func main() {
  defer func() {
    err, ok := recover().(error.Error)

    if ok {
      fmt.Println(err.Message())
    }
  }()

  fmt.Printf("%s\n\n", NewHand("Th Jh Qh Kh Ah")) // Royal Flush
  fmt.Printf("%s\n\n", NewHand("Ts Js Qs 9s 8s")) // Straight Flush
  fmt.Printf("%s\n\n", NewHand("Ts Tc Td 9s Th")) // Four of a kind
  fmt.Printf("%s\n\n", NewHand("5s 8h 5c 8s 5d")) // Full House
  fmt.Printf("%s\n\n", NewHand("Kd 8d 4d Td 2d")) // Flush
  fmt.Printf("%s\n\n", NewHand("5s 8h 4c 6s 7d")) // Straight
  fmt.Printf("%s\n\n", NewHand("5s 7h 4c 7s 7d")) // Three of a kind
  fmt.Printf("%s\n\n", NewHand("5s 6h 4c 6s 5d")) // Two Pairs
  fmt.Printf("%s\n\n", NewHand("5s 7h 4c 7s 8d")) // One Pair
  fmt.Printf("%s", NewHand("2s 7h 4c Ts Qd"))     // Nothing
}
```

The execution of the main program produces the following output:

```
Kommandotolken                                              —   □   ✕
C:\Users\Stefan\Documents\Go\src>go run poker
{Ace of Hearts, King of Hearts, Queen of Hearts, Jack of Hearts, Ten of Hearts}
Royal Flush, {1: [14, 13, 12, 11, 10]}

{Queen of Spades, Jack of Spades, Ten of Spades, Nine of Spades, Eight of Spades}
Straight Flush, max value 12, {1: [12, 11, 10, 9, 8]}

{Ten of Spades, Ten of Clubs, Ten of Diamonds, Ten of Hearts, Nine of Spades}
Four Of A Kind, max value 10, {1: [9], 4: [10]}

{Eight of Hearts, Eight of Spades, Five of Spades, Five of Clubs, Five of Diamonds}
Full House, max value 5, {3: [5], 2: [8]}

{King of Diamonds, Ten of Diamonds, Eight of Diamonds, Four of Diamonds, Two of Diamonds}

Flush, {1: [13, 10, 8, 4, 2]}

{Eight of Hearts, Seven of Diamonds, Six of Spades, Five of Spades, Four of Clubs}
Straight, max value 8, {1: [8, 7, 6, 5, 4]}

{Seven of Hearts, Seven of Spades, Seven of Diamonds, Five of Spades, Four of Clubs}
Three Of A Kind, max value 7, {3: [7], 1: [5, 4]}

{Six of Hearts, Six of Spades, Five of Spades, Five of Diamonds, Four of Clubs}
Two Pairs, max value 6, min value 5, {2: [6, 5], 1: [4]}

{Eight of Diamonds, Seven of Hearts, Seven of Spades, Five of Spades, Four of Clubs}
One Pair, max value 7, {1: [8, 5, 4], 2: [7]}

{Queen of Diamonds, Ten of Spades, Seven of Hearts, Four of Clubs, Two of Spades}
Nothing, {1: [12, 10, 7, 4, 2]}
C:\Users\Stefan\Documents\Go\src>
```

Testing

I have tested the code on the test cases given in the test_result.log file.

main_test.go

```go
package main

import (
  "testing"
  "github.com/stretchr/testify/assert"
  "poker/result"
)

func TestStraightThreeOfAKind(t *testing.T) {
  hand1 := NewHand("Kh 2h 5h Jh Ah") // Straigh with ace low
  hand2 := NewHand("Kc 2s Ks 4c Kd") // Three of a kind
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestStraightThreeOfAKindReversed(t *testing.T) {
  hand1 := NewHand("Kc 2s Ks 4c Kd") // Three of a kind
  hand2 := NewHand("Kh 2h 5h Jh Ah") // Straigh with ace low
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}

func TestThreeOfAKind(t *testing.T) {
  hand1 := NewHand("Kc 2s Kh 4c Kd") // Three of a kind, Kings
  hand2 := NewHand("2c Js Ks Jc Jd") // Three of a kind, Jacks
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestThreeOfAKindReverse(t *testing.T) {
  hand1 := NewHand("2c Js Ks Jc Jd") // Three of a kind, Jacks
  hand2 := NewHand("Kc 2s Kh 4c Kd") // Three of a kind, Kings
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}

func TestTwoPairs(t *testing.T) {
  hand1 := NewHand("Kc 2s Kh 4c 4d") // Two pairs, Kings ansd Twos
  hand2 := NewHand("8c Js Ah Jc 8d") // Two pairs, Jacks and Eights
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestTwoPairsReversed(t *testing.T) {
  hand1 := NewHand("8c Js Ah Jc 8d") // Two pairs, Jacks and Eights
  hand2 := NewHand("Kc 2s Kh 4c 4d") // Two pairs, Kings ansd Twos
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}

func TestFullHouse(t *testing.T) {
  hand1 := NewHand("5c Ks 5h Kc Kd") // Full House, 3 Kings, 2 Fives
  hand2 := NewHand("Qc As Qh Qd Ad") // Full House, 3 Queens, 2 Aces
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestFullHouseReversed(t *testing.T) {
  hand1 := NewHand("Qc As Qh Qd Ad") // Full House, 3 Queens, 2 Aces
  hand2 := NewHand("5c Ks 5h Kc Kd") // Full House, 3 Kings, 2 Fives
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}
```
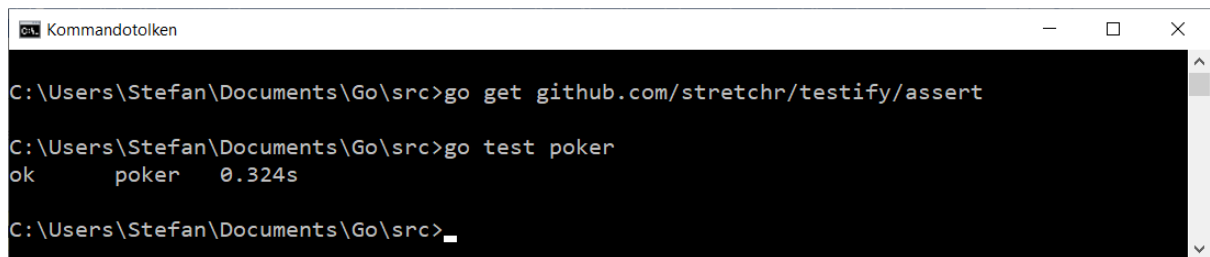
```
func TestFullFourOfAKind(t *testing.T) {
  hand1 := NewHand("Qc Qs Qh Qd 2d") // Four Of A Kind, Queens
  hand2 := NewHand("Ac Ts Th Tc Td") // Four Of A Kind, Tens
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestFullFourOfAKindReversed(t *testing.T) {
  hand1 := NewHand("Ac Ts Th Tc Td") // Four Of A Kind, Tens
  hand2 := NewHand("Qc Qs Qh Qd 2d") // Four Of A Kind, Queens
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}

func TestStraightFlushFullFourOfAKind(t *testing.T) {
  hand1 := NewHand("2d 5d 4d 3d 6d") // Four Of A Kind, Tens
  hand2 := NewHand("Qc Qs Qh Qd 2c") // Four Of A Kind, Queens
  assert.Equal(t, int(result.Win), hand1.CompareWith(hand2))
}

func TestStraightFlushFullFourOfAKindReversed(t *testing.T) {
  hand1 := NewHand("Qc Qs Qh Qd 2c") // Four Of A Kind, Queens
  hand2 := NewHand("2d 5d 4d 3d 6d") // Four Of A Kind, Tens
  assert.Equal(t, int(result.Lose), hand1.CompareWith(hand2))
}
```

```
Kommandotolken                                                    —  □  ×

C:\Users\Stefan\Documents\Go\src>go get github.com/stretchr/testify/assert

C:\Users\Stefan\Documents\Go\src>go test poker
ok      poker   0.324s

C:\Users\Stefan\Documents\Go\src>_
```

Conclusions

This has been a very stimulating task, and I feel that I have learned a lot about programming in Go.

Appendix: C# Code

This appendix holds code solving the same problem in C#.

Values.cs
```
namespace Poker {
  enum Value {Jack = 11, Queen, King, Ace};
}
```

Suit.cs
```
namespace Poker {
  public enum Suit {Clubs, Diamonds, Hearts, Spades}
}
```

Rank.cs
```
namespace Poker {
  enum Rank {Nothing, OnePair, TwoPairs, ThreeOfAKind, Straight,
            Flush, FullHouse, FourOfAKind, StraightFlush, RoyalFlush};
}
```

Result.cs
```
namespace Poker {
  public enum Result {Tie, Win, Lose}
}
```

Card.cs
```
using System;
using System.Collections.Generic;

namespace Poker {
  public class Card {
    private int m_value;
    private Suit m_suit;

    public Card(int value, Suit suit) {
      m_value = value;
      m_suit = suit;
    }

    public int Value {
      get {return m_value;}
    }

    public Suit Suit {
      get {return m_suit;}
    }

    private static string[] m_valueArray =
      {"Two", "Three", "Four", "Five", "Six", "Seven",
       "Eight", "Nine", "Ten", "Jack", "Queen", "King", "Ace"};

    public override string ToString() {
      return m_valueArray[m_value - 2] + " of " +
             Enum.GetName(typeof(Suit), m_suit);
    }
```

```csharp
public override bool Equals(object obj) {
    if (obj is Card) {
      Card card = (Card) obj;
      return (m_value == card.m_value) &&
             (m_suit == card.m_suit);
    }

    return false;
  }

  public override int GetHashCode() {
    return base.GetHashCode();
  }
}

public class HighestFirst : IComparer<Card> {
  public int Compare(Card leftCard, Card rightCard) {
    return rightCard.Value - leftCard.Value;
  }
}

public class LowestFirst : IComparer<Card> {
  public int Compare(Card leftCard, Card rightCard) {
    return leftCard.Value - rightCard.Value;
  }
}
}
```

ComparableHand.cs

```csharp
namespace Poker {
  public interface ComparableHand {
    Result CompareWith(ComparableHand compareTo);
  }
}
```

Hand.cs

```csharp
using System.Text;
using System.Linq;
using System.Collections.Generic;

namespace Poker {
  public class Hand : ComparableHand {
    private List<string> m_rankTextList = new List<string>()
            {"Nothing", "One Pair", "Two Pairs", "Three Of A Kind",
             "Straight", "Flush", "Full House", "Four Of A Kind",
             "Straight Flush", "Royal Flush"};

    private List<Card> m_cards;
    private Rank m_rank;
    private int m_maxValue = 0, m_minValue = 0;
    private List<int> m_restValues;
    private IDictionary<int, List<int>> m_map;

    public Hand(string text) {
      m_cards = TextToCards(text);
      m_cards.Sort(new HighestFirst());
      ExtractSameOfAKind();
      SetRankOfHand();
    }
```

```csharp
private static List<Card> TextToCards(string handText) {
  string[] cardTextArray = handText.Split(' ');

  if (cardTextArray.Length != 5) {
    throw (new InvalidNumberOfCards(cardTextArray.Length));
  }

  List<Card> cardList = new List<Card>();
  foreach (string cardText in cardTextArray) {
    if (cardText.Length != 2) {
      throw (new InvalidLengthOfCardText(cardText));
    }

    char valueChar = cardText[0];
    char valueCharUpper = char.ToUpper(valueChar);
    const string valueText = "23456789TJQKA";
    int value;

    if (valueText.Contains(valueCharUpper)) {
      value = valueText.IndexOf(valueCharUpper) + 2;
    }
    else {
      throw (new InvalidCardValue(valueChar));
    }

    char suitChar = cardText[1];
    char suitCharUpper = char.ToUpper(suitChar);
    IDictionary<char, Suit> suitMap = new Dictionary<char, Suit>()
      {{'S', Suit.Spades}, {'H', Suit.Hearts},
       {'D', Suit.Diamonds}, {'C', Suit.Clubs}};
    Suit suit;

    if (suitMap.ContainsKey(suitCharUpper)) {
      suit = suitMap[suitCharUpper];
    }
    else {
      throw (new InvalidCardSuit(cardText[1]));
    }

    Card card = new Card(value, suit);
    if (cardList.Contains(card)) {
      throw (new CardOccoursTwice(card));
    }

    cardList.Add(card);
  }

  return cardList;
}
```

```
private void SetRankOfHand() {
  if (IsRoyalFlush()) {
    m_rank = Rank.RoyalFlush;
  }
  else if (IsStraightFlush()) {
    m_rank = Rank.StraightFlush;
    m_maxValue = m_cards.First().Value;
  }
  else if (IsFourOfAKind()) {
    m_rank = Rank.FourOfAKind;
    m_maxValue = m_map[4].First();
  }
  else if (IsFullHouse()) {
    m_rank = Rank.FullHouse;
    m_maxValue = m_map[3].First();
  }
  else if (IsFlush()) {
    m_rank = Rank.Flush;
    m_restValues = new List<int>();

    foreach (Card card in m_cards) {
      m_restValues.Add(card.Value);
    }
  }
  else if (IsStraight()) {
    m_rank = Rank.Straight;
    m_maxValue = m_cards.First().Value;
  }
  else if (IsThreeOfAKind()) {
    m_rank = Rank.ThreeOfAKind;
    m_maxValue = m_map[3].First();
  }
  else if (IsTwoPairs()) {
    m_rank = Rank.TwoPairs;
    m_maxValue = m_map[2].First();
    m_minValue = m_map[2].Last();
    m_restValues = m_map[1];
  }
  else if (IsOnePair()) {
    m_rank = Rank.OnePair;
    m_maxValue = m_map[2].First();
    m_restValues = m_map[1];
  }
  else {
    m_rank = Rank.Nothing;
    m_restValues = new List<int>();

    foreach (Card card in m_cards) {
      m_restValues.Add(card.Value);
    }
  }
}

private bool IsRoyalFlush() {
  return IsStraightFlush() &&
         (m_cards.First().Value == ((int) Value.Ace));
}

private bool IsStraightFlush() {
  return IsStraight() && IsFlush();
}
```

```
private bool IsFourOfAKind() {
  return m_map.Keys.Contains(4);
}

private bool IsFullHouse() {
  return IsThreeOfAKind() && IsOnePair();
}
private bool IsFlush() {
  Suit firstSuit = m_cards[0].Suit;
  return (m_cards.FindAll
          (card => card.Suit == firstSuit).Count == m_cards.Count);
}

private bool IsStraight() {
  return (m_map.Keys.Count == 1) && m_map.Keys.Contains(1) &&
         (m_cards.First().Value == (m_cards.Last().Value + 4));
}

private bool IsThreeOfAKind() {
  return m_map.Keys.Contains(3);
}

private bool IsTwoPairs() {
  return m_map.Keys.Contains(2) && (m_map[2].Count == 2);
}

private bool IsOnePair() {
  return m_map.Keys.Contains(2);
}

private void ExtractSameOfAKind() {
  m_map = new Dictionary<int,List<int>>();

  foreach (Card outerCard in m_cards) {
    int count = 0;

    foreach (Card innerCard in m_cards) {
      if (outerCard.Value == innerCard.Value) {
        ++count;
      }
    }

    if (!m_map.ContainsKey(count)) {
      m_map[count] = new List<int>();
    }

    if (!m_map[count].Contains(outerCard.Value)) {
      m_map[count].Add(outerCard.Value);
    }
  }
}

public Result CompareWith(ComparableHand comparableHand) {
  int compare = 0;

  if (comparableHand is Hand) {
    Hand hand = (Hand) comparableHand;
```

```
      foreach (Card card in m_cards) {
        if (hand.m_cards.Contains(card)) {
          throw (new HandsOverlap(card));
        }
      }

      if (m_rank != hand.m_rank) {
        compare = m_rank - hand.m_rank;
      }
      else if (m_maxValue != hand.m_maxValue) {
        compare = m_maxValue - hand.m_maxValue;
      }
      else if (m_minValue != hand.m_minValue) {
        compare = m_minValue - hand.m_minValue;
      }
      else {
        for (int index = 0; index < m_restValues.Count; ++index) {
          if (m_restValues[index] != hand.m_restValues[index]) {
            compare = m_restValues[index] - hand.m_restValues[index];
            break;
          }
        }
      }
    }

    if (compare > 0) {
      return Result.Win;
    }
    else if (compare < 0) {
      return Result.Lose;
    }
    else {
      return Result.Tie;
    }
  }

  public override string ToString() {
    StringBuilder buffer = new StringBuilder("{");
    bool first = true;

    foreach (Card card in m_cards) {
      buffer.Append((first ? "" : ", ") + card.ToString());
      first = false;
    }

    buffer.Append("}\n" + m_rankTextList[(int) m_rank]);

    if (m_maxValue != 0) {
      buffer.Append(", max value " + m_maxValue);
    }

    if (m_minValue != 0) {
      buffer.Append(", min value " + m_minValue);
    }

    buffer.Append(", {");
    bool firstList = true;
    foreach (KeyValuePair<int,List<int>> entry in m_map) {
      buffer.Append((firstList ? "" : ", ") + entry.Key + ": [");

      bool firstValue = true;
```

```
          foreach (int value in entry.Value) {
            buffer.Append((firstValue ? "" : ", ") + value.ToString());
            firstValue = false;
          }

          firstList = false;
          buffer.Append("]");
        }

      return buffer.ToString() + "}\n";
    }
  }
}
```

PokerException.cs

```
using System;

namespace Poker {
  public class PokerException : ApplicationException  {
    public PokerException(string message)
     :base(message) {
    }
  }

  public class InvalidNumberOfCards : PokerException {
    public InvalidNumberOfCards(int number)
     :base("Invalid number of cards: " + number.ToString() + ".") {
    }
  }

  public class InvalidLengthOfCardText : PokerException {
    public InvalidLengthOfCardText(string cardText)
     :base("Invalid length of card text: " + cardText + ".") {
    }
  }

  public class InvalidCardValue : PokerException {
    public InvalidCardValue(char valueChar)
     :base("Invalid card value: " + valueChar + ".") {
    }
  }

  public class InvalidCardSuit : PokerException {
    public InvalidCardSuit(char suitChar)
     :base("Invalid card suit: " + suitChar + ".") {
    }
  }

  public class CardOccoursTwice : PokerException {
    public CardOccoursTwice(Card card)
     :base("Card occours twice in the hand: " + card.ToString() + ".") {
    }
  }

  public class HandsOverlap : PokerException {
    public HandsOverlap(Card card)
     :base("The same card occours in both hands: " +
           card.ToString() + ".") {
    }
  }
}
```

Main.cs

```csharp
using System;

namespace Poker {
  class Start {
    static void Main(string[] args) {
      try {
        Console.Out.WriteLine(new Hand("Th Jh Qh Kh Ah")); // Royal Flush
        Console.Out.WriteLine(new Hand("Ts Js Qs 9s 8s")); // Straight Flush
        Console.Out.WriteLine(new Hand("Ts Tc Td 9s Th")); // Four of a kind
        Console.Out.WriteLine(new Hand("5s 8h 5c 8s 5d")); // Full House
        Console.Out.WriteLine(new Hand("Kd 8d 4d Td 2d")); // Flush
        Console.Out.WriteLine(new Hand("5s 8h 4c 6s 7d")); // Straight
        Console.Out.WriteLine(new Hand("5s 7h 4c 7s 7d")); // Three of akind
        Console.Out.WriteLine(new Hand("5s 6h 4c 6s 5d")); // Two Pairs
        Console.Out.WriteLine(new Hand("5s 7h 4c 7s 8d")); // One Pair
        Console.Out.Write(new Hand("2s 7h 4c Ts Qd"));     // Nothing
      }
      catch (PokerException pokerException) {
        Console.Out.WriteLine(pokerException.Message);
      }
    }
  }
}
```