Poker

Poker! What more classic card game is there?

Programming poker requires creating and connecting many different parts. You must deal cards, manage betting, determine winners, and maintain the game over many rounds. None of these are impossible. But where is the best place to start?

Whenever you begin programming a new project you need to ask yourself this question: what is the most

Detecting Poker Hands

We will need to write code that determines if a set of five cards matches one of these winning hands.

In addition to determining if a hand meets a specific winning condition we also need to determine which is the best winning condition it meets. Given sets of cards from multiple players we need to find out which player has the strongest hand. Finally, we need to be able to break ties.

Let's write code that detects each hand before we get too complicated with ranking hands, breaking ties, and determining winners.

Here's a list of winning poker hands, ranked from most-winning to least-winning.

- Royal flush. A, K, Q, J, 10, all the same suit.
- Straight flush. Five cards in a sequence, all in the same suit.
- Four of a kind. All four cards of the same rank.
- Full house. Three of a kind with a pair.
- Flush. ...
- Straight. ...
- Three of a kind. ...
- Two pair.
- Pair
- High card - whoever has the highest card

We're going to program these in the opposite order of this list. From the easiest hand to detect to the hardest.

High Card

Iterate through the cards and keep track of which card is the highest you've seen. Initialize the variable that keeps track of the highest card to the first card in the list to avoid your-program-is-lying-to-you problems.

```
function highCard(cards) {
  let high = cards[0];
  for (let i = 0; i < cards.length; i++) {
    let card = cards[i];
    if (card.value > high.value) {
      high = card
    }
  }
  return high;
}
```

Two of a Kind

Use a map/dictionary to count how many of each face value appear.

```
function twoOfAKind(cards) {
  let counts = {};
  let bestPair = [];

  for (let i = 0; i < cards.length; i++) {
    let card = cards[i];

    // initialize the face as zero
    if (counts[card.face] === undefined) {
      counts[card.face] = [];
    }

    // now that the face is guaranteed to be instantiated as an array we can add another card
    const currentFace = counts[card.face];
    currentFace.push(card);

    // keep track of which face we've seen doubles of
    // so we don't have to go back and search for it later
    if (currentFace.length >= 2) {
      // if no pair has been found yet save this pair as the best
      if (bestPair.length === 0) {
        bestPair = currentFace;
      } else if (bestPair[0].value < currentFace[0].value) {
        // only update bestPair if another pair is actually of higher value
        // (and actually this means we're a contender for two-pair, not just best-pair)
        bestPair = currentFace;
      }
    }
  }

  return bestPair;
}
```

Three of a Kind

Use a map/dictionary to count how many of each face value appear.

```
function threeOfAKind(cards) {
  let counts = {};
  let bestTriple = [];

  for (let i = 0; i < cards.length; i++) {
    let card = cards[i];

    // initialize the face as zero
    if (counts[card.face] === undefined) {
      counts[card.face] = [];
    }

    // now that the face is guaranteed to be instantiated as an array we can add another card
    const currentFace = counts[card.face];
    currentFace.push(card);

    // keep track of which face we've seen doubles of
    // so we don't have to go back and search for it later
    if (currentFace.length >= 3) {
      // if no pair has been found yet save this pair as the best
      if (bestTriple.length === 0) {
        bestTriple = currentFace;
      } else if (bestTriple[0].value < currentFace[0].value) {
        // only update bestTriple if another triple is actually of higher value
        bestTriple = currentFace;
      }
    }
```

```
    }

    return bestTriple;
  }
```

Four of a Kind

Use a map/dictionary to count how many of each face value appear.

```
function fourOfAKind(cards) {
  let counts = {};
  let bestQuadruple = [];

  for (let i = 0; i < cards.length; i++) {
    let card = cards[i];

    // initialize the face as zero
    if (counts[card.face] === undefined) {
      counts[card.face] = [];
    }

    // now that the face is guaranteed to be instantiated as an array we can add another card
    const currentFace = counts[card.face];
    currentFace.push(card);

    // keep track of which face we've seen doubles of
    // so we don't have to go back and search for it later
    if (currentFace.length >= 4) {
      // if no pair has been found yet save this pair as the best
      if (bestQuadruple.length === 0) {
        bestQuadruple = currentFace;
      } else if (bestQuadruple[0].value < currentFace[0].value) {
        // only update bestQuadruple if another quadruple is actually of higher value
        bestQuadruple = currentFace;
      }
    }
  }

  return bestQuadruple;
}
```

Five of a Kind

This is a joke. There is no five-of-a-kind in Poker.

Flush

OK, we got a bit off track running the pattern of detecting pairs, triples, and four-of-a-kind. There's also flush and straight hands squeezed in the rankings amidst these other hands.

To detect a flush we can do two things: either count the occurences of each suit and see if any have five, or iterate through every card and see if every card has the same suit.

These are slightly different approaches. Counting the occurences of each suit requires iterating through all of the cards. Seeing if every suit is the same requires looking at each card one at a time until we see one suit that doesn't match the consecutive card. Since this is poker we only have to look at five cards, we're not super concerned about efficiency. If you ever approached a similar problem with larger (huger!) data sets it would be better to opt for the fail-early approach. If you you have a large data set and see that your condition doesn't is invalidated early on then stop searching!

```
function flush(cards) {
  const suits = {
    hearts: 0, diamonds: 0, clubs: 0, spades: 0
  }

  let flushSuit = null;

  for (let i = 0; i < cards.length; i++>) {
    const card = cards[i];
    suits[card.suit]++;

    if (suits[card.suit].length === 5) {
      flushSuit = card.suit;
    }
  }

  return flushSuit;
}
```

Here's an interesting thing anticipating breaking ties. It will be useful to know what the highest card of the flush is. The way this is written it just returns the suit of the flush, or `null`.

We could rewrite this so it returns a set of cards that match the flush.

Instead of initializing flushSuit to `null` we can set it to an empty list to indicate that there's an empty set of cards that match the requirements.

```
function flush(cards) {
  const suits = {
    hearts: 0, diamonds: 0, clubs: 0, spades: 0
  }

  let flush = [];

  for (let i = 0; i < cards.length; i++>) {
    const card = cards[i];
    suits[card.suit]++;

    if (suits[card.suit].length === 5) {
      // set the flush to be the entire set of cards
      flush = cards;
    }
  }

  return flush;
}
```

Now since we're returning an array of cards we can use the fail-if-not-satisfied strategy to return either an empty list or return the hand of cards that is necessarily a flush.

```
function flush(cards) {
  for (let i = 1; i < cards.length; i++) {
    const lastCard = cards[i - 1];
    const thisCard = cards[i];
    if (thisCard.suit !== lastCard.suit) {
      // return an empty set of cards.
      return [];
    }
  }

  // if there were any non-matching suits in the hand the function
  // would have returned earlier.
```

```
    return cards;
  }
```

Straight

There's two options to detect if all the cards run in a straight. We can either sort the sorts and iterate through them to see if there's ever a more-than-one-card gap between any two cards. This depends on sorting the cards. We can also use a bucket-tally to mark off what cards are in the set and look at the tally to make sure there's no gaps in a five-slot sequence. Finally, we can keep track of the lowest and the highest card and see if their difference is within a five range. If the difference between the highest and the lowest card exceeds five the set of cards necessarily must not be a straight because at least one card is missing.

```
function straightSort(cards) {
  const sorted = sort(cards);
  for (let i = 0; i < cards.length; i++>) {
    let lastValue = cards[i - 1].value;
    let thisValue = cards[i].value;

    // if any two cards are not in sequential order then there
    // are no cards in this hand that are a straight.
    if (lastValue + 1 !== thisValue) {
      return [];
    }
  }

  // if any card was not in sequential order we would have
  // returned an empty array in the for loop
  return cards;
}
```

Here's how we can use something like the tally to determine if there's a straight. This approach does not require pre-sorting the cards.

```
function straightTally(cards) {
  // create an array with a slot for each value 2-10, J, Q, K, A note: it may be
  // more readable and useful to create an array slightly larger than cards
  // actually exist to easily account for the discrepency in range of cards
  // starting at 2, and ace being the highest card. Indices 0 and 1 are never
  // used.
  // let range = new Array(15)
  let range = new Array(13);

  // initialize the min and max values
  let min = cards[0].value;
  let max = cards[0].value;

  for (let i = 0; i < cards.length; i++>) {
    const card = cards[i];

    min = Math.min(min, card.value);
    max = Math.max(max, card.value);

    // if there's a gap of more than five this hand must not be a straight.
    // for example: 2,3,4,5,6 has 6-2 = 4 and is a straight.
    // for example: 2,3,4,5,7 has 7-2 = 5 and is not a straight.
    // I like giving these concrete examples because subtractions are rifled with off-by-one errors.
    if (max - min >= 5) {
      return [];
    }

    // initialize card values that haven't been seen before
    if (range[card.value] === undefined) {
      range[card.value] = 0;
```

```
      }

      // now that values are guaranteed to be initialized it is safe to increment them
      range[card.value]++;

      // if any card appears twice this hand must not be a straight
      if (range[card.value] === 2) {
        return [];
      }
    }

    // if we made it this far without running into deal-breakers
    // then this hand must be a straight.
    return cards;
  }
```

Full House

At this point after writing code to detect other simpler types of hands we are starting to notice programming patterns that repeat themselves when detecting various poker hands. Doing work to detect one poker hand may duplicate work done elsewhere to detect another poker hand. We'll try to combine all the efforts in to one super-efficient hand-analyzing machine once we've proven that we can detect each of the types of hands individually.

Premature optimization is the root of all evil!

Tallying card faces has been a useful mechanism. We've tallied faces in several hand detection routines. Let's write a function that tallies the card faces and use the tally to determine if the hand is a full house.

```
function tallyFaces(cards) {
  const tally = {}
  for (let i = 0; i < cards.length; i++) {
    const card = cards[i];

    if (tally[card.face] === undefined) {
      tally[card.face] = 0;
    }

    tally[card.face]++;
  }

  return tally;
}
```

Detecting a full house we need three of one face and two of another. It would be useful to have information about the tally made for the hand so we can see how many times each card was tallied.

For instance a proper full house hand will look like this:

```
hand [3, 3, 3, 8, 8]
tally [3: 3, 8: 2]
meta-tally [1: 0, 2: 1, 3: 1, 4: 0]
```

Here's the possible meta-tallies of all possible poker hands:

All cards have different faces: {1: 5} Four of a kind: {4: 1, 1: 1} Full house: {3: 1, 2: 1} Three of a kind: {3: 1, 1: 2} Two pair: {2: 2, 1: 1} Pair: {2: 1, 1: 3} High card: {1: 5} (same as all cards have different faces)

Meta-tallies for suits don't matter. Suits only matter for flushes, tie-breaking high cards, and royal flushes.

```
function metaTally(tally) {
  const metaTally = [0, 0, 0, 0, 0];
  for (const face of tally) {
    const occurences = tally[face];
    metaTally[occurences]++;
  }
  return metaTally;
}
```

Straight Flush

A straight flush is a high-ranking poker hand comprosed of simultaneously having a straight and a flush.

We can combine the two subroutines that detect a straight and a flush to determine if the staright-flush is present.

```
function straightFlush(cards) {
  const straight = isStraight(cards);
  const flush = isFlush(cards);

  if (straight.length === 5 && flush.length === 5) {
    return cards;
  }

  return [];
}
```

Royal Flush

Further the royal flush is the best-of-the-best poken hand. The Royal flush is a sub-catagory of the straight flush. Someone's hand must have 10,J,Q,K,A all of the same suit.

- If a hand is a flush it is worth seeing if it is a straight flush.
- If a hand is a straight flush it is worth seeing if it is a royal flush.

```
function royalFlush(cards) {
  const hand = straightFlush(cards);

  if (hand.length !== 5) {
    return [];
  }

  const loCard = hand[0];
  const hiCard = hand[hand.length - 1];

  // if the straight flush runs from 10 to ace this hand is the real deal!
  if (loCard.face === CARDS.faces.ten && hiCard.face === CARDS.faces.ace) {
    return cards;
  }

  // more than likely this hand does not contain a royal flush
  return [];
}
```

Combining

Fantastic. Now we've seen how to write code that detects each of the poker hands. We have the ability to write a program that combines these hand-analyzers into a program that figures out the highest poker hand each player has.

```
function highestHand(cards) {
  if (isRoyalFlush(cards)) { return 'royal-flush' }
  else if (isStraightFlush(cards)) { return 'straight-flush' }
  else if (isFourOfAKind(cards)) { return 'four-of-a-kind' }
  else if (isFullhouse(cards)) { return 'full-house' }
  else if (isFlush(cards)) { return 'flush' }
  else if (isStraight(cards)) { return 'straight' }
  else if (isThreeOfAKind(cards)) { return 'three-of-a-kind' }
  else if (isTwoPair(cards)) { return 'two-pair' }
  else if (isPair(cards)) { return 'pair' }
  else { return 'high-card' }
}



function winningPlayer(hands) {
  const rankings = {};


}
```

Numerical Rankings

Maybe there's a way to score each hand so we can rank players by the type of hand they have and the highest card they have in the hand for tie-breakers

High card hands could be assigned points from 1-52 for each card in the deck.

The ranking would look like this:

```
52, 51, 50, 49: Ace of Hearts, Diamonds, Clubs, Spades
48, 47, 46, 45: King of Hearts, Diamonds, Clubs, Spades
44, 43, 42, 41: Queen of Hearts, Diamonds, Clubs, Spades
40, 39, 38, 37: Jack of Hearts, Diamonds, Clubs, Spades
36, 35, 34, 43: ten of Hearts, Diamonds, Clubs, Spades
...
12, 11, 10, 9: three of Hearts, Diamonds, Clubs, Spades
08, 07, 06, 05: three of Hearts, Diamonds, Clubs, Spades
04, 03, 02, 01: two of Hearts, Diamonds, Clubs, Spades
```

Given these rankings any ace beats any other face card, any three beats any two and suit orders are preserved so the two of hearts beats the two of diamonds.

We can break ties between players with pairs by looking at their highest card, which falls back to the 1-52 ranking.

We can assign a numerical ranking to each hand instead of returning strings representing `'royal-flush'` or `'two-pair'`.

- 09 Royal flush. A, K, Q, J, 10, all the same suit.
- 08 Straight flush. Five cards in a sequence, all in the same suit.
- 07 Four of a kind. All four cards of the same rank.
- 06 Full house. Three of a kind with a pair.
- 05 Flush. ...
- 04 Straight. ...
- 03 Three of a kind. ...
- 02 Two pair.
- 01 Pair
- 00 High card - whoever has the highest card

If we combine returning the numerical ranking of the type of hand with the numerical ranking of the highest card we can completely account for tie-breaking.

```
function bestHand(hands) {
  let bestHand = hands[0];
  let bestRank = rankHand(bestHand);

  for (let i = 1; i < hands.length; i++) {
    const hand = hands[i];
    const rank = rankHand(hand);

    if (isRankBetterThanBest(rank, bestRank)) {
      bestHand = hand;
      bestRank = rank;
    }
  }
}

function isRankBetterThanBest(rank, bestRank) {
  if (rank.categoryScore > bestRank.categoryScore) {
    return true;
  } else if (rank.categoryScore < bestRank.categoryScore) {
    return false;
  }

  return rank.highCardScore > bestRank.highCardScore;
}
```

We may need to account for special tie-breaking where two hands of the same category have more precedence than a high card.

For example:

[3, 3, 3, 3, A] [4, 4, 4, 4, 2]

Four fours should beat four threes no matter what the individual high card is.

We can factor this in to our program by keeping track of the high card contributing to the category as well as the high card contributing just along for the ride.

Human Version Computation

I had an idea here to illustrate all the computation happening here in terms of having an individual person checking for each type of card hand. Each person would have a whiteboard where they can mark down what information they're keeping track of, just like a function would.

Once I figure out an efficient way to combine all of these checks in to one sequence we can compare the individual human approach to the effectiveness of one person following a sophisticated flow chart.

Putting It All Together

Now we've seen how to write code to detect each of the ten types of hands individually. Let's look at what we've written and see how we can combine all of the different detections in to one algorithms that checks for all of the detections efficiently.

Let's write one for loop that keeps track of several things in one pass:

- Highest card
- Is the hand a flush?
- Is the hand a straight?

- Counts occurences of each face

```
function tally(hand) {
  hand = hand.sort();

  let isFlush = true;
  let isStraight = true;

  let highestCard = hand[0];
  let previousCard = hand[0];

  let nOfAKind = 1;
  let nPairs = 0;

  // keep track of what cards appear two, three or four times.
  // cards appearing two times can happen twice in a five-card hand.
  // cards appearing three or four times can only happen once in a five-card hand.
  let pairs = [];
  let triples = null;
  let quads = null;

  const suits = {[previousCard.suit]: [previousCard]};
  const faces = {[previousCard.face]: [previousCard]};

  for (let i = 1; i < hand.length; i++>) {
    let card = hand[i];

    // keep track of whether the hand is a straight
    if (card.rank !== previousCard.rank + 1) {
      isStraight = false;
    }

    // keep track of whether the hand is a straight
    if (card.suit !== previousCard.suit + 1) {
      isFlush = false;
    }

    if (suits[card.suit] === undefined) {
      suits[card.suit] = [];
    }

    if (faces[card.face] === undefined) {
      faces[card.face] = [];
    }

    suits[card.suit].push(card);
    faces[card.face].push(card);

    // keep track of the highest occurences of face cards
    // like seeing 2 sevens, or 3 jacks, of 4 of a kind
    if (faces[card.face].length > nOfAKind) {
      nOfAKind++;
    }

    // count how many pairs occur
    if (faces[card.face].length === 2) {
      nPairs++;
      pairs.push(card);
    }

    // keep track of which cards are triples
    if (faces[card.face].length === 3) {
      triples = card;
    }

    // keep track of which card appeared four times.
    if (faces[card.face].length === 4) {
```

```
          quads = card;
        }

        let stats = {
          sorted, suits, faces,
          nOfAKind, isFlush, isStraight,
        }

        return stats;
      }
    }



    // * 09 Royal flush. A, K, Q, J, 10, all the same suit.
    // * 08 Straight flush. Five cards in a sequence, all in the same suit.
    // * 07 Four of a kind. All four cards of the same rank.
    // * 06 Full house. Three of a kind with a pair.
    // * 05 Flush. ...
    // * 04 Straight. ...
    // * 03 Three of a kind. ...
    // * 02 Two pair.
    // * 01 Pair
    // * 00 High card - whoever has the highest card
    function rank(stats) {
      if (stats.isFlush && stats.isStraight) {
        if (stats.sorted[0].face === 10) {
          return 'royal-flush';
        } else {
          return 'straight-flush';
        }
      }

      if (stats.nOfAKind === 4) {
        return 'four-of-a-kind';
      }

      if (stats.nOfAKind === 3) {
        // if there's 3-of-a-kind and there's only two groups of face cards
        // present then you must have a full house, like [8, 8, 8, 7, 7]
        if (stats.faces.length === 2) {
          return 'full-house';
        }
      }

      if (stats.isFlush) {
        return 'flush';
      }

      if (stats.isStraight) {
        return 'straight';
      }

      if (stats.nOfAKind === 3) {
        return 'three-of-a-kind';
      }

      if (stats.nPairs === 2) {
        return 'two-pairs';
      }

      if (stats.nOfAKind === 2) {
        return 'pair'
      }

      return 'high-card'
    }
```

```
function score(hands) {
  let best = null;

  for (let hand in hands) {
    let tally = tally(hand);
    let rank = rank(tally);

    if (best === null || rank.score > best.score) {
      best = rank;
    }

    if (rank.score === best.score) {
      best = breakTie(best, rank);
    }
  }

  return best;
}



function tiebreakRoyalFlush(hand1, hand2) {
  // there is no tie breaking a royal flush.
  // players split the spot.
  return [hand1, hand2]:
}



function tiebreakStraightFlush(hand1, hand2) {
  if (hand1.cards[0].value === hand2.cards[0].value) {
    return [hand1, hand2];
  } else if (hand1.cards[0].value > hand2.cards[0].value) {
    return [hand1];
  }
  return [hand2];
}



function tiebreakFourOfAKind(hand1, hand2) {
}



function tiebreakFullHouse(hand1, hand2) {
  // whoever has the highest triple necessarily wins. it is impossible for
  // players to have identical triples because there are only four of each card.
}



function tiebreakFlush(hand1, hand2) {
}



function tiebreakStraight(hand1, hand2) {
}



function tiebreakThreeOfAKind(hand1, hand2) {
}



function tiebreakTwoPair(hand1, hand2) {
}
```

```
function tiebreakOnePair(hand1, hand2) {
}



function tiebreakHighCard(hand1, hand2) {
  // the player with the highest card wins.
  // proceed card by card from the highest card to the lowest card
  // until one player has a higher card. if the players have identical
  // hands of cards they split the pot.

  for (let i = 0; i < hand1.length; i++) {
    let card1 = hand1[i];
    let card2 = hand2[i];

    if (card1.value > card2.value) {
      return [hand1];
    } else if (card2.value > card1.value) {
      return [hand2];
    }
  }

  return [hand1, hand2];
}
```

Tie-Breaker Rules

https://www.adda52.com/poker/poker-rules/cash-game-rules/tie-breaker-rules

ROYAL FLUSH Royal Flush Cards NA An Ace-High Straight Flush Is Called Royal Flush. A Royal Flush Is The Highest Hand In Poker. Between Two Royal Flushes, There Can Be No Tie Breaker. If Two Players Have Royal Flushes, They Split The Pot. The Odds Of This Happening Though Are Very Rare And Almost Impossible In Texas Holdem Because The Board Requires Three Cards Of One Suit For Anyone To Have A Flush In That Suit.

STRAIGHT FLUSH Top Card NA Straight Flushes Come In Varying Strengths From Five High To A King High. A King High Straight Flush Loses Only To A Royal. If More Than One Player Has A Straight Flush, The Winner Is The Player With The Highest Card Used In The Straight. A Queen High Straight Flush Beats A Jack High And A Jack High Beats A Ten High And So On. The Suit Never Comes Into Play I.E. A Seven High Straight Flush Of Diamonds Will Split The Pot With A Seven High Straight Flush Of Hearts.

FOUR OF A KIND Four Of A Kind Card Remaining 1 This One Is Simple. Four Aces Beats Any Other Four Of A Kind, Four Kings Beats Four Queens Or Less And So On. The Only Tricky Part Of A Tie Breaker With Four Of A Kind Is When The Four Falls On The Table In A Game Of Texas Holdem And Is Therefore Shared Between Two (Or More) Players. A Kicker Can Be Used, However, If The Fifth Community Card Is Higher Than Any Card Held By Any Player Still In The Hand, Then The Hand Is Considered A Tie And The Pot Is Split.

FULL HOUSE Trips & Pair Card NA When Two Or More Players Have Full Houses, We Look First At The Strength Of The Three Of A Kind To Determine The Winner. For Example, Aces Full Of Deuces (AAA22) Beats Kings Full Of Jacks (KKKJJ). If There Are Three Of A Kind On The Table (Community Cards) In A Texas Holdem Game That Are Used By Two Or More Players To Make A Full House, Then We Would Look At The Strength Of The Pair To Determine A Winner.

FLUSH Flush Cards NA A Flush Is Any Hand With Five Cards Of The Same Suit. If Two Or More Players Hold A Flush, The Flush With The Highest Card Wins. If More Than One Player Has The Same Strength High Card, Then The Strength Of The Second Highest Card Held Wins. This Continues Through The Five Highest Cards In The Player's Hands.

STRAIGHT Top Card NA A Straight Is Any Five Cards In Sequence, But Not Necessarily Of The Same Suit. If More Than One Player Has A Straight, The Straight Ending In The Card Wins. If Both Straights End In A Card Of The Same Strength, The Hand Is Tied.

THREE OF A KIND Trips Card Remaining 2 If More Than One Player Holds Three Of A Kind, Then The Higher Value Of The Cards Used To Make The Three Of A Kind Determines The Winner. If Two Or More Players Have The Same Three Of A Kind, Then A Fourth Card (And A Fifth If Necessary) Can Be Used As Kickers To Determine The Winner.

TWO PAIR 1st & 2nd Pair Card Remaining 1 The Highest Pair Is Used To Determine The Winner. If Two Or More Players Have The Same Highest Pair, Then The Highest Of The Second Pair Determines The Winner. If Both Players Hold Identical Two Pairs, The Fifth Card Is Used To Break The Tie.

ONE PAIR Pair Card Remaining 3 If Two Or More Players Hold A Single Pair Then Highest Pair Wins. If The Pairs Are Of The Same Value, The Highest Kicker Card Determines The Winner. A Second And Even Third Kicker Can Be Used If Necessary.

HIGH CARD Top Card Remaining 4 When No Player Has Even A Pair, Then The Highest Card Wins. When Both Players Have Identical High Cards, The Next Highest Card Wins, And So On Until Five Cards Have Been Used. In The Unusual Circumstance That Two Players Hold The Identical Five Cards, The Pot Would Be Split.

Pot Splitting

What do Poker and Soccer games have in common? Ties exist.

It is possible there is no one ultimate winner in a given poker hand. For example, two players tie if they each have a royal flush. If two players ever tie the pot is split among all the tied players. Ties are rare, but they are possible and therefore we must program to account for them.

An aside: here's a random soccer story. There was a soccer match in 1994 where it was in both teams self-interest to score own-goals. Read more about it on Wikipedia and check out footage of the game on YouTube

[Barbados 4–2 Grenada (1994 Caribbean Cup qualification)]
(https://en.wikipedia.org/wiki/Barbados_4%E2%80%932_Grenada_(1994_Caribbean_Cup_qualification)

Game footage

```
function payouts(playerHands) {
  let best = playerHands[0];
  for (let hand in playerHands) {
    let scores = score(winners, hand);
    for (let score in score) {

    }
  }
  return winners
}
```

Data Structures: Sets Make Things Easier