

# Problem 2347: Best Poker Hand

## Problem Information

**Difficulty:** Easy

**Acceptance Rate:** 0.00%

**Paid Only:** No

## Problem Description

You are given an integer array

`ranks`

and a character array

`suits`

. You have

5

cards where the

`i`

th

card has a rank of

`ranks[i]`

and a suit of

`suits[i]`

.

The following are the types of

poker hands

you can make from best to worst:

"Flush"

: Five cards of the same suit.

"Three of a Kind"

: Three cards of the same rank.

"Pair"

: Two cards of the same rank.

"High Card"

: Any single card.

Return

a string representing the

best

type of

poker hand

you can make with the given cards.

Note

that the return values are

case-sensitive

.

Example 1:

Input:

ranks = [13,2,3,1,9], suits = ["a","a","a","a","a"]

Output:

"Flush"

Explanation:

The hand with all the cards consists of 5 cards with the same suit, so we have a "Flush".

Example 2:

Input:

ranks = [4,4,2,4,4], suits = ["d","a","a","b","c"]

Output:

"Three of a Kind"

Explanation:

The hand with the first, second, and fourth card consists of 3 cards with the same rank, so we have a "Three of a Kind". Note that we could also make a "Pair" hand but "Three of a Kind" is a better hand. Also note that other cards could be used to make the "Three of a Kind" hand.

Example 3:

Input:

ranks = [10,10,2,12,9], suits = ["a","b","c","a","d"]

Output:

"Pair"

Explanation:

The hand with the first and second card consists of 2 cards with the same rank, so we have a "Pair". Note that we cannot make a "Flush" or a "Three of a Kind".

Constraints:

`ranks.length == suits.length == 5`

`1 <= ranks[i] <= 13`

`'a' <= suits[i] <= 'd'`

No two cards have the same rank and suit.

## Code Snippets

**C++:**

```
class Solution {
public:
    string bestHand(vector<int>& ranks, vector<char>& suits) {

    }
};
```

**Java:**

```
class Solution {
    public String bestHand(int[] ranks, char[] suits) {

    }
}
```

**Python3:**

```
class Solution:
    def bestHand(self, ranks: List[int], suits: List[str]) -> str:
```

## Python:

```
class Solution(object):
    def bestHand(self, ranks, suits):
        """
        :type ranks: List[int]
        :type suits: List[str]
        :rtype: str
        """
```

## JavaScript:

```
/**
 * @param {number[]} ranks
 * @param {character[]} suits
 * @return {string}
 */
var bestHand = function(ranks, suits) {

};
```

## TypeScript:

```
function bestHand(ranks: number[], suits: string[]): string {

};
```

## C#:

```
public class Solution {
    public string BestHand(int[] ranks, char[] suits) {

    }
}
```

## C:

```
char* bestHand(int* ranks, int ranksSize, char* suits, int suitsSize) {

}
```

**Go:**

```
func bestHand(ranks []int, suits []byte) string {  
  
}
```

**Kotlin:**

```
class Solution {  
    fun bestHand(ranks: IntArray, suits: CharArray): String {  
  
    }  
}
```

**Swift:**

```
class Solution {  
    func bestHand(_ ranks: [Int], _ suits: [Character]) -> String {  
  
    }  
}
```

**Rust:**

```
impl Solution {  
    pub fn best_hand(ranks: Vec<i32>, suits: Vec<char>) -> String {  
  
    }  
}
```

**Ruby:**

```
# @param {Integer[]} ranks  
# @param {Character[]} suits  
# @return {String}  
def best_hand(ranks, suits)  
  
end
```

**PHP:**

```
class Solution {
```

```

/**
 * @param Integer[] $ranks
 * @param String[] $suits
 * @return String
 */
function bestHand($ranks, $suits) {

}
}

```

### Dart:

```

class Solution {
  String bestHand(List<int> ranks, List<String> suits) {

  }
}

```

### Scala:

```

object Solution {
  def bestHand(ranks: Array[Int], suits: Array[Char]): String = {

  }
}

```

### Elixir:

```

defmodule Solution do
  @spec best_hand(ranks :: [integer], suits :: [char]) :: String.t
  def best_hand(ranks, suits) do

  end
end

```

### Erlang:

```

-spec best_hand(Ranks :: [integer()], Suits :: [char()]) ->
  unicode:unicode_binary().
best_hand(Ranks, Suits) ->
.

```

## Racket:

```
(define/contract (best-hand ranks suits)
  (-> (listof exact-integer?) (listof char?) string?)
)
```

## Solutions

### C++ Solution:

```
/*
 * Problem: Best Poker Hand
 * Difficulty: Easy
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class Solution {
public:
    string bestHand(vector<int>& ranks, vector<char>& suits) {

    }
};
```

### Java Solution:

```
/**
 * Problem: Best Poker Hand
 * Difficulty: Easy
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class Solution {
    public String bestHand(int[] ranks, char[] suits) {
```

```
}  
}
```

### Python3 Solution:

```
"""  
Problem: Best Poker Hand  
Difficulty: Easy  
Tags: array, string, hash  
  
Approach: Use two pointers or sliding window technique  
Time Complexity: O(n) or O(n log n)  
Space Complexity: O(n) for hash map  
"""  
  
class Solution:  
    def bestHand(self, ranks: List[int], suits: List[str]) -> str:  
        # TODO: Implement optimized solution  
        pass
```

### Python Solution:

```
class Solution(object):  
    def bestHand(self, ranks, suits):  
        """  
        :type ranks: List[int]  
        :type suits: List[str]  
        :rtype: str  
        """
```

### JavaScript Solution:

```
/**  
 * Problem: Best Poker Hand  
 * Difficulty: Easy  
 * Tags: array, string, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */
```

```

*/

/**
 * @param {number[]} ranks
 * @param {character[]} suits
 * @return {string}
 */
var bestHand = function(ranks, suits) {

};

```

### TypeScript Solution:

```

/**
 * Problem: Best Poker Hand
 * Difficulty: Easy
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

function bestHand(ranks: number[], suits: string[]): string {

};

```

### C# Solution:

```

/*
 * Problem: Best Poker Hand
 * Difficulty: Easy
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

public class Solution {
    public string BestHand(int[] ranks, char[] suits) {

```

```
}  
}
```

### C Solution:

```
/*  
 * Problem: Best Poker Hand  
 * Difficulty: Easy  
 * Tags: array, string, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
char* bestHand(int* ranks, int ranksSize, char* suits, int suitsSize) {  
  
}
```

### Go Solution:

```
// Problem: Best Poker Hand  
// Difficulty: Easy  
// Tags: array, string, hash  
//  
// Approach: Use two pointers or sliding window technique  
// Time Complexity: O(n) or O(n log n)  
// Space Complexity: O(n) for hash map  
  
func bestHand(ranks []int, suits []byte) string {  
  
}
```

### Kotlin Solution:

```
class Solution {  
    fun bestHand(ranks: IntArray, suits: CharArray): String {  
  
    }  
}
```

### Swift Solution:

```
class Solution {  
    func bestHand(_ ranks: [Int], _ suits: [Character]) -> String {  
  
    }  
}
```

### Rust Solution:

```
// Problem: Best Poker Hand  
// Difficulty: Easy  
// Tags: array, string, hash  
//  
// Approach: Use two pointers or sliding window technique  
// Time Complexity: O(n) or O(n log n)  
// Space Complexity: O(n) for hash map  
  
impl Solution {  
    pub fn best_hand(ranks: Vec<i32>, suits: Vec<char>) -> String {  
  
    }  
}
```

### Ruby Solution:

```
# @param {Integer[]} ranks  
# @param {Character[]} suits  
# @return {String}  
def best_hand(ranks, suits)  
  
end
```

### PHP Solution:

```
class Solution {  
  
    /**  
     * @param Integer[] $ranks  
     * @param String[] $suits  
     * @return String  
     */  
}
```

```
function bestHand($ranks, $suits) {

}

}
```

### Dart Solution:

```
class Solution {
  String bestHand(List<int> ranks, List<String> suits) {

  }
}
```

### Scala Solution:

```
object Solution {
  def bestHand(ranks: Array[Int], suits: Array[Char]): String = {

  }
}
```

### Elixir Solution:

```
defmodule Solution do
  @spec best_hand(ranks :: [integer], suits :: [char]) :: String.t
  def best_hand(ranks, suits) do

  end
end
```

### Erlang Solution:

```
-spec best_hand(Ranks :: [integer()], Suits :: [char()]) ->
  unicode:unicode_binary().
best_hand(Ranks, Suits) ->
  .
```

### Racket Solution:

```
(define/contract (best-hand ranks suits)
  (-> (listof exact-integer?) (listof char?) string?))
```

