Cactus Kev's Poker Hand Evaluator

A while ago, I decided to take a shot at writing a poker hand evaluator in the programming language "C". There are already numerous evaluators out there, but I had an idea for an algorithm that might be faster than anything already out there. The basic concept is to write a routine that would take a five card poker hand and return it's overall "value". This is extremely valuable in any poker-related software, since the code will constantly be comparing various player's hand with each other to determine the "winner". Here is my concept on how I thought I could write a fast evaluator.

Okay, before we start digging into my algorithm, please read this first. I usually get one or two emails every month from somebody interested in my algorithm or poker code. And they typically ask me if I happen to have a seven-card poker evaluator. The answer is yes. I did indeed write a seven-card hand evaluator a few years after writing the five-card one. It used a completely new algorithm, completely unrelated to my five-card version. I just never posted it on my web site because (a) it was pretty lengthy, and (b) I was too lazy to write up all that HTML.

*However, one day, I got an email from an actual poker software company from Canada called **Poker Academy**. They also wanted to know if I had written a seven-card evaluator, and if so, could they test it to see if it was faster than their current code? We converted my code from 'C' to Java, and they gave it a whirl. Turns out it was about three times faster than what they were currently using, so they asked if I'd be willing to sell/license it to their company for use in the next version of their software. We worked out a contract and the deal was inked. If you visit their site, download version 2.5, and select Version History, you can see my name in the notes for the February 15, 2006 2.5.0 Release (b164). The downside of all this, is that I cannot pass on my seven-card evaluator algorithm to any curious poker math-geeks who stumble upon my site. So don't email me asking for the code or algorithm, because I can't give it to you. Sorry, mate.*

First off, any person who has studied combinatorics will know that there are `C(52,5)`, or 2,598,960 possible unique poker hands, even more combinations than you get even from when you play slots. I realized that even though there are nearly 2.6 million unique hands, many of those hands actually have the *same* poker hand value. In other words, somebody holding an AJ942 flush in spades has the exact same value hand as somebody with an AJ942 flush in clubs. Even though both hands are unique, they still hold the identical value, and would therefore "tie" in poker games.

Here's another way to look at it. Suppose you were able to round up 2,598,960 of your friends on a football field, and you gave each of them one of the unique 2,598,960 poker hands to hold. You then yell in a loud voice, asking everyone to compare their hand with everybody else's hand. (This will take some time, of course :) Anyway, once they are done, you ask the person holding the best hand to step forward. Of course, four people will step forward, each holding a Royal Flush in each of the four suits. They all "tied" for having the best hand. You group those people together, tie a rope around them, label them with the number "1", and ask them to leave the field. Now, you ask your friends to compare hands again and figure out who has the best hand now. This time, four more people come forward, each holding a King-High Straight Flush. You group them together, label them with a "2", and escort them off the field. You keep repeating this process of finding out who has the highest hand and marking them with the next number. The next eight queries should yield:

- 3: four people holding Queen-High Straight Flushes
- 4: four people holding Jack-High Straight Flushes
- 5: four people holding Ten-High Straight Flushes
- 6: four people holding Nine-High Straight Flushes
- 7: four people holding Eight-High Straight Flushes
- 8: four people holding Seven-High Straight Flushes

- 9: four people holding Six-High Straight Flushes
- 10: four people holding Five-High Straight Flushes

Now, when you query for the eleventh time, four people should step forward, each holding Four Aces with a King kicker. Then, Four Aces with a Queen kicker. Then, Four Aces with a Jack kicker. And so on, until we have Four Aces with a Deuce kicker. Next comes Four Kings with an Ace kicker. This continues until we finally get down to Four Deuces with a Trey kicker.

- 11: four people holding Four Aces with a King kicker
- 12: four people holding Four Aces with a Queen kicker
- ...
- 165: four people holding Four Deuces with a Four kicker
- 166: four people holding Four Deuces with a Trey kicker

Next, for the 167th query, we find twenty-four people coming forward, each holding a Full House of Aces over Kings. Then 24 people with Aces over Queens. And so on, for the remaining Full House hands.

- 167: twenty-four people holding Full Houses of Aces over Kings
- 168: twenty-four people holding Full Houses of Aces over Queens
- ...
- 321: twenty-four people holding Full Houses of Deuces over Fours
- 322: twenty-four people holding Full Houses of Deuces over Treys

Note that by using combinatorics, we can verify these totals. Assume you have a Full House of Aces over Kings. There are `C(4,3)` possible ways to select three Aces out of a possible four, and `C(4,2)` possible ways to choose two Kings out of a possible four. This yields 4 x 6, or 24 possible combinations of such hands.

For query #323, four people will step foward, each holding a `AKQJ9` Flush. Then, four people holding `AKQJ8` Flushes, and so forth.

- 323: four people holding AKQJ9 Flushes
- 324: four people holding AKQJ8 Flushes
- ...
- 1598: four people holding 76432 Flushes
- 1599: four people holding 75432 Flushes

For our next query, we get a whopping 1020 people step forward with Ace High Straights. This is followed by another 1020 people with King High Straights, and so on down the line.

- 1600: 1,020 people holding Ace High Straights
- 1601: 1,020 people holding King High Straights
- ...
- 1608: 1,020 people holding Six High Straights
- 1609: 1,020 people holding Five High Straights

Next comes all the Three of a Kinds, 64 people each.

- 1610: sixty-four people holding AAAKQ
- 1611: sixty-four people holding AAAKJ
- ...
- 2466: sixty-four people holding 22253

- 2467: sixty-four people holding 22243

Then the people in groups of 144, each holding Two Pair.

- 2468: 144 people holding AAKKQ
- 2469: 144 people holding AAKKJ
- ...
- 3324: 144 people holding 33225
- 3325: 144 people holding 33224

Almost done! Next comes the One Pair hands, 384 people each.

- 3326: 384 people holding AAKQJ
- 3327: 384 people holding AAKQT
- ...
- 6184: 384 people holding 22643
- 6185: 384 people holding 22543

And finally, we have the "dreck" High Card hands, with 1,020 people per hand. Notice they are identical to the above "Flush" hands, except these are not flushes.

- 6186: 1,020 people holding AKQJ9
- 6187: 1,020 people holding AKQJ8
- ...
- 7461: 1,020 people holding 76432
- 7462: 1,020 people holding 75432

There! After enumerating and collapsing all the 2.6 million unique five-card poker hands, we wind up with just 7462 distinct poker hand values. That was pretty surprising to me when I saw the final total. If you are interested in seeing a table of all 7462 values, you'll find it here. This is how the numbers stack up:

Hand Value	Unique	Distinct
Straight Flush	40	10
Four of a Kind	624	156
Full Houses	3744	156
Flush	5108	1277
Straight	10200	10
Three of a Kind	54912	858
Two Pair	123552	858
One Pair	1098240	2860
High Card	1302540	1277
TOTAL	**2598960**	**7462**

Once I determined that there were only 7462 distinct values of poker hands, I needed a way to quickly transform each of the 2,598,960 unique five-card poker hands into its actual value. To complicate matters, the algorithm needed to be order independant. In other words, if I pass the cards **Kd Qs Jc Th 9s** to my evaluator, it must generate the value 1601. However, if I change the order of the cards in any fashion, it must *still* return the value of 1601. Mixing up the five cards does not change the overall

value of the hand. At first, I thought that I could always simply sort the hand first before passing it to the evaluator; but sorting takes time, and I didn't want to waste any CPU cycles sorting hands. I needed a method that didn't care what order the five cards were given as.

After a lot of thought, I had a brainstorm to use prime numbers. I would assign a prime number value to each of the thirteen card ranks, in this manner:

Rank	Deuce	Trey	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King	Ace
Prime	2	3	5	7	11	13	17	19	23	29	31	37	41

The beauty of this system is that if you ***multiply*** the prime values of the rank of each card in your hand, you get a unique product, regardless of the order of the five cards. In my above example, the King High Straight hand will *always* generate a product value of 14,535,931. Since multiplication is one of the fastest calculations a computer can make, we have shaved hundreds of millseconds off our time had we been forced to sort each hand before evaluation.

One last step is required, however, before multiplying our prime rank values. We must first check to see if all five cards are of the same suit. It is extremely important that our evaluator realize that the value of a `KQJT9` hand is *much* higher if all the suits are the same.

Okay. At this point, I was ready to start writing some code. I decided to use the following bit scheme for a single card, where each card is of type **`integer`** (and therefore, four bytes long).

```
+--------+--------+--------+--------+
|xxxbbbbb|bbbbbbbb|cdhsrrrr|xxpppppp|
+--------+--------+--------+--------+

p = prime number of rank (deuce=2,trey=3,four=5,...,ace=41)
 r = rank of card (deuce=0,trey=1,four=2,five=3,...,ace=12)
  cdhs = suit of card (bit turned on based on suit of card)
         b = bit turned on depending on rank of card
```

Using such a scheme, here are some bit pattern examples:

```
xxxAKQJT 98765432 CDHSrrrr xxPPPPPP
00001000 00000000 01001011 00100101    King of Diamonds
00000000 00001000 00010011 00000111    Five of Spades
00000010 00000000 10001001 00011101    Jack of Clubs
```

Now, for my evaluator function, I would pass in five integers, each representing one of the five cards in the hand to evaluate. We will label these **`c1`** through **`c5`**. First, we check to see if all the suits are the same. If so, then we have either a *Flush* (or *Straight Flush*). The quickest way to calculate this is by evaluating this:

```
c1 AND c2 AND c3 AND c4 AND c5 AND 0xF000
```

This is bit-wise AND, not a boolean AND, by the way. If the above expression yields a value of zero, then we don't have a *Flush*. If it is non-zero, then we do. Assuming we have a *Flush*, I needed a fast way to calculate the hand's actual value (i.e. convert it to a value from 1 to 7462). Table lookups are one of the fastest ways to generate values based on index keys, so I used the following coding technique.

First, I do another bitwise OR of all five cards, and then bit shift that value 16 bits to the right:

```
q = (c1 OR c2 OR c3 OR c4 OR c5) >> 16
```

Note that if we have a *Flush*, then all five card ranks are guaranteed to be unique. We should have exactly ***five*** bits set in our shifted value. The smallest bit pattern would be **`0x001F`** (decimal 31), and the highest would be **`0x1F00`** (decimal 7936). I created a lookup array containing 7937 elements, and then populated it with the correct hand values based on each valid bit pattern. Let's call this array **`flushes[]`**. If you checked the value of **`flushes[31]`**, you should find the value 9, because 9 is the distinct value for a Six-High Straight Flush. As another example, let's say you were holding an AKQJ9 Flush. The bit pattern for that hand would be **`0x1E80`**, or decimal 7808. This means that **`flushes[7808]`** should hold the value 323, which is the distinct value for AKQJ9 Flushes. Obviously, there are a lot of entries in this array that will never be accessed. That is the price of lookup tables. You gain speed in favor of "wasted" space. However, since the highest distinct hand value is 7462, we can make our array up of type **`short`**. That means, our array will take up only 15874 bytes. Not very large at all.

If we should determine that we do not have *Flush* or *Straight Flush* hand, we move on to tackle *Straight* and *High Card* hands. Again, we use a lookup table for speed. We create another array, which will we will call **`unique5[]`**. We use the same index **`q`** as the index into this new array. If the value found there is zero, then we do not have a *Straight* or *High Card* hand. If it is non-zero, then it holds the actual hand's distinct value. For example, a King-High Straight has a bit pattern of **`0x0F80`**, or decimal 3968. This means that **`unique5[3968]`** should hold the value 1601. The absolute worst poker hand of 75432 (unsuited) would yield **`unique5[0x002F] = unique5[47]`**, or 7462.

With just two easy calculations and two quick lookups, we have eliminated 2574 out of the 7462 possible distinct hand values. That's a little over a third of all the hands. Now it's time to use our trick with prime numbers. We take each card, extract its prime number value, and then multiply them all together:

```
q = (c1 AND 0xFF) * (c2 AND 0xFF) * ... * (c5 AND 0xFF)
```

Again, these are bitwise ANDs. The resulting values will range from the small to the very large. The smallest value is obtained if you hold the hand **22223** *(four Deuces with a Trey kicker)*. Multiplying its prime values yields the number 48. The largest value comes with holding **AAAAK**. Doing the math, we get 104,553,157 (41 x 41 x 41 x 41 x 37). These numbers are way too large to use a lookup table, so we must find some other method to convert these values to their proper distinct hand value. Since there are only 4888 remaining hand values (7462 minus 2574), I decided to use binary search algorithm. It's fast, with a complexity of O(*log n*). I took the remaining 4888 hands, calculated the prime multiplication value for each, placed them in an array, and then sorted it. When I need to find a distinct hand value, after all previous efforts fail, I do a binary search, which returns an index. That index is used in a regular array, which holds the remaining distinct values.

That's it! I coded this algorithm, and ran it against some of the other poker evaluators out there, and it beat them all. Now all you need to do is get over to an online casino to give it a try! I'm pretty proud of it, and kudos to you if you've actually read this far. For your prize, you get to download the actual source code to experiment with.

> **LATE BREAKING NEWS!!!** Paul Senzee of Florida decided that he could speed up my evaluator by using a pre-computed perfect hash function instead of a binary search for those final 4888 hand values. He says he obtained a speedup factor of 2.7x! Not too shabby! Anyway, you can read about his optimization and download his new evaluator code here.

- **poker.h**
- **pokerlib.c**
- **arrays.h**
- **allfive.c**
- **Makefile**