UNIVERSIDADE FEDERAL DO ABC

PROJETO – Dezembro 2016

ALGORITMOS E ESTRUTURAs DE DADOS II

MCTA002-13

Descrição: Comparação entre diferentes abordagens de estruturas de dados aplicado ao problema de avaliar a força relativa de uma mão de pôquer.

Camila Ferreira Rodrigues
Fabricio Ribeiro Bueno Yamamoto
Victor Yoshidi Sanches Natumi

RA:11082810, 11036814, 11053310

Introdução

Este trabalho tem como objetivo analisar três diferentes abordagens de, dada uma determinada mão de pôquer com cinco cartas, quantificar a força dessa mão seguindo as regras do jogo.

Cada mão de pôquer pode estar dentre os seguintes 9 grupos, em ordem decrescente de força:

1. Straight Flush : Cinco cartas na sequência, do mesmo naipe. Em caso de uma mão similar, o valor da carta mais alta da sequência vence.

2. Quadra : Quatro cartas de mesmo valor, e uma outra carta como 'kicker'. Em caso de empate, o jogador com a maior carta restante ('kicker') vence.

3. Full House : Três cartas do mesmo valor, e duas outras cartas diferentes de mesmo valor. Em caso de empate, as três cartas de mesmo valor mais altas vencem.

4. Flush : Cinco cartas do mesmo naipe, não em sequência. Em caso de empate, o jogador com a maior carta de maior valor vence.

5. Sequência : Cinco cartas de naipes diferentes em sequência. Em caso de uma mão similar, o valor da carta mais alta da sequência vence.

6. Trinca : Três cartas do mesmo valor, e duas outras cartas não relacionadas. Em caso de empate, o jogador com a maior carta restante, e caso necessário a segunda mais alta carta restante ('kicker') vence.

7. Dois Pares : Duas cartas de mesmo valor e mais duas cartas diferentes de mesmo valor, além do kicker. No caso dos jogadores terem Dois Pares idênticos, o maior kicker vence.

8. Par : Duas cartas do mesmo valor, e três outras cartas não relacionadas. Em caso de empate, o jogador com a maior carta restante, e caso necessário a segunda ou terceira carta restante mais alta ('kicker') vence.

9. Carta Alta : Qualquer mão que não esteja nas categorias acima. Em caso de empate, a carta mais alta vence, como um 'as-alto'.

Nota-se que, de acordo com as regras acima, a ordem das cartas em uma mão é irrelevante para classificá-la entre os grupos explanados, restringindo nosso cenário para um problema de combinação. Portanto, considerando um baralho comum de 52 cartas, temos que existem C(52,5) possíveis mãos, ou 2.598.960 mãos únicas de pôquer.

1ª Abordagem : Naive evaluator

Esta é, provavelmente, a abordagem mais comum ao problema.

Não existe uma estratégia única neste tipo de abordagem, qualquer tipo de algoritmo que tente classificar uma mão de pôker de uma maneira que intuitivamente faça sentido pode ser definido como um "Naive evaluator".

Para este trabalho iremos assumir uma implementação que, acreditamos, seja o mais próximo do senso comum. [2]

- Leitura. Armazenamos os valores de rank e naipe, checando por cartas inexistentes ou duplicadas.
- Classificação. Determina-se qual a classificação da mão de acordo com testes específicos para cada grupo.
- Ordenação. Ordenamos a mão.
- Comparação. Quando duas mãos são comparadas, primeiro compare os grupos, e então se forem da mesma categoria, compare cada carta restante individualmente para determinar qual das duas mãos é mais forte.

Código

```
int main(void)
{
  int r1[3], r2[3];
  printf("-------First hand--------\n");
  read_cards();
  analyze_hand();
  print_result(r1);
  qsort(hand1,sizeof(hand1)/sizeof(*hand1), sizeof(*hand1), comp);

  printf("-------Second hand--------\n");
  cont = 2;
  read_cards();
  analyze_hand();
  print_result(r2);
  qsort(hand2,sizeof(hand2)/sizeof(*hand2), sizeof(*hand2), comp);

  evaluate(r1,r2,hand1,hand2);
}
```

Listing 1: Função principal

```
int comp (const void * elem1, const void * elem2)
{
    int f = *((int*)elem1);
    int s = *((int*)elem2);
    if (f > s) return  1;
    if (f < s) return -1;
    return 0;
}
```

Listing 2: Ordenação

```
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS], bad_card;
  char ch, rank_ch, suit_ch;
  int rank, suit, i = 0, cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
    for (suit = 0; suit < NUM_SUITS; suit++)    card_exists[rank][suit] = false;
  }

  for (suit = 0; suit < NUM_SUITS; suit++)      num_in_suit[suit] = 0;

  while (cards_read < NUM_CARDS) {
    bad_card = false;
    printf("Enter a card: ");
    rank_ch = getchar();

    switch (rank_ch) {
      case '0':           exit(EXIT_SUCCESS);
      case '2':           rank = 0; break;
      case '3':           rank = 1; break;
      case '4':           rank = 2; break;
      case '5':           rank = 3; break;
      case '6':           rank = 4; break;
      case '7':           rank = 5; break;
      case '8':           rank = 6; break;
      case '9':           rank = 7; break;
      case 't': case 'T': rank = 8; break;
      case 'j': case 'J': rank = 9; break;
      case 'q': case 'Q': rank = 10; break;
      case 'k': case 'K': rank = 11; break;
      case 'a': case 'A': rank = 12; break;
      default:            bad_card = true;
    }

    if (cont == 1){
      hand1[i] = rank;
      i++;
    } else {
      hand2[i] = rank;
      i++;
    }

    suit_ch = getchar();
    switch (suit_ch) {
      case 'c': case 'C': suit = 0; break;
      case 'd': case 'D': suit = 1; break;
      case 'h': case 'H': suit = 2; break;
      case 's': case 'S': suit = 3; break;
      default:            bad_card = true;
    }

    while ((ch = getchar()) != '\n')
      if (ch != ' ') bad_card = true;

    if (bad_card)                       printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])   printf("Duplicate card; ignored.\n");
    else {
      num_in_rank[rank]++;
      num_in_suit[suit]++;
      card_exists[rank][suit] = true;
      cards_read++;
    }
  }
}
```

Listing 3: Leitura

```
void analyze_hand(void)
{
  int  rank, suit, num_consec = 0;

  straight = false; flush = false;  four = false;
  three = false;    pairs = 0;

  // check for flush
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS) flush = true;

  // check for straight
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = true;
    return;
  }

  // check for 4-of-a-kind, 3-of-a-kind, and pairs
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4){
      four = true;
      quadra = rank;
    }
    if (num_in_rank[rank] == 3){
      three = true;
      trinca = rank;
    }
    if (num_in_rank[rank] == 2){
      pairs++;
      if (pairs == 1){
        par1 = rank;
      } else if (pairs == 2) {
        par2 = rank;
      }
    }
  }
}
```

Listing 4: Classificação

```
void evaluate(int *r1, int *r2, int *hand1, int *hand2)
{
  if (r1[0] != r2[0]) {
    if (r1[0] > r2[0])  printf ("First hand wins");
    else                printf("Second hand wins");
  }
  else {
    if (r1[0] == 9 || r1[0]== 5){ //sf , s
      if (hand1[4] > hand2[4])          printf ("First hand wins");
      else if (hand1[4] < hand2[4])     printf ("Second hand wins");
      else if (hand1[4] == hand2[4])    printf ("Tie");
    }
    if (r1[0] == 6  || r1[0] == 1){ //f, hc
      if (hand1[4] != hand1[4]){
        if (hand1[4] > hand2[4])    printf ("First hand wins");
        else                        printf ("Second hand wins");
      }
      else {
        if (hand1[3] != hand2[3]){
          if (hand1[3] > hand2[3])  printf ("First hand wins");
          else                      printf ("Second hand wins");
        }
        else {
          if (hand1[2] != hand2[2]){
            if (hand1[2] > hand2[2])    printf ("First hand wins");
```

```
            else                          printf ("Second hand wins");
          }
        }
      }
    }
    else if (r1[0] == 8) { //four
      if (r1[1] > r2[1])        printf ("First hand wins");
      else if (r1[1] < r2[1])   printf ("Second hand wins");
    }
    else if (r1[0] == 7) { //full house
      if (r1[1] > r2[1])    printf ("First hand wins");
      else                  printf ("Second hand wins");
    }
    else if (r1[0] == 4 || r1[0] == 3) { //trinca ou 2 pares
      if (r1[1] != r2[1]) {
        if (r1[1] > r2[1])  printf ("First hand wins");
        else                printf ("Second hand wins");
      }
      else {
        if (r1[2] > r2[2])  printf ("First hand wins");
        else                printf ("Second hand wins");
      }
    }
    else if (r1[0] == 2) { //1 par
      if (r1[1] != r2[1]) {
        if (r1[1] > r2[1])  printf ("First hand wins");
        else                printf ("Second hand wins");
      }
      else {
        if (hand1[4] > hand2[4])    printf ("First hand wins");
        else                        printf ("Second hand wins");
      }
    }
  }
}
```

Listing 5: Comparação

A versão que abordamos neste trabalho, por envolver a ordenação das cartas, tem complexidade proporcional a $O(nlogn)$, porém é válido salientar que por contar com muitas condicionais, iterações e laços, sua eficiência tende a diminuir ainda mais.

Seu melhor caso seria se as duas mãos tivessem categorias diferentes, pois bastaria uma única comparação para determinar a mão vencedora. Já no caso de as duas mãos pertencerem a categorias iguais e serem muito parecidas nos valores das cartas, mais comparações seriam necessárias, aumentando assim o tempo de execução do algoritmo.

Ao alterarmos o código para, em vez de comparar duas mãos, somente classificasse em categorias todas as mãos possíveis, o tempo médio que obtivemos foi de cerca de 1300 milisegundos. E caso as comparações entre todas as mãos fossem feitas (necessário para quantificar um valor de força relativa para cada mão), a complexidade seria proporcional a $O(n^2)$, onde n é 2.598.960.

Os Naive Evaluators raramente são utilizados, pois são dificeis de implementar, ineficientes e existem muitas outras versões de avaliadores melhores.

2ª Abordagem : Cactus Kev

Esta seção abordará a estratégia desenvolvidada por Kevin Suffecool [1].

Sabemos que existem 2.598.960 mãos únicas de pôquer, entretanto, se formos ver, muitas dessas mãos possuem o mesmo valor.

Um flush com naipes diferentes, por exemplo, apesar de serem mãos diferentes, empatariam no seu valor.

Ao percorrer todas as quase 2,6 milhões de mãos, é possível perceber que há apenas 7462 valores de mãos diferentes no pôquer (este cálculo não será abordado neste trabalho).

Temos então a seguinte tabela que corresponde aos grupos de mãos, únicas e distintas:

Grupo da mão	Mãos Únicas	Mãos Distintas
Straight Flush	40	10
Quadra	624	156
Full House	3744	156
Flush	5108	1277
Sequência	10200	10
Trinca	54912	858
Dois Pares	123552	858
Um Par	1098240	2860
Carta Alta	1302540	1277
Total	2598960	7462

Portanto, nosso problema se limita em relacionar uma determinada mão de cinco cartas com seu respectivo valor em uma escala de 0 a 7462.

Representação das cartas em binário e com números primos

O cactus kev's tem como propósito pegar qualquer uma desses milhões de mãos e retornar seu resultado correspondente, dentre esses 7462 valores.

Ao pensar no algoritmo, é necessário perceber que a mão deve poder ser dada como entrada em qualquer ordem, e independente da ordem dada, o valor é o mesmo.

Seria possível ordenar as cartas antes de executar o algoritmo, mas o cactus kev's não gasta tempo ordenando as cartas.

O cactus kev's é dependente de lookup tables, e é nelas que ele encontra sua eficiência.

Cada carta possui um valor primo e binário:

Carta	Primo	Binário
2	2	0000
3	3	0001
4	5	0010
5	7	0011
6	11	0100
7	13	0101
8	17	0110
9	19	0111
10	23	1000
J	29	1001
Q	31	1010
K	37	1011
A	41	1100

A ideia seria que se multiplicar os valores primos dados, independente da ordem, o resultado será sempre o mesmo.

Caso tivéssemos uma mão AJ942 nessa ordem, e multiplicássemos os valores, seria o mesmo resultado pra qualquer ordem dada desta mesma mão.

Isso otimiza o algoritmo, visto que a multiplicação de valores é uma operação muito mais rápida do que uma ordenação.

Entretanto, o que aconteceria caso os naipes fossem iguais? Como dizer o valor caso for um flush ou straight (sequência)?

No cactus kev's, cada carta é representada por uma variável do tipo integer (4 bytes).

Onde é preenchido por 'x' é sempre 0.

máscara			
xxxAKQJT	98765432	CDHSrrrr	xxPPPPPP
xxxbbbbb	bbbbbbbb	cdhsrrrr	xxpppppp

Os dois primeiros bytes são utilizados para identificar o rank da carta.

Para isso, é utilizado uma "máscara"do tipo xxxAKQJT 98765432.

Ou seja, todos os bits são 0, menos o que representa o valor da sua carta, este tem valor 1.

No terceiro byte, os 4 primeiros bits são direcionados para a identificação do naipe.

O cactus kev's é escrito em inglês, então a inicial dos naipes seriam clubs (paus), diamonds (ouros), hearts (copas) e spades (espadas).

Ao colocar o bit como 1, significa que ele é daquele naipe.

Os 4 últimos bits (rrrr), também informam o rank da carta, mas dessa vez em binário.

Já o último byte (xxPPPPPP) é responsável por informar o rank da carta em binário a partir da tabela de números primos.

Determinação de valor caso Flush

Para saber se possui flush é executada a seguinte operação:

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

nota: é um AND bitwise e 0xF000 = 1111000000000000

xxxAKQJT	98765432	CDHSrrrr	xxPPPPPP	
00000000	00000000	11110000	00000000	(0xF000)
00001000	00000000	01001011	00100101	(Rei de Ouros)
00000000	00001000	01000011	00000111	(Cinco de Ouros)
00000010	00000000	01001001	00011101	(Valete de Ouros)
00010000	00000000	01001100	00101001	(Ás de Ouros)
00000100	00000000	01001010	00011111	(Rainha de Ouros)
00000000	00000000	01000000	00000000	(Resultado do AND)

Caso o resultado seja diferente de 0, é um flush.

Para calcular o valor do flush, o cactus kev's possui uma lookup table apenas para flushes.

Porém, antes disso é necessário saber o valor das cartas, que é feito a partir da seguinte operação:

q = (c1 OR c2 OR c3 OR c4 OR c5) » 16

nota: OR bitwise tal que o resultado será apenas os 16 bits da esquerda

xxxAKQJT	98765432	CDHSrrrr	xxPPPPPP	
00001000	00000000	01001011	00100101	(Rei de Ouros)
00000000	00001000	01000011	00000111	(Cinco de Ouros)
00000010	00000000	01001001	00011101	(Valete de Ouros)
00010000	00000000	01001100	00101001	(Ás de Ouros)
00000100	00000000	01001010	00011111	(Rainha de Ouros)
00011110	00001000	xxxxxxxx	xxxxxxxx	(Resultado do OR » 16)

Sabendo que, por ser um flush, todas minhas cartas possuem o mesmo naipe e valores distintos, é possível concluir que necessariamente meu resultado sempre terá 5 bits acesos.

Então, podemos dizer que:

0000000 0011111 (0x001F = 31) é o menor valor possível para um flush
1111100 0000000 (0x1F00 = 7936) é o maior valor possível para um flush

Portanto, para adquirir o valor do flush/straight flush, a lookup table *flushes[]* deve possuir 7937 elementos distintos.

Ele deverá ser acessado no index *flushes[q]*. E então lá você tem o valor da sua mão, dentre os 7462 possíveis.

Muitos desses elementos nunca serão acessados, e esse é o preço da lookup table, você ganha velocidade mas "desperdiça"memória.

Entretanto, essa lookup table pode ser do tipo numérico *short* (já que só são possíveis 7462 valores de mãos), o que dá apenas 15874 bytes.

Determinação de valor caso Sequência ou Carta Alta

Caso não seja um flush, mas mesmo assim o q tenha 5 bits acesos, ou seja, as cartas possuem valores distintos, é necessário utilizar outra lookup table.

A lookup table *unique5[]* é utilizada para dar valor as mãos Sequência e Carta Alta (desde que possua valores distintos).

Ao acessar *unique5[q]* você possui o valor da sua mão.

Com essas duas lookup tables já eliminamos 2574 valores de 7462.

Determinação de valor caso qualquer outro grupo

Caso a mão não seja flush e nem sequência, utilizamos os números primos para fazer o cálculo.

Para isso, segue a seguinte operação:

q = (c1 AND 0xFF) * (c2 AND 0xFF) * (c3 AND 0xFF) * (c4 AND 0xFF) * (c5 AND 0xFF)

nota: bitwise AND com 0xFF deixa apenas os últimos 8 bits (xxPPPPPP).

Já que os valores são muito grandes, não é possível criar uma lookup table diretamente, veja que a menor multiplicação possível seria caso a mão fosse 22223, que multiplicando os valores primos daria 48.

A maior multiplicação seria uma mão AAAAK, que daria 41*41*41*41*37 = 104.553.157.

Para isso é criado mais um array *products[]* e um *values[]*, ambos de 4888 espaços correspondentes as 4888 mãos possíveis restantes (7462 - 2547).

O *products[]* possui valores a partir da multiplicação dos números primos das mãos, enquanto *values[]* possui o valor das mãos, de forma que estejam correlacionados no mesmo índice.

Então, dada uma mão que não é um flush e nem uma sequência, é calculado a multiplicação dos primos, o cactus kev's utiliza uma binary search para procurar esse valor dentro do array *products[]*. Ao encontrar o valor, seu índice é utilizado em *values[]*, que possui o valor da mão.

Código

```
void init_deck( int *deck )
{
    int i, j;
    int n = 0, suit = 0x8000;

    for ( i = 0; i < 4; i++, suit >>= 1 )
    {
        for ( j = 0; j < 13; j++, n++ )
        {
            deck[n] = primes[j] | (j << 8) | suit | (1 << (16+j));
        }
    }
}
```

Listing 6: Inicialização do deck com a representação Cactus Kev

```
int findit( int key )
{
    int low = 0, high = 4887, mid;
    while ( low <= high )
    {
        mid = (high+low) >> 1;           // divide by two
        if ( key < products[mid] )       high = mid - 1;
        else if ( key > products[mid] ) low = mid + 1;
        else                             return( mid );
    }
    fprintf( stderr, "ERROR:  no match found; key = %d\n", key );
    return( -1 );
}
```

Listing 7: função de binary search

```
short eval_5hand( int *hand )
{
    int c1, c2, c3, c4, c5, q;
    short s;

    c1 = *hand++;
    c2 = *hand++;
    c3 = *hand++;
    c4 = *hand++;
    c5 = *hand;
    q = (c1|c2|c3|c4|c5) >> 16;

    // check for Flushes and StraightFlushes
    if ( c1 & c2 & c3 & c4 & c5 & 0xF000 )      return( flushes[q] );

    // check for Straights and HighCard hands
    s = unique5[q];
    if ( s )  return ( s );

    // let's do it the hard way
    q = (c1&0xFF) * (c2&0xFF) * (c3&0xFF) * (c4&0xFF) * (c5&0xFF);
    q = findit( q );

    return( values[q] );
}
```

Listing 8: função principal

Por possuir lookup tables, dizemos que o Cactus Kev possui tempo constante para as consultas em *flushes* e *uniques* (sequência ou carta alta).

Entretanto, se formos analisar as cartas que não se enquadram nesses quesitos, sua execução passa a ser de tempo logarítmico devido ao binary search empregado nessa condição.

Como a grande maioria das mãos de pôquer nãos se enquadram em *flushes* ou *uniques*, podemos dizer então que a complexidade do algoritmo é

- pior caso : $O(logn)$
- caso médio : $O(logn)$
- melhor caso: $O(1)$

para cada mão de pôquer.

Em nossos testes com o código classificando todas as 2.598.960 mãos possíveis, a operação foi realizada em um tempo de execução de 187 milissegundos.

3ª Abordagem : Cactus Kev com Perfect Hashing

Mesmo que o Cactus Kev seja bastante rápido para classificar uma mão de 5 cartas, ao analisar o código percebemos que a etapa que mais consome ciclos de clock é justamente o caso em que não temos flush, sequência ou carta alta, onde realizamos uma busca binária pelos valores das mãos não classificadas em um array.

Apesar da busca binária ser relativamente rápida, com complexidade O(log n), percebe-se que o evaluator pode ser melhorado com uma técnica parecida com a lookup table, tornando esta última etapa O(k), uma busca em tempo constante.

Esta técnica é o hashing, e como a tabela com que estamos trabalhando possui um tamanho fixo é possível utilizarmos uma variante ainda mais sofisticada, o perfect hashing, onde garantimos que não ocorrem colisões.

A solução apresentada neste trabalho foi desenvolvida por Paul Senzee [3] e praticamente sempre que alguém se refere ao evaluator Cactus Kev estão se referindo a versão com perfect hashing.

Ao substituir a busca binária por uma lista pré-computada de perfect hash com 4888 valores, Senzee relatou uma melhora no tempo de execução de 2.7 vezes.

A única alteração no código do Cactus Kev foi a substituição da função eval_5hand por eval_5hand_fast.

Código

```
unsigned find_fast(unsigned u)
{
    unsigned a, b, r;
    u += 0xe91aaa35;
    u ^= u >> 16;
    u += u << 8;
    u ^= u >> 4;
    b  = (u >> 8) & 0x1ff;
    a  = (u + (u << 2)) >> 19;
    r  = a ^ hash_adjust[b];
    return r;
}
```

Listing 9: função para consultar tabela de hash

```
int eval_5hand_fast(int c1, int c2, int c3, int c4, int c5)
{
    int q = (c1 | c2 | c3 | c4 | c5) >> 16;
    short s;
    if (c1 & c2 & c3 & c4 & c5 & 0xf000) return flushes[q]; // check for flushes and
    straight flushes
    if ((s = unique5[q]))                return s;          // check for straights and
    high card hands
    return hash_values[find_fast((c1 & 0xff) * (c2 & 0xff) * (c3 & 0xff) * (c4 & 0xff) *
    (c5 & 0xff))];
}
```

Listing 10: função principal

O cabeçalho, declarações e as tabelas não são considerados no cálculo da análise de complexidade, porém as tabelas são grandes o suficiente para não serem desprezadas em uma análise de complexidade espacial. Aliás, devido a grande quantidade de elementos, foi necessário omiti-las neste trabalho.

Uma observação pertinente: As tabelas foram computadas previamente e o código para executar esta etapa não está disponível para o público.

Em relação a função de consulta a tabela de hash, esta possui apenas manipulações de bits e operações básicas de lógica e aritmética, portanto podemos afirmar que possui complexidade $\theta(1)$.

E em relação a função principal, esta possui manipulações de bits, operações básicas de lógica e aritmética, e busca em array, portanto podemos, também, afirmar que possui complexidade $\theta(1)$.

Podemos dizer então que a complexidade do algoritmo é

- pior caso : $O(1)$
- caso médio : $O(1)$
- melhor caso: $O(1)$

para cada mão de pôquer.

Logo, concluímos que a função eval_5hand_fast é executada em tempo constante, o que, em nossos testes com código para classificação de todas as 2.598.960 mãos de pôquer, resulta em uma operação com tempo de execução de 75 milissegundos, uma melhora de 112 milissegundos para o Cactus Kev original que é equivalente a um processamento 2.5 vezes mais rápido (muito próximo dos 2.7 relatados por Senzee).

Bibliografia

1. Kevin Suffecool: *Cactus Kev's Poker Hand Evaluator*. http://suffe.cool/poker/evaluator.html.

2. Coding the wheel: *The Great Poker Hand Evaluator Roundup, Naive Hand Evaluators*. http://www.codingthewheel.com/archives/poker-hand-evaluator-roundup#xpokereval. Acessado via web archive : http://web.archive.org/web/20140625212722/

3. Paul Senzee: *Some Perfect Hash*. http://www.paulsenzee.com/2006/06/some-perfect-hash.html.