

# 数组

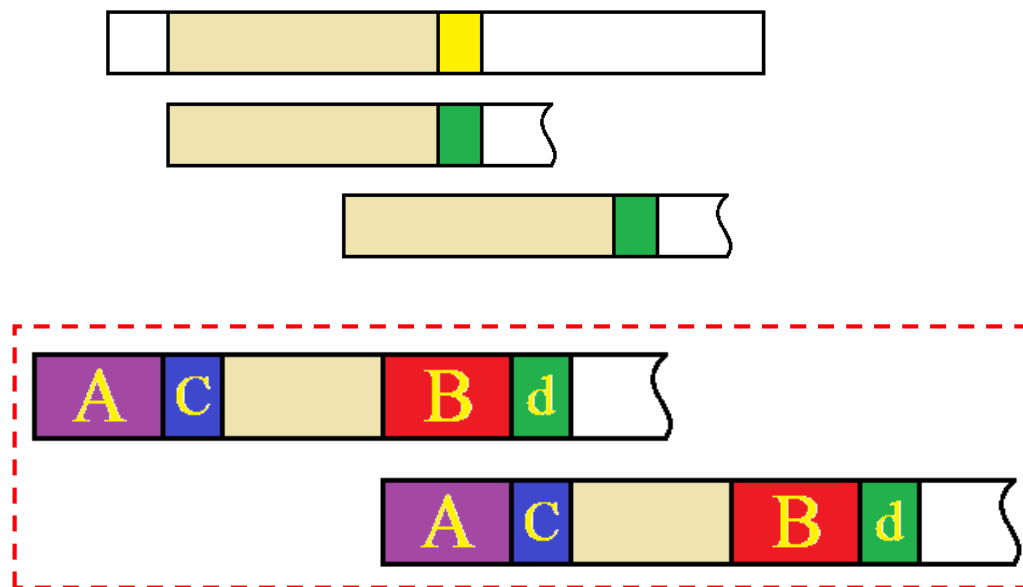
---

七月算法 邹博

2015年4月14日

# 历史遗留问题：KMP

- 问：KMP里有一个观点没引起注意，就是为什么pattern回溯，而text串可以不用？

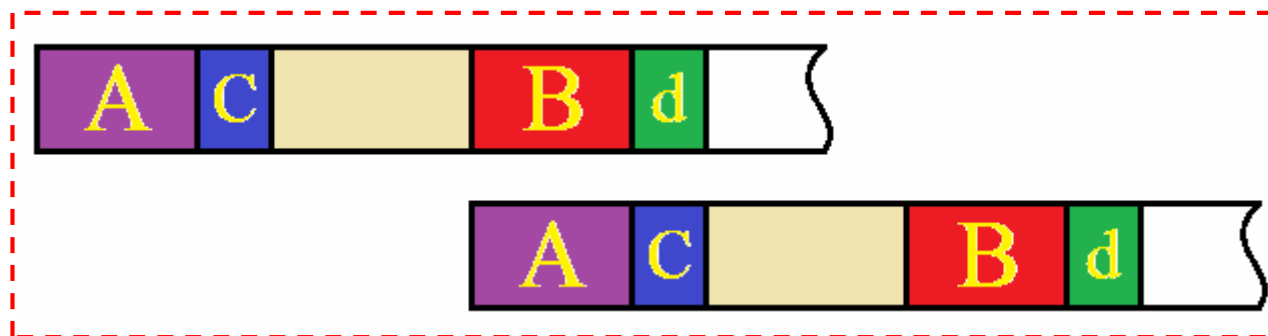
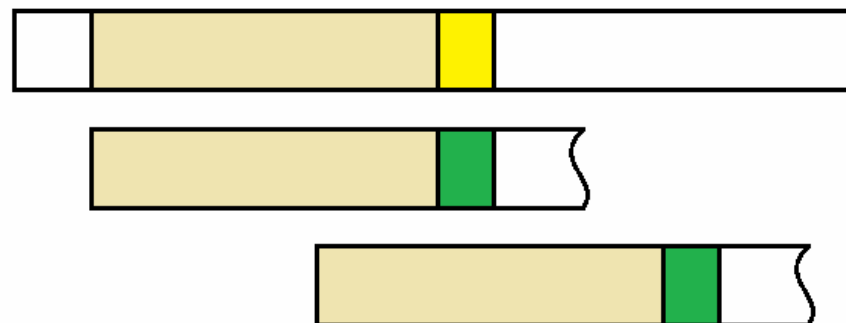


## 遗留问题：Manacher数组为何不考虑奇数？

- 字符串12212321 →  $S[] = "\$ \# 1 \# 2 \# 2 \# 1 \# 2 \# 3 \# 2 \# 1 \# "$ ;
  - trick: 为处理统一, 最前面加一位未出现的字符, 如\$
- 用一个数组 $P[i]$ 来记录以字符 $S[i]$ 为中心的最长回文子串向左/右扩张的长度(包括 $S[i]$ ), 比如S和P的对应关系:
  - S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
  - P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1
  - $P[i]-1$ 正好是原字符串中回文串的总长度
    - 若 $P[i]$ 为偶数, 考察 $x=P[i]/2$ 、 $2*x-1$
    - 思考: 若 $P[i]$ 为奇数呢?
      - 答: 不考虑! (为何?)



# 挖掘字符串比较的机制



# 数组

---

- 仅指存储方式为数组，**非**考察语言级数组的问题
  - `int a[20];int*p=a;sizeof(p)=?`
- 有些题目非常难，甚至是NP难解的
  - 后面将有NP的实例
- 往往有很难的面试压轴大题
  - **完美洗牌算法**
- 本次目标：
  - 除了学会**算法本身**，在这个过程中，探讨算法如何**设计**？理清**从特殊到一般**的分析思路。



# 寻找和为定值的两个数

---

- 输入一个数组  $A[0 \dots N-1]$  和一个数字  $\text{Sum}$ ，在数组中查找两个数  $A_i, A_j$ ，使得  $A_i + A_j = \text{Sum}$ 。



# 暴力求解

---

- 从数组中任意选取两个数 $x, y$ ，判定它们的和是否为输入的数字 $Sum$ 。时间复杂度为 $O(N^2)$ ，空间复杂度 $O(1)$ 。



# 稍好一点的方法

## □ 两头扫

- 如果数组是无序的，先排序 $O(N\log N)$ ，然后用两个指针 $i, j$ ，各自指向数组的首尾两端，令 $i=0, j=n-1$ ，然后 $i++$ ， $j--$ ，逐次判断 $a[i]+a[j]$ 是否等于 $Sum$ ：
- 若 $a[i]+a[j]>sum$ ，则 $i$ 不变， $j--$ ；
- 若 $a[i]+a[j]<sum$ ，则 $i++$ ， $j$ 不变；
- 若 $a[i]+a[j]==sum$ ，如果只要求输出一个结果，则退出；否则，输出结果后 $i++$ ， $j--$ ；

- 数组无序的时候，时间复杂度最终为 $O(N\log N+N)=O(N\log N)$ 。





# Code

```
void TwoSum(int data[], unsigned int length, int sum)
{
    //sort(s, s+n); 如果数组非有序的, 那就事先排好序O(N Log N)
    int begin = 0;
    int end = length - 1;
    //两端扫描法, 很经典的方法, O(N)
    while (begin < end)
    {
        long currSum = data[begin] + data[end];

        if (currSum == sum)
        {
            printf("%d %d\n", data[begin], data[end]);
            //如果需要所有满足条件的数组对, 则需要加上下面两条语句:
            //begin++
            //end--
            break;
        }
        else{
            if (currSum < sum)
                begin++;
            else
                end--;
        }
    }
}
```



10	15	8	64	76	72	84	...	...	...
----	----	---	----	----	----	----	-----	-----	-----

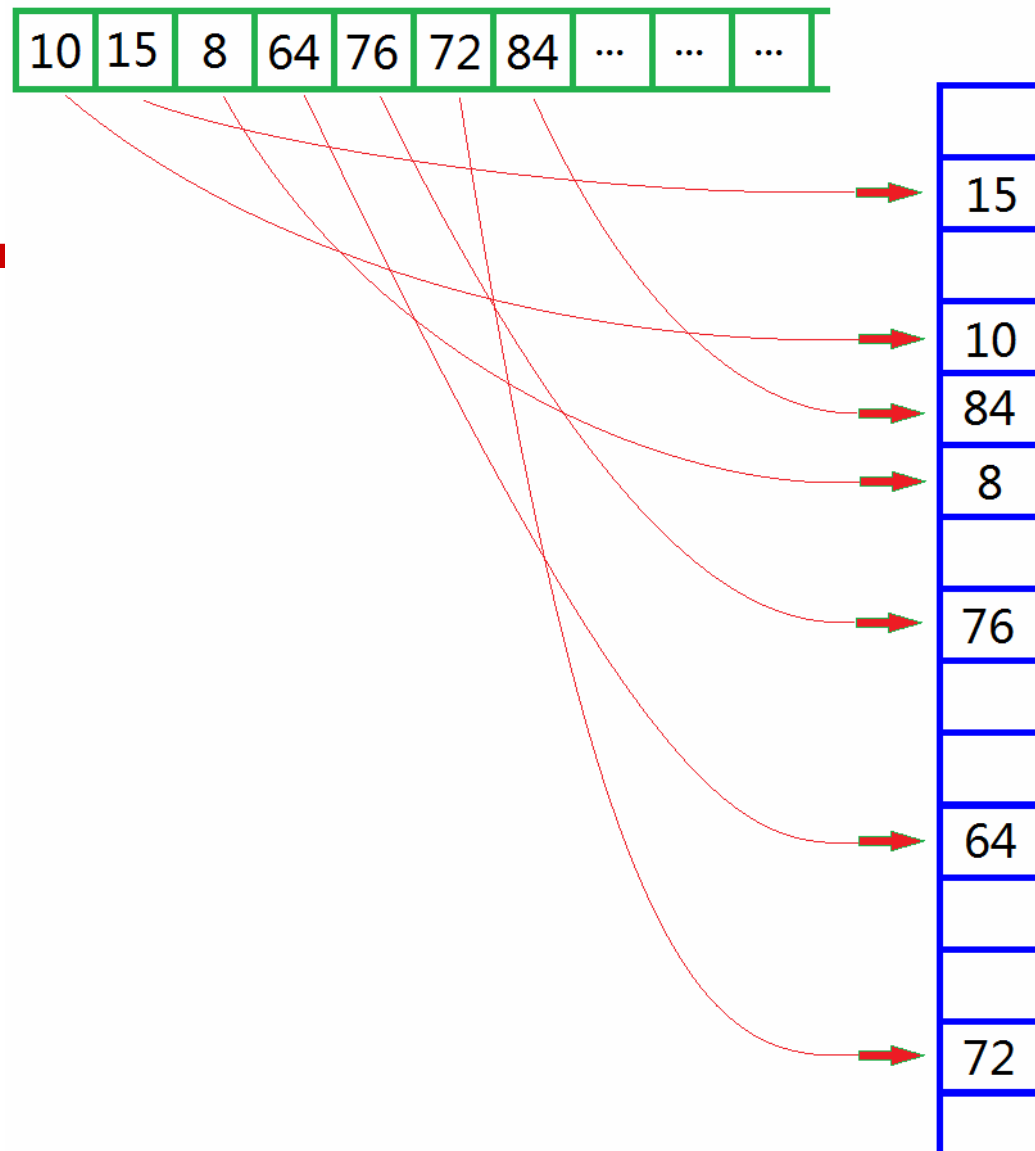
## 引：Hash结构

- 事先开辟一个足够大的空间H。以数组元素值x为自变量，通过某函数f，将其映射为某个整数值index，将该整数值x存储在H[index]处，这样，在以后查找元素a是否存在时，只需要计算f(a)，得到一个值y，看H[y]处是否存在a即可。
- 显然，它是通过足够大的存储空间，保证了O(1)的时间找到某元素。(如果把函数值的计算认为是常数级的)
- 以下简要介绍Hash结构，详细内容将在下次“Hash和树”等章节中再次叙述。

15
10
84
8
76
64
72



# Hash示例



# Hash函数

---

□ Hash函数在Hash结构设计中居于核心地位。  
一个好的Hash函数，能够将数据映射到“杂乱”的位置。

■ dbj2

■ sdbm

■ MurmurHash



# djb2

---

- this algorithm ( $k=33$ ) was first reported by dan bernstein many years ago in comp.lang.c. another version of this algorithm (now favored by bernstein) uses xor:  $\text{hash}(i) = \text{hash}(i - 1) * 33 \oplus \text{str}[i]$ ; the magic of number 33 (why it works better than many other constants, prime or not) has never been adequately explained.
- 这个算法( $k=33$ )是多年前首先由dan bernstein 在 comp.lang.c中提出的, 这个算法的另外一个版本(bernstein的贡献)是使用异或方式:  $\text{hash}(i) = \text{hash}(i - 1) * 33 \oplus \text{str}[i]$ ;魔数33尚未得到足够的解释(为何它能够比其他很多素数的效果更好)



# djb2 Code

---

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```



# sdbm Code(65599)

---

```
static unsigned long  
sdbm(str)  
unsigned char *str;  
{  
    unsigned long hash = 0;  
    int c;  
  
    while (c = *str++)  
        hash = c + (hash << 6) + (hash << 16) - hash;  
  
    return hash;  
}
```



# sdbm的说明

---

- This algorithm was created for sdbm (a public-domain reimplementation of ndbm) database library. it was found to do well in scrambling bits, causing better distribution of the keys and fewer splits.
- It also happens to be a good general hashing function **with good distribution**. the actual function is  $\text{hash}(i) = \text{hash}(i - 1) * 65599 + \text{str}[i]$ ; which is the faster version used in **gawk**. The magic constant 65599 was picked out of thin air while experimenting with different constants, and turns out to be a prime. this is one of the algorithms used in berkeley db and elsewhere.





# MurmurHash

---

- ❑ MurmurHash是一种非加密型哈希函数，适用于一般的哈希检索操作。由Austin Appleby在2008年发明，并出现了多个变种，都已经发布到了公有领域(public domain)。与其它流行的哈希函数相比，对于规律性较强的key，Murmur Hash的随机分布特征表现更良好。
- ❑ 当前的版本是MurmurHash3，能够产生出32-bit或128-bit哈希值。
- ❑ 较早的MurmurHash2能产生32-bit或64-bit哈希值。对于大端存储和强制对齐的硬件环境有一个较慢的MurmurHash2可以用。MurmurHash2A变种增加了Merkle–Damgård构造，所以能够以增量方式调用。有两个变种产生64-bit哈希值：MurmurHash64A，为64位处理器做了优化；MurmurHash64B，为32位处理器做了优化。MurmurHash2-160用于产生160-bit哈希值，而MurmurHash1已经不再使用。



# MurmurHash

```
Murmur3_32(key, len, seed)
  c1 ← 0xcc9e2d51
  c2 ← 0x1b873593
  r1 ← 15
  r2 ← 13
  m ← 5
  n ← 0xe6546b64

  hash ← seed

  for each fourByteChunk of key
    k ← fourByteChunk

    k ← k * c1
    k ← (k << r1) OR (k >> (32-r1))
    k ← k * c2

    hash ← hash XOR k
    hash ← (hash << r2) OR (hash >> (32-r2))
    hash ← hash * m + n

  with any remainingBytesInKey
    remainingBytes ← SwapEndianOrderOf(remainingBytesInKey)
    remainingBytes ← remainingBytes * c1
    remainingBytes ← (remainingBytes << r1) OR (remainingBytes >> (32 - r1))
    remainingBytes ← remainingBytes * c2

    hash ← hash XOR remainingBytes

  hash ← hash XOR len

  hash ← hash XOR (hash >> 16)
  hash ← hash * 0x85ebca6b
  hash ← hash XOR (hash >> 13)
  hash ← hash * 0xc2b2ae35
  hash ← hash XOR (hash >> 16)
```



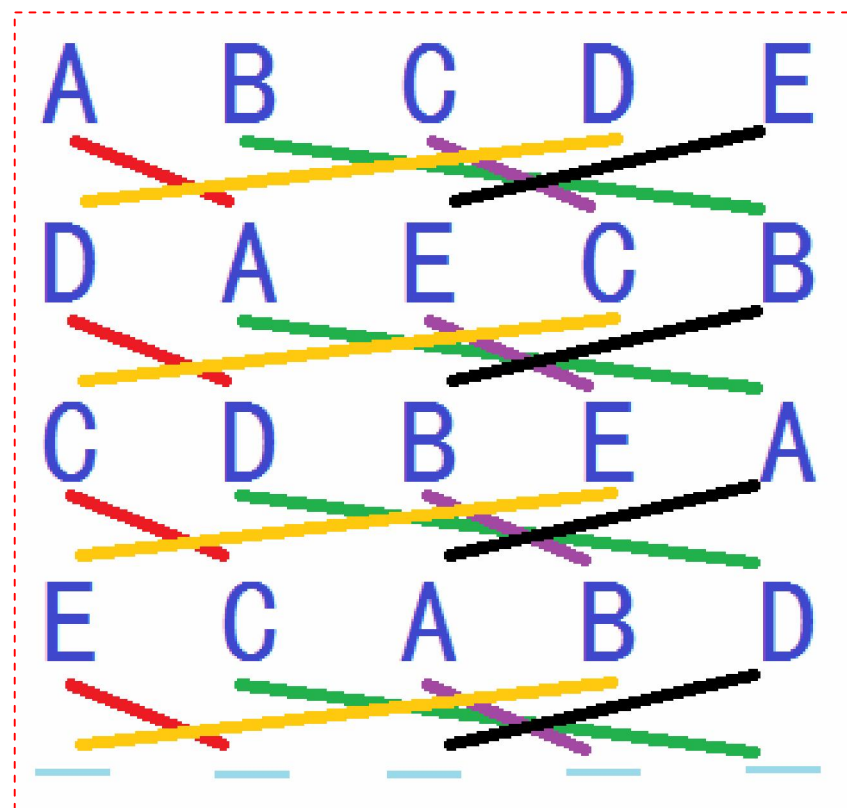
# 玩乐中带来的算法思维——完美Hash

哈佛大学智商测试

请补填上第4行字母。

A	B	C	D	E
D	A	E	C	B
C	D	B	E	A
—	—	—	—	—

A B C D E



# Hash冲突

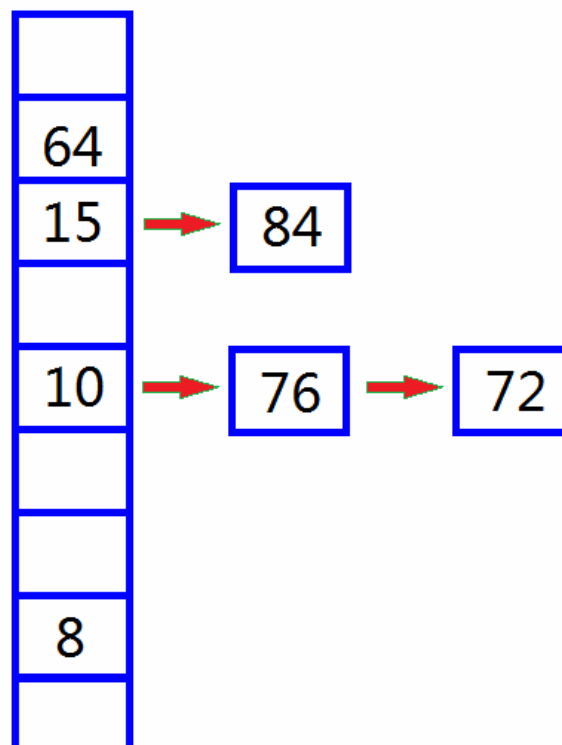
---

- 当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突（collision）。



# Hash冲突

10	15	8	64	76	72	84	...	...	...
----	----	---	----	----	----	----	-----	-----	-----



# 讨论：Hash方案的可行性

---

## □ 算法步骤

- 选择适当的Hash函数，对原数组建立Hash结构
- 遍历数组 $a[i]$ ，计算 $\text{Hash}(\text{Sum}-a[i])$ 是否存在

## □ 算法可行性

- 时间复杂度，空间复杂度



# 和为定值的m个数

---

- 已知数组  $A[0 \dots N-1]$ , 给定某数值  $sum$ , 找出数组中的若干个数, 使得这些数的和为  $sum$ 。
- 布尔向量  $x[0 \dots N-1]$ 
  - $x[i]=0$  表示不取  $A[i]$ ,  $x[i]=1$  表示取  $A[i]$
  - 假定数组中的元素都大于0:  $A[i] > 0$
  - 这是个NP问题!



# 分析方法

---

- ☐ 直接递归法（枚举）
- ☐ 分支限界
- ☐ 存在负数的处理办法





# 直接递归法

1:	1	2	3	4
2:	1	4	5	
3:	2	3	5	

```
int a[] = {1, 2, 3, 4, 5};  
int size = sizeof(a) / sizeof(int);  
int sum = 10;    //sum为计算的和
```

//x[]为最终解, i为考察第x[i]是否加入, has表示当前的和

```
void EnumNumber(bool* x, int i, int has)
```

```
{  
    if(i >= size)  
        return;  
    if(has + a[i] == sum)  
    {  
        x[i] = true;  
        Print(x);  
        x[i] = false;  
    }  
    x[i] = true;  
    EnumNumber(x, i+1, has+a[i]);  
    x[i] = false;  
    EnumNumber(x, i+1, has);  
}
```

```
int _tmain(int argc, _TCHAR* argv[])  
{  
    bool* x = new bool[size];  
    memset(x, 0, size);  
    EnumNumber(x, 0, 0);  
    delete[] x;  
    return 0;  
}
```



# 考虑对于分支如何限界

- 前提：数组  $A[0 \dots N-1]$  的元素都大于0
- 考察向量  $x[0 \dots N-1]$ ，假定已经确定了前  $i$  个值，现在要判定第  $i+1$  个值  $x[i]$  为0还是1。
- 假定由  $x[0 \dots i-1]$  确定的  $A[0 \dots i-1]$  的和为  $has$ ；
- $A[i, i+1, \dots N-1]$  的和为  $residue$ （简记为  $r$ ）；
  - $has + a[i] \leq sum$  并且  $has + r \geq sum$ ：  $x[i]$  可以为1；
  - $has + (r - a[i]) \geq sum$ ：  $x[i]$  可以为0；
- 注意，这里是“可以”——可以能够：可能。



# 分支限界法

```
1:  1  2  3  4  5  6  9 10
2:  1  2  3  4  5  7  8 10
3:  1  2  3  4  6  7  8  9
4:  1  2  3  7  8  9 10
5:  1  2  4  6  8  9 10
6:  1  2  5  6  7  9 10
7:  1  3  4  5  8  9 10
8:  1  3  4  6  7  9 10
9:  1  3  5  6  7  8 10
10: 1  4  5  6  7  8  9
11: 1  5  7  8  9 10
12: 2  3  4  5  7  9 10
13: 2  3  4  6  7  8 10
14: 2  3  5  6  7  8  9
15: 2  4  7  8  9 10
16: 2  5  6  8  9 10
17: 3  4  6  8  9 10
18: 3  5  6  7  9 10
19: 4  5  6  7  8 10
20: 6  7  8  9 10
```

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int size = sizeof(a) / sizeof(int);
int sum = 40; //sum为计算的和
```

//x[]为最终解, i为考察第x[i]是否加入, has表示当前的和  
//residue是剩余数的全部和

```
void FindNumber(bool* x, int i, int has, int residue)
{
    if(i >= size)
        return;
    if(has + a[i] == sum)
    {
        x[i] = true;
        Print(x);
        x[i] = false;
    }
    else if((has + residue >= sum) && (has + a[i] <= sum))
    {
        x[i] = true;
        FindNumber(x, i+1, has+a[i], residue-a[i]);
    }
    if(has + residue - a[i] >= sum)
    {
        x[i] = false;
        FindNumber(x, i+1, has, residue-a[i]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int residue = Sum(a, size);
    bool* x = new bool[size];
    memset(x, 0, size);
    FindNumber(x, 0, 0, residue);
    delete[] x;
    return 0;
}
```



# 数理逻辑的重要应用：分支限界的条件

---

- 分支限界的条件是充分条件吗？
- 在新题目中，如何发现分支限界的条件。

■ 学会该方法，比此问题本身更重要



# 考虑负数的情况

- 枚举法肯定能得到正确的解
- 如何对负数进行分支限界？
  - 可对整个数组 $A[0 \dots N-1]$ 正负排序，使得负数都在前面，正数都在后面，使用剩余正数的和作为分支限界的约束：
  - 如果 $A[i]$ 为负数：如果全部正数都算上还不够，就不能选 $A[i]$ ；
  - 如果递归进入了正数范围，按照数组是全正数的情况正常处理；
  - 注：正负排序马上就要讲到



# 带负数的分支限界

A = {-3, -5, -2, 4, 2, 1, 3}  
sum = 5

1:	-3	-2	4	2	1	3
2:	-3	4	1	3		
3:	-5	4	2	1	3	
4:	-2	4	2	1		
5:	-2	4	3			
6:	4	1				
7:	2	3				

```
int _tmain(int argc, _TCHAR* argv[])
{
    int positive, negative;
    Sum(a, size, negative, positive);
    bool* x = new bool[size];
    memset(x, 0, size);
    FindNumber2(x, 0, 0, negative, positive);
    delete[] x;
    return 0;
}
```

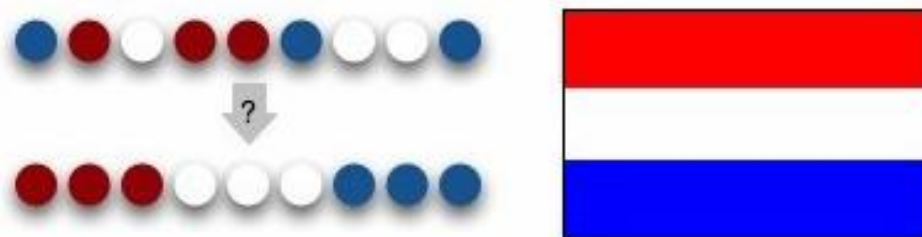
```
//residue剩余的所有正数的和
void FindNumber2(bool* x, int i, int has, int negative, int positive)
{
    if(i >= size)
        return;
    if(has + a[i] == sum)
    {
        x[i] = true;
        Print(x);
        x[i] = false;
    }

    if(a[i] >= 0)
    {
        if((has + positive >= sum) && (has + a[i] <= sum))
        {
            x[i] = true;
            FindNumber2(x, i+1, has+a[i], negative, positive-a[i]);
            x[i] = false;
        }
        if(has + positive - a[i] >= sum)
        {
            x[i] = false;
            FindNumber2(x, i+1, has, negative, positive-a[i]);
        }
    }
    else
    {
        if(has + x[i] + positive >= sum)
        {
            x[i] = true;
            FindNumber2(x, i+1, has+a[i], negative-a[i], positive);
            x[i] = false;
        }
        if((has + negative <= sum) && (has + positive >= sum))
        {
            x[i] = false;
            FindNumber2(x, i+1, has, negative-a[i], positive);
        }
    }
}
```



# 荷兰国旗问题

- 现有红、白、蓝三个不同颜色的小球，乱序排列在一起，请重新排列这些小球，使得红白蓝三色的同颜色的球在一起。这个问题之所以叫荷兰国旗，是因为我们可以将红白蓝三色小球想象成条状物，有序排列后正好组成荷兰国旗。



# 问题分析

- 问题转换为：给定数组 $A[0 \dots N-1]$ ，元素只能取0、1、2三个值，设计算法，使得数组排列成“00...0011...1122...22”的形式。
- 借鉴快速排序中partition的过程。定义三个指针： $begin=0$ 、 $current=0$ 、 $end=N-1$ ；
- $A[cur]==2$ ，则 $A[cur]$ 与 $A[end]$ 交换， $end--$ ， $cur$ 不变
- $A[cur]==1$ ，则 $cur++$ ， $begin$ 不变， $end$ 不变
- $A[cur]==0$ ，则：
  - 若 $begin==cur$ ，则 $begin++$ ， $cur++$
  - 若 $begin \neq cur$ ，则 $A[cur]$ 与 $A[begin]$ 交换， $begin++$ ， $cur$ 不变





# Code1

- $A[cur] == 2$ , 则  $A[cur]$  与  $A[end]$  交换,  $end--$ ,  $cur$  不变
- $A[cur] == 1$ , 则  $cur++$ ,  $begin$  不变,  $end$  不变
- $A[cur] == 0$ , 则:
  - 若  $begin == cur$ , 则  $begin++$ ,  $cur++$
  - 若  $begin \neq cur$ , 则  $A[cur]$  与  $A[begin]$  交换,  $begin++$ ,  $cur$  不变

```
void Holland(int* a, int length)
{
    int begin = 0;
    int current = 0;
    int end = length - 1;
    while(current <= end)
    {
        if(a[current] == 2)
        {
            swap(a[end], a[current]);
            end--;
        }
        else if(a[current] == 1)
        {
            current++;
        }
        else // if(a[current] == 0)
        {
            if(begin == current)
            {
                begin++;
                current++;
            }
            else
            {
                swap(a[current], a[begin]);
                begin++;
            }
        }
    }
}
```

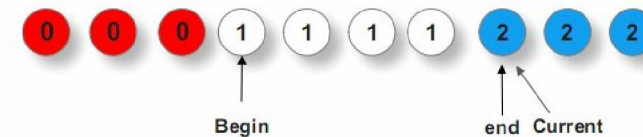
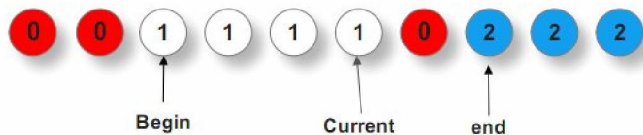
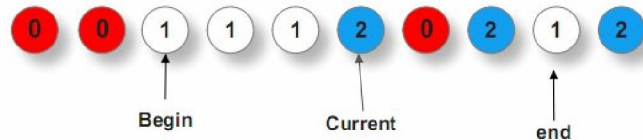
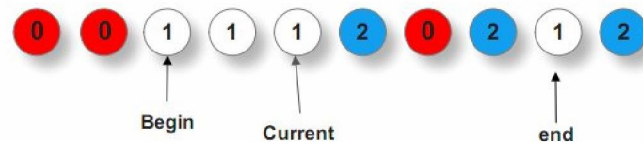
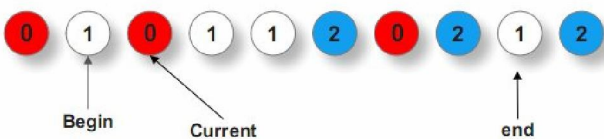
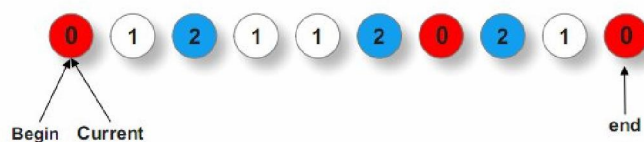


# 进一步的考虑：略做优化

- current扫过的位置，即： $[begin, cur)$ 区间内，一定没有2
  - 在前面的 $A[cur]==2$ 中，已经被替换到数组后面了
- 因此： $A[begin]$ 要么是0，要么是1，不可能是2
- 考察begin指向的元素的值：
- 若 $begin \neq cur$ ，则必有 $A[begin]=1$ 
  - 归纳法
- 因此，当 $A[cur]==0$ 时，
  - 若 $begin \neq cur$ ，因为 $A[begin]==1$ ，则交换后， $A[cur]==1$ ，此时，可以 $cur++$ ；



# 优化后的图示演示



# Code2

- $A[cur] == 2$ , 则  $A[cur]$  与  $A[end]$  交换,  $end--$ ,  $cur$  不变
- $A[cur] == 1$ , 则  $cur++$ ,  $begin$  不变,  $end$  不变
- $A[cur] == 0$ , 则:
  - 若  $begin == cur$ , 则  $begin++$ ,  $cur++$
  - 若  $begin \neq cur$ , 则  $A[cur]$  与  $A[begin]$  交换,  $begin++$ ,  $cur++$

```
void Holland(int* a, int length)
{
    int begin = 0;
    int current = 0;
    int end = length - 1;
    while(current <= end)
    {
        if(a[current] == 2)
        {
            swap(a[end], a[current]);
            end--;
        }
        else if(a[current] == 1)
        {
            current++;
        }
        else // if(a[current] == 0)
        {
            if(begin == current)
            {
                begin++;
                current++;
            }
            else
            {
                swap(a[current], a[begin]);
                begin++;
                current++;
            }
        }
    }
}
```



# Code2'

```
void Holland(int* a, int length)
{
    int begin = 0;
    int current = 0;
    int end = length - 1;
    while(current <= end)
    {
        if(a[current] == 2)
        {
            swap(a[end], a[current]);
            end--;
        }
        else if(a[current] == 1)
        {
            current++;
        }
        else// if(a[current] == 0)
        {
            //1、或者用更直接的判断if(a[current] != a[begin]);
            //2、因为不等的次数远远大于相等的次数，可以直接删去该判断
            if(current != begin)
                swap(a[current], a[begin]);
            begin++;
            current++;
        }
    }
}
```



# 荷兰国旗问题带来的思考

---

- 在  $\text{begin}/\text{cur}/\text{end}$  的循环中， $\text{cur}$  遇到 0 和遇到 2， $\text{begin}$  和  $\text{end}$  的处理方式不对称
  - 遇到 0:  $\text{begin}++$ ,  $\text{cur}++$
  - 遇到 2:  $\text{end}--$ ,  $\text{cur}$  不变
  - 网络版本的荷兰国旗算法，往往直接给出上述结论
- 若初值给定如下：
  - $\text{begin}=0$ 、 $\text{current}=\text{N}-1$ 、 $\text{end}=\text{N}-1$ ，如何完成代码？



# Code3

---

```
void Holland(int* a, int length)
{
    int begin = 0;
    int end = length - 1;
    int current = end;
    while(current >= begin)
    {
        if(a[current] == 2)
        {
            swap(a[end], a[current]);
            end--;
            current--;
        }
        else if(a[current] == 1)
        {
            current--;
        }
        else// if(a[current] == 0)
        {
            swap(a[current], a[begin]);
            begin++;
        }
    }
}
```



# “乌克兰国旗”问题

---

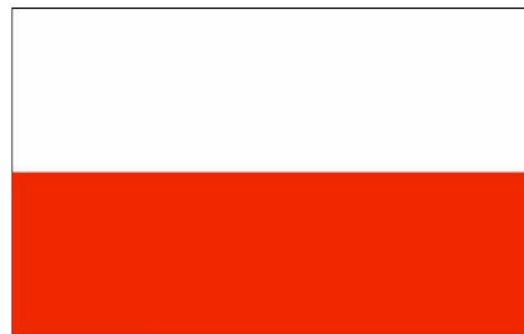
□ 如果是分成两部分呢？

■ 给定整数数组，要求奇数在前，偶数在后

□ 奇偶排序

■ 给定实数数组，要求负数在前，正数在后

□ 正负排序





# 讨论：荷兰国旗问题的其他方案

---

- 荷兰国旗中，0，1，2分别计数，然后根据三个计数值，赋值数组；
  - 是否可行
- 将 (0,1) (2) 根据快速排序的Partition，分成两堆（不妨使用1.5或者其他数作为PivotKey），然后，将 (0) (1) 根据快速排序的Partition，分成两堆（不妨使用0.5或者其他数作为PivotKey）。



# 完美洗牌算法

---

□ 长度为 $2n$ 的数组

$\{a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n\}$ ，经过整理后变成 $\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ ，要求时间复杂度 $O(n)$ ，空间复杂度 $O(1)$ 。



# 步步前移

- 观察变换前后两个序列的特点，我们可做如下一系列操作：
- 第①步确定b1的位置，即让b1跟它前面的a2, a3, a4交换：
  - a1, b1, a2, a3, a4, b2, b3, b4
- 第②步、接着确定b2的位置，即让b2跟它前面的a3, a4交换：
  - a1, b1, a2, b2, a3, a4, b3, b4
- 第③步、b3跟它前面的a4交换位置：
  - a1, b1, a2, b2, a3, b3, a4, b4
- b4已在最后的位置，不需要再交换。如此，经过上述3个步骤后，得到我们最后想要的序列。
- 移动n-1次，第i次将n-i个元素后移。时间复杂度为 $O(N^2)$ 。



# 中间交换

---

- 每次让序列中最中间的元素进行交换。
- 对于  $a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4$
- 第①步：交换最中间的两个元素  $a_4, b_1$ ，序列变成：
  - $a_1, a_2, a_3, b_1, a_4, b_2, b_3, b_4$
- 第②步，让最中间的两对元素各自交换：
  - $a_1, a_2, b_1, a_3, b_2, a_4, b_3, b_4$
- 第③步，交换最中间的三对元素，序列变成：
  - $a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4$



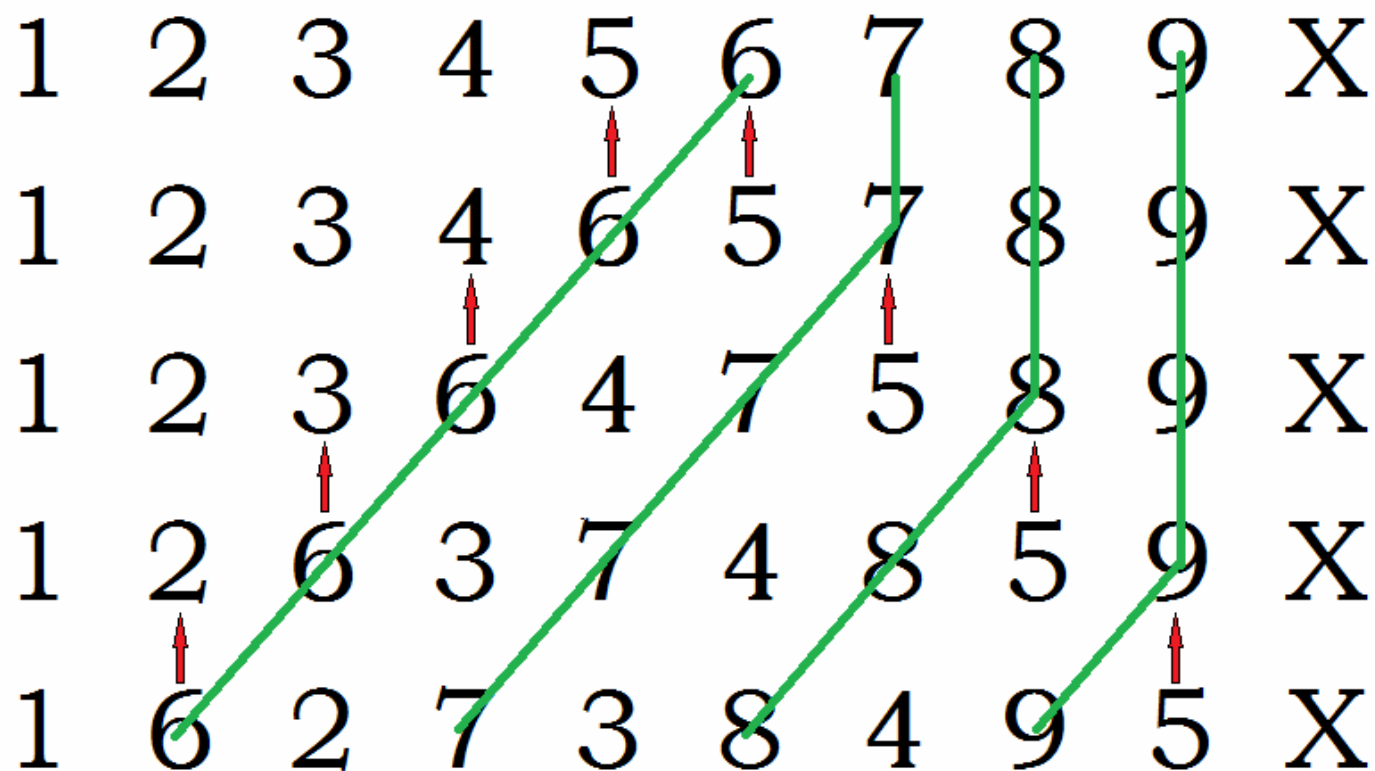
# 中间交换

---

1	2	3	4	5	6	7	8	9	X
				↑	↑				
1	2	3	4	6	5	7	8	9	X
			↑			↑			
1	2	3	6	4	7	5	8	9	X
		↑					↑		
1	2	6	3	7	4	8	5	9	X
	↑							↑	
1	6	2	7	3	8	4	9	5	X



# 中间交换



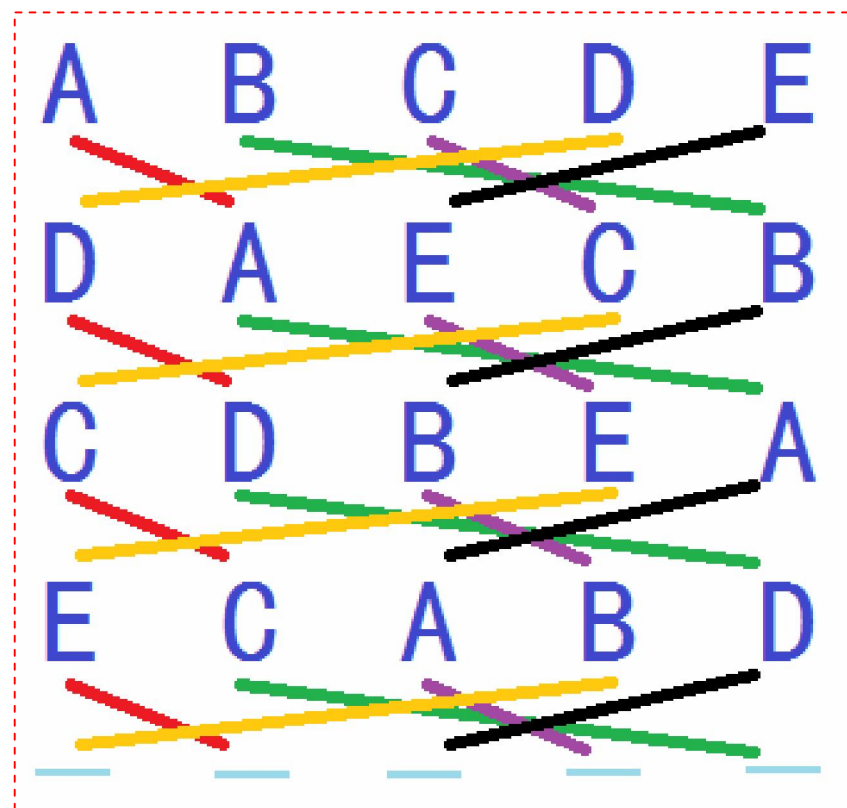
# 玩乐中带来的算法思维——再次遇到

哈佛大学智商测试

请补填上第4行字母。

A	B	C	D	E
D	A	E	C	B
C	D	B	E	A
—	—	—	—	—

A B C D E



# 完美洗牌算法

---

- 2004年，microsoft的Peiyush Jain在他发表一篇名为：“A Simple In-Place Algorithm for In-Shuffle”的论文中提出了完美洗牌算法。





# 位置变换

- $a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n \rightarrow b_1, a_1, b_2, a_2, b_3, a_3, \dots, b_n, a_n$
- 设定数组的下标范围是 $[1..2n]$ 。考察元素的最终位置：
- 以 $n=4$ 为例，前 $n$ 个元素中，
  - 第1个元素 $a_1$ 到了原第2个元素 $a_2$ 的位置，即 $1 \rightarrow 2$ ；
  - 第2个元素 $a_2$ 到了原第4个元素 $a_4$ 的位置，即 $2 \rightarrow 4$ ；
  - 第3个元素 $a_3$ 到了原第6个元素 $b_2$ 的位置，即 $3 \rightarrow 6$ ；
  - 第4个元素 $a_4$ 到了原第8个元素 $b_4$ 的位置，即 $4 \rightarrow 8$ ；
- 前 $n$ 个元素中，第 $i$ 个元素的最终位置为 $(2 * i)$ 。
- 后 $n$ 个元素，可以看出：
  - 第5个元素 $b_1$ 到了原第1个元素 $a_1$ 的位置，即 $5 \rightarrow 1$ ；
  - 第6个元素 $b_2$ 到了原第3个元素 $a_3$ 的位置，即 $6 \rightarrow 3$ ；
  - 第7个元素 $b_3$ 到了原第5个元素 $b_1$ 的位置，即 $7 \rightarrow 5$ ；
  - 第8个元素 $b_4$ 到了原第7个元素 $b_3$ 的位置，即 $8 \rightarrow 7$ ；
- 后 $n$ 个元素，第 $i$ 个元素的最终位置为： $(2 * (i - n)) - 1 = 2 * i - 2 * n - 1 = (2 * i) \% (2 * n + 1)$



# 两个圈

□ 我们得到两个圈

□  $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 1$

□  $3 \rightarrow 6 \rightarrow 3$

*//数组下标从1开始, from是圈的头部, mod为  $2 * n + 1$*

```
void CycleLeader(int *a, int from, int mod)
{
    int t,i;

    for(i = from * 2 % mod; i != from; i = i * 2 % mod)
    {
        t = a[i];
        a[i] = a[from];
        a[from] = t;
    }
}
```



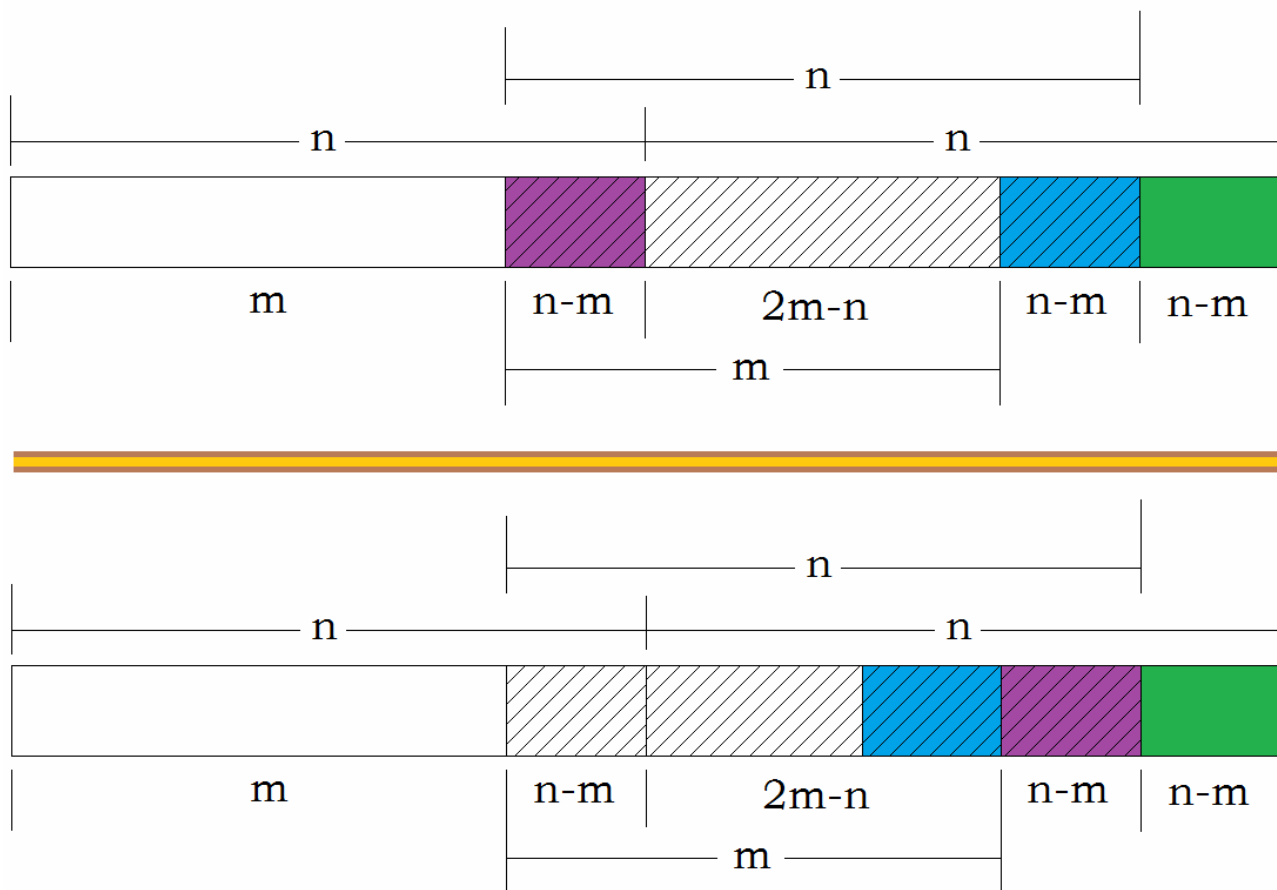
# K个圈

---

- 对于  $2^n = (3^k - 1)$  这种长度的数组，恰好只有  $k$  个圈，且每个圈的起始位置分别是  $1, 3, 9, \dots, 3^{k-1}$



若： $2m$ 可以写成 $3^k-1$ 的形式



# 任意长度数组的完美洗牌算法

---

**Input:** An array  $A[1, \dots, 2n]$

**Step 1.** Find a  $2m = 3^k - 1$  such that  $3^k \leq 2n < 3^{k+1}$

**Step 2.** Do a right cyclic shift of  $A[m + 1, \dots, n + m]$  by a distance  $m$

**Step 3.** For each  $i \in \{0, 1, \dots, k - 1\}$ , starting at  $3^i$ , do the cycle leader algorithm for the in-shuffle permutation of order  $2m$

**Step 4.** Recursively do the in-shuffle algorithm on  $A[2m + 1, \dots, 2n]$ .



# 循环移位

□  $(AB)' = B'A'$

```
// 翻转字符串时间复杂度O(to - from)
void reverse(int *a, int from, int to)
{
    int t;
    for (; from < to; ++from, --to)
    {
        t = a[from];
        a[from] = a[to];
        a[to] = t;
    }
}

// 循环右移num位 时间复杂度O(n)
void RightRotate(int *a, int num, int n)
{
    reverse(a, 1, n - num);
    reverse(a, n - num + 1, n);
    reverse(a, 1, n);
}
```



# 完美洗牌算法流程

---

- 输入数组  $A[1..2 * n]$
- step 1 找到  $2*m=3^k-1$ ，且  $3^k \leq 2*n < 3^{(k+1)}$
- step 2 把  $a[m+1..m+n]$  那部分循环右移  $m$  位
- step 3 对每个  $i = 0, 1, 2..k - 1$ ， $3^i$  是每个圈的起始位置，做 cycle\_leader 算法；
  - 注：因为子数组长度为  $m$ ，所以对  $2*m+1$  取模
- step 4 对数组的剩余部分  $A[2*m+1.. 2*n]$  继续使用本算法。



# 完美洗牌代码

```
void PerfectShuffle2(int *a, int n)
{
    int n2, m, i, k, t;
    for (; n > 1;)
    {
        // step 1
        n2 = n * 2;
        for (k = 0, m = 1; n2 / m >= 3; ++k, m *= 3)
            ;
        m /= 2;
        //  $2m = 3^k - 1$ ,  $3^k \leq 2n < 3^{k+1}$ 

        // step 2
        right_rotate(a + m, m, n);

        // step 3
        for (i = 0, t = 1; i < k; ++i, t *= 3)
        {
            cycle_leader(a, t, m * 2 + 1);
        }

        // step 4
        a += m * 2;
        n -= m;
    }
    // n = 1
    t = a[1];
    a[1] = a[2];
    a[2] = t;
}
```





# 依据

---

□ 2是3的原根, 2是9的原根

□  $\{2^0, 2^1\} = \{1, 2\}$

□  $\{2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8\} \bmod 9$

□  $= \{1, 2, 4, 8, 7, 5\}$

□ 而  $\phi(9) = 6$



# 附：算法原文

---

We show that, when  $2n$  is of the form  $3^k - 1$ , we can easily determine the cycles of the in-shuffle permutation of order  $2n$ . We will need the following theorem from number theory:

**Theorem 1** *If  $p$  is an odd prime and  $g$  is a primitive root of  $p^2$ , then  $g$  is a primitive root of  $p^k$  for any  $k \geq 1$ .*

A proof of this theorem can be found in [Nar00, p 20-21].

It can be easily seen that 2 is a primitive root of 9. From the above theorem it follows that 2 is also a primitive root of  $3^k$  for any  $k \geq 1$ . This implies that the group  $(\mathbb{Z}/3^k)^*$  is cyclic with 2 being its generator.

Now let us analyse the cycles of an in-shuffle permutation when  $2n = 3^k - 1$ .

The cycle containing 1 is nothing but the group  $(\mathbb{Z}/3^k)^*$ , which consists of all numbers relatively prime to  $3^k$  and less than it.

Let  $1 \leq s < k$ . Consider the cycle containing  $3^s$ . Every number in this cycle is of the form  $3^s 2^t \pmod{3^k}$  for  $1 \leq t \leq \varphi(3^k)$  (where  $\varphi$  is the Euler-totient function). Since 2 is a generator of  $(\mathbb{Z}/3^k)^*$ , this cycle contains *exactly* the numbers less than  $3^k$  which are divisible by  $3^s$  but not by any higher power of 3.

This means that in an in-shuffle permutation of order  $3^k - 1$ , we have exactly  $k$  cycles with  $1, 3, 3^2, \dots, 3^{k-1}$  each belonging to a different cycle. Thus for these permutations, it becomes easy to pick the 'next' cycle in order to apply the cycle leader algorithm. Note that the length of the cycle containing  $3^s$  is  $\varphi(3^k)/3^s$ , which helps us implement the cycle leader algorithm more efficiently.



# 进一步的思考

- 要求输出是 $a_1, b_1, a_2, b_2, \dots, a_n, b_n$ ，而完美洗牌算法输出是 $b_1, a_1, b_2, a_2, \dots, b_n, a_n$ ，怎么办？
  - 先把a部分和b部分交换，或者最后再交换相邻的两个位置——不够美观。
  - 原数组第一个和最后一个不变，中间的 $2*(n-1)$ 项用原始的完美洗牌算法。
- 逆完美洗牌问题：给定 $b_1, a_1, b_2, a_2, \dots, b_n, a_n$ ，要求输出 $a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n$ 。
  - 既然完美洗牌问题可以通过若干圈来解决，那么，逆完美洗牌问题仍然存在是若干圈，并且 $2*n = (3^k - 1)$ 这种长度的数组恰好只有k个圈的结论仍然成立。
- 完美洗多付牌：给定 $a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n, c_1, c_2, \dots, c_n$ ，要求输出是 $c_1, b_1, a_1, c_2, b_2, a_2, \dots, c_n, b_n, a_n$ 
  - 2付牌的结论：2是群 $(\mathbb{Z}/3^k)^*$ 最小生成元，且 $(3^k - 1)$ 这种长度的数组，恰好只有k个圈
  - 考察是否存在某数字p（如5、7、11、13等），使得数字3是群 $(\mathbb{Z}/p^k)^*$ 的最小生成元，再验证p是否存在结论 $(p^k - 1)$ 这种长度的数组，恰好只有k个圈。
  - 提示：3是7的原根，是49的原根，于是3是 $7^k$ 的原根



# 参考文献

---

- [http://blog.csdn.net/v\\_JULY\\_v/article/details/6419466](http://blog.csdn.net/v_JULY_v/article/details/6419466)(和为定值的N个数)
- <http://blog.csdn.net/jinyongqing/article/details/12054495> (和为定值的N个数)
- <https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.03.md> (和为定值的N个数)
- Peiyush Jain, A Simple In-Place Algorithm for In-Shuffle, Microsoft, July, 2004(完美洗牌)
- <https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md>(完美洗牌)
- <http://blog.csdn.net/caopengcs/article/details/10521603>(完美洗牌)
- <http://zh.wikipedia.org/wiki/%E6%AC%A7%E6%8B%89%E5%87%BD%E6%95%B0>(欧拉函数)
- <http://www.cnblogs.com/frog112111/archive/2012/08/13/2636334.html> (欧拉函数, 原根)
- [http://blog.csdn.net/v\\_july\\_v/article/details/18824517](http://blog.csdn.net/v_july_v/article/details/18824517)(荷兰国旗问题)
- <https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.07.md>(荷兰国旗问题)
- <https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.06.md>(“乌克兰国旗”问题: 奇偶排序)



# 我们在这里

---

☐ 更多算法面试题在 **7** | 七月算法 官网

■ <http://www.julyedu.com/>

☐ 免费视频

☐ 直播课程

☐ 问答社区

☐ contact us: 微博

■ @七月问答

■ @七月算法



---

感谢大家  
恳请大家批评指正！

