法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

数组

小象学院 ChinaHadoop.cn

邹博

主要内容

- 天平称量问题
- 查找众数和绝对众数
- 求局部最大值
- 第一个缺失的整数
- 旋转数组的最小值
- 寻找零子数组
- 数组的最大间隔
- 最大连续子数组
- 数字连续的子数组
- 荷兰国旗问题
- Cantor数组
- 附：子集和数问题/完美洗牌问题

天平与假币

- 有12枚硬币，其中有且只有1枚是假币，其重量与真币不同，但不知是重还是轻。现给定一架没有砝码的天平，问**至少**需要多少次称量才能**确保**找到这枚假币？

  - 进一步：如何证明某个方案是最少次数？

解析

- 随机将12枚硬币等分成3份，每份4枚；标记为A、B、C三份。
- 将A放于左侧，B放于右侧，用天平称量A和B，分三种情况：
  - 1. 天平平衡
  - 2. A(左)比B(右)重
  - 3. A(左)比B(右)轻
    - 与2对称，只分析2即可

1.天平平衡

- 天平平衡，说明A、B中都没有假币，假币在C中，将C中的4枚编号为甲乙丙丁。
- 取甲乙用天平称量，若平衡，说明甲乙是真币，丙丁有一枚是假币。
- 取甲丙用天平称量，若不平衡，说明丙是假币；若平衡，说明丙是真币，丁是假币。

2.A(左)比B(右)重

- 说明假币必然在A、B中，C中的4枚都是真币。将A中4枚硬币编号为1234，B中编号为5678，C中编号为甲乙丙丁。
- 选125放于左侧，34甲放于右侧；天平有三种情况：
  - 天平平衡：说明678含假币，且假币轻
  - 125比34甲重
    - 说明12含假币，且假币重
  - 125比34甲轻
    - 说明34含假币，且假币重
    - 或者5是假币，且假币轻
- 无论如何，最多再一次称量即可找到假币。

理论下界

- 一次天平称量能得到左倾、右倾、平衡3种情况，则把一次称量当成一位编码，该编码是3进制的。问题转换为：需要多少位编码，能够表示12呢？
  - 由于12的轻重未知，有两种可能，因此，需要用3进制表示24。
- 答：假定需要n位，则：$3^n \geq 24$
  - 取对数后计算得到$n \geq 2.89$，这表示至少3次才能找到该假币。

小结与思考

- "1.天平平衡"的情况下，若丁是假币，是无法知道丁是轻还是重，可否换一个称量方式，确保得知假币的轻重呢?
- 题目变成13枚硬币呢?
  - 有13枚硬币，其中有1枚是假币，但不知道是重还是轻。现给定一架没有砝码的天平，问至少需要多少次称量才能找到这枚假币?
  - 答：3次。

数组

- 类型相同的若干元素的有序集合，即数组。
- 数组的下标和元素建立了单向联系：
  - 通过下标可以在O(1)找到元素
  - 若元素之间无序，则查找某元素是否在数组中需要O(N)的时间复杂度。
  - 若元素之间有序，则可以使用折半查找确定某元素是否在数组，需要O(logN)的时间复杂度。

折半查找Code

```
int Find(const int* array, int size, int a)
{
    int nFrom = 0;
    int nTo = size-1;
    int nIndex;
    bool bFind = false;
    while(nFrom <= nTo)
    {
        nIndex = (nFrom + nTo) / 2;
        if(array[nIndex] == a)
        {
            bFind = true;
            break;
        }
        if(array[nIndex] > a)
            nTo = nIndex-1;
        else
            nFrom = nIndex+1;
    }
    if(bFind)
        return nIndex;
    return -1;
}
```

绝对众数

- 定义：给定N个数，称出现次数最多的数为众数；若某众数出现的次数大于N/2，称该众数为绝对众数。
  - 如：A={1,2,1,3,2}中，1和2都是众数，但都不是绝对众数。
  - 如：A={1,2,1,3,1}中，1是绝对众数。
- 已知给定的N个整数存在绝对众数，以最低的时空复杂度计算该绝对众数。

算法分析

- 删除数组A中两个不同的数，绝对众数不变。
  - 若两个数中有1个是绝对众数，则剩余的N-2个数中，绝对众数仍然大于(N-2)/2；
  - 若两个数中没有绝对众数，显然不影响绝对众数。
- 算法描述：
- 记m为候选绝对众数，出现次数为c，初始化为0。
- 遍历数组A：
  - 若c==0，则m=A[i]
  - 若c≠0且m≠A[i]，则同时删掉m和A[i]
  - 若c≠0且m==A[i]，则c++

Code

```
int Mode(int* a, int size)
{
    int count = 0;
    int m = a[0];
    for(int i = 0; i < size; i++)
    {
        if(count == 0)
        {
            m = a[i];
            count = 1;
        }
        else if(m != a[i])
        {
            count--;
        }
        else    //if(m == a[i])
        {
            count++;
        }
    }
    return m;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {8,8,1,1,1,8,1,1,6,1,8};
    int m = Mode(a, sizeof(a)/sizeof(int));
    cout << m << endl;
    return 0;
}
```

众数

- 给定N个整数，查找出现次数超过N/3次的的所有可能的数。
  - 注：有可能这样的数不存在。
  - 要求：时间复杂度为O(N)，空间复杂度为O(1)

问题分析

- 为表述方便，称出现次数大于N/3的数为“1/3众数”。
- 有：一个数组中的1/3众数不会超过2个。
  - 若存在3个1/3众数，则至少有3 (N/3+1)个数，与N个数矛盾
  - 3*(N/3+1)>N
- 采用“绝对众数”相似的统计思路：
  - 任意删除数组中的三个不相同的数，其中一个为1/3众数，则1/3众数不变。
    - 使用cm记录候选值m的出现次数，若cm为0，则重新选择候选值。

Code

```
void FindMode(const int* a, int size, vector<int>& mode)
{
    int m, n;    //候选值
    int cm = 0, cn = 0;
    int i;
    for(i = 0; i < size; i++)
    {
        if(cm == 0)
            m = a[i], cm = 1;
        else if(cn == 0)
            n = a[i], cn = 1;
        else if(a[i] == m)
            cm++;
        else if(a[i] == n)
            cn++;
        else    //同时删除a[i]、m、n
            cm--, cn--;
    }
    cm = cn = 0;
    for(i = 0; i < size; i++)
    {
        if(a[i] == m)
            cm++;
        else if(a[i] == n)
            cn++;
    }
    if(cm > size/3)
        mode.push_back(m);
    if(cn > size/3)
        mode.push_back(n);
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1, 2, 3, 2, 5, 2, 2, 3, 3, 2, 3};
    vector<int> mode;
    FindMode(a, sizeof(a)/sizeof(int), mode);
    Print(mode);
    return 0;
}
```

求局部最大值

- 给定一个无重复元素的数组A[0…N-1]，求找到一个该数组的局部最大值。
- 规定：在数组边界外的值无穷小。即：A[0] ＞A[-1]，A[N-1] ＞A[N]。从而可得如下局部最大值的形式化定义：

$$a^* = one\ of\ \{a[i] \mid a[i] > a[i-1]\text{且}a[i] > a[i+1], 0 \le i \le n-1\}$$

  - 遍历一遍得全局最大值，它显然是局部最大值
  - 可否有更快的办法？

问题分析

- 定义：若子数组Array[from,…,to]满足
  - Array[from] > Array[from-1]
  - Array[to] > Array[to+1]
- 称该子数组为“高原数组”。
  - 若高原数组长度为1，则该高原数组的元素为局部最大值。

算法描述

- 使用索引left、right分别指向数组首尾，根据定义，该数组为高原数组。
- 求中点mid=(left+right)/2
- A[mid] > A[mid+1]，子数组A[left…mid]为高原数组
  - 丢弃后半段：right=mid
- A[mid+1] > A[mid]，子数组A[mid…right]高原数组
  - 丢弃前半段：left=mid+1
- 递归直至left==right
  - 时间复杂度为O(logN)。

Code

```
int LocalMaximum(const int* A, int size)
{
    int left = 0;
    int right = size-1;
    int mid;
    while(left < right)
    {
        mid = (left + right) / 2;
        cout << mid << endl;
        if((A[mid] > A[mid+1])) //mid一定小于size-1
            right = mid;
        else
            left = mid+1;
    }
    return A[left];
}
```

第一个缺失的整数

- 给定一个数组A[0…N-1]，找到从1开始，第一个不在数组中的正整数。
  - 如3,5,1,2,-3,7,14,8输出4。

循环不变式

- 思路：将找到的元素放到正确的位置上，如果最终发现某个元素一直没有找到，则该元素即为所求。
- 循环不变式：如果某命题初始为真，且每次更改后仍然保持该命题为真，则若干次更改后该命题仍然为真。
- 为表述方便，下面的算法描述从1开始数。

利用循环不变式设计算法

- 假定前i-1个数已经找到，并且依次存放在A[1,2,…,i-1]中，继续考察A[i]：
  - 若A[i]＜i且A[i]≥1，则A[i]在A[1,2,…,i-1]中已经出现过，可以直接丢弃。
    - 若A[i]为负，则更应该丢弃它。
  - 若A[i]＞i且A[i]≤N，则A[i]应该置于后面的位置，即将A[A[i]]和A[i]交换。
    - 若A[i]＞N，由于缺失数据≥N，则A[i]丢弃。
    - 若A[A[i]]＝A[i]，则显然不必交换，直接丢弃A[i]即可。
  - 若A[i]＝i，则A[i]位于正确的位置上，则i加1，循环不变式扩大，继续比较后面的元素。

合并相同的分支

- 整理算法描述：
  - 若A[i]=i，i加1，继续比较后面的元素。
  - 若A[i]<i或A[i]>N或A[A[i]]=A[i]，丢弃A[i]
  - 若A[i]>i，则将A[A[i]]和A[i]交换。
- 思考：如何快速丢弃A[i]?
  - 将A[N]赋值给A[i]，然后N减1。

Code

```
int FirstMissNumber(int* a, int size)
{
    a--;    //从1开始数
    int i = 1;
    while(i <= size)
    {
        if(a[i] == i)
        {
            i++;
        }
        else if((a[i] < i) || (a[i] > size) || (a[i] == a[a[i]]))
        {
            a[i] = a[size];
            size--;
        }
        else //if(a[i] > i)
        {
            swap(a[a[i]], a[i]);
        }
    }
    return i;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {3,5,1,2,-3,7,4,8};
    int m = FirstMissNumber(a, sizeof(a) / sizeof(int));
    cout << m << endl;
    return 0;
}
```

查找旋转数组的最小值

- 假定一个排序数组以某个未知元素为支点做了旋转，如：原数组0 1 2 4 5 6 7旋转后得到4 5 6 7 0 1 2。请找出旋转后数组的最小值。假定数组中没有重复数字。

分析

- 旋转之后的数组实际上可以划分成两个有序的子数组：前面子数组的大小都大于后面子数组中的元素；
  - 4 5 6 7 0 1 2
  - 注意到实际上最小的元素就是两个子数组的分界线。

寻找循环数组最小值：4 5 6 7 0 1 2

- 用索引left，right分别指向首尾元素，元素不重复。
  - 若子数组是普通升序数组，则A[left]<A[right]。
  - 若子数组是循环升序数组，前半段子数组的元素全都大于后半段子数组中的元素：A[left]>A[right]
- 计算中间位置mid = (low+high)/2;
  - 显然，A[low…mid]与A[mid+1…high]必有一个是循环升序数组，一个是普通升序数组。
  - 若：A[mid]>A[high]，说明子数组A[mid+1,mid+2,…high]循环升序；更新low=mid+1；
  - 若：A[mid]<A[high]，说明子数组A[mid+1,mid+2,…high]普通升序；更新：high=mid

代码

```
int FindMin(int* num, int size)
{
    int low = 0;
    int high = size - 1;
    int mid;
    while(low < high)
    {
        mid = (high + low) / 2;
        if (num[mid] < num[high])    //最小值在左半部分
            high = mid;
        else if (num[mid] > num[high])  //最小值在右半部分
            low = mid + 1;
    }
    return num[low];
}
```

零子数组

- 求对于长度为N的数组A，求连续子数组的和最接近0的值。
- 如：
  - 数组A、1, -2, 3, 10, -4, 7, 2, -5
  - 它是所有子数组中，和最接近0的是哪个？

算法流程

- 申请比A长1的空间sum[-1,0…,N-1]，sum[i]是A的前i项和。
  - trick：定义sum[-1] = 0
- 显然有：$$\sum_{k=i}^{j} A_k = sum(j) - sum(i-1)$$
- 算法思路：
  - 对sum[-1,0…,N-1]排序，然后计算sum相邻元素的差的绝对值，最小值即为所求
  - 在A中任意取两个前缀子数组的和求差的最小值

零子数组的讨论

- 计算前n项和数组sum和计算sum相邻元素差的时间复杂度，都是O(N)，排序的时间复杂度认为是O(NlogN)，因此，总时间复杂度：O(NlogN)。
- 思考：如果需要返回绝对值最小的子数组本身呢？

Code

```
int MinSubarray(const int* a, int size)
{
    int* sum = new int[size+1]; //sum[i]:a[0...i-1]的和
    sum[0] = 0;
    int i;
    for(i = 0; i < size; i++)
    {
        sum[i+1] = sum[i] + a[i];
    }
    sort(sum, sum+size+1);
    int difference = abs(sum[1] - sum[0]);  //初始化
    int result = difference;
    for(i = 1; i < size; i++)
    {
        difference = abs(sum[i+1] - sum[i]);
        result = min(difference, result);
    }
    delete[] sum;
    return result;
}
```

最大连续子数组

- 给定一个数组A[0,…,n-1]，求A的连续子数组，使得该子数组的和最大。
- 例如
  - 数组： 1, -2, 3, 10, -4, 7, 2, -5,
  - 最大子数组：3, 10, -4, 7, 2

分析

- 记S[i]为以A[i]结尾的数组中和最大的子数组
- 则：$S[i+1] = \max(S[i]+A[i+1], A[i+1])$
- $S[0]=A[0]$
- 遍历i：$0 \le i \le n-1$
- 动态规划：最优子问题
- 时间复杂度：$O(n)$

动态规划Code

```
int MaxSubarray(const int* a, int size)
{
    if(!a || (size <= 0))
        return 0;

    int sum = a[0];        //当前子串的和
    int result = sum;      //当前找到的最优解
    for(int i = 1; i < size; i++)
    {
        if(sum > 0)
        {
            sum += a[i];
        }
        else
        {
            sum = a[i];
        }
        result = max(sum, result);
    }
    return result;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a[] = {1, -2, 3, 10, -4, 7, 2, -5};
    int m = MaxSubarray(a, sizeof(a)/sizeof(int));
    cout << m << '\n';
    return 0;
}
```

进一步思考该算法的可行性

- 定义：前缀和sum[i] = a[0] + a[1] + ...+a[i]
- 则：a[i,j]=sum[j]-sum[i-1](定义p[-1] = 0)

$$\sum_{k=i}^{j} a_k = sum(j) - sum(i-1)$$

- 算法过程
- 1. 求i前缀sum[i]：
  - 遍历i：0≤i≤n-1
  - sum[i]=sum[i-1]+a[i]
- 2. 计算以a[i]结尾的子数组的最大值
  - 对于某个i：遍历0≤j≤i，求sum[j]的最小值m
  - sum[i]-m即为以a[i]结尾的数组中最大的子数组的值
- 3. 统计sum[i]-m的最大值， 0≤i≤n-1
- 1、2、3步都是线性的，因此，时间复杂度O(n)。

思考

- 若除了输出最大子数组的和，还需要输出最大子数组本身，应该怎么做?

Code

```
int MaxSubarray(const int* a, int size, int& from, int& to)
{
    if(!a || (size <= 0))
    {
        from = to = -1;
        return 0;
    }

    from = to = 0;
    int sum = a[0];
    int result = sum;
    int fromNew;    //新的子数组起点
    for(int i = 1; i < size; i++)
    {
        if(sum > 0)
        {
            sum += a[i];
        }
        else
        {
            sum = a[i];
            fromNew = i;
        }
        if(result < sum)
        {
            result = sum;
            from = fromNew;
            to = i;
        }
    }
    return result;
}
```

数字连续的子数组

- 给定长度为N的数组A[0…N-1]，求递增且连续数字最长的子数组。
  - 如数组：1, 2, 3, 34, 56, 57, 58, 59, 60, 61, 99, 121的连续数字最长的一段为56, 57, 58, 59, 60, 61。

Code

```
int MaxSequence(const int* a, int size)
{
    int* p = new int[size];
    int i;
    for(i = 0; i < size; i++)
        p[i] = 1;

    int m = 1;
    for(i = 1; i < size; i++)
    {
        if(a[i] - a[i-1] == 1)
        {
            p[i] += p[i-1];
            m = max(p[i], m);
        }
    }
    delete[] p;
    return m;
}
```

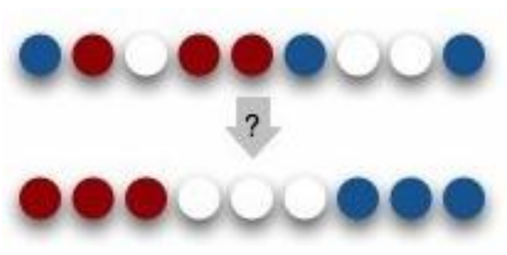
Code

```
int MaxSequence(const int* a, int size, int& from, int& to)
{
    from = to = 0;
    int* p = new int[size];
    int i;
    for(i = 0; i < size; i++)
        p[i] = 1;

    int m = 1;
    for(i = 1; i < size; i++)
    {
        if(a[i] - a[i-1] == 1)
        {
            p[i] += p[i-1];
            m = max(p[i], m);
            to = i;
        }
    }
    from = to - m + 1;
    delete[] p;
    return m;
}
```

荷兰国旗问题

- 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。
  - 之所以叫荷兰国旗问题，是因为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…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≠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≠cur，则A[cur]与A[begin]交换，begin++，cur不变

```
void Holland1(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

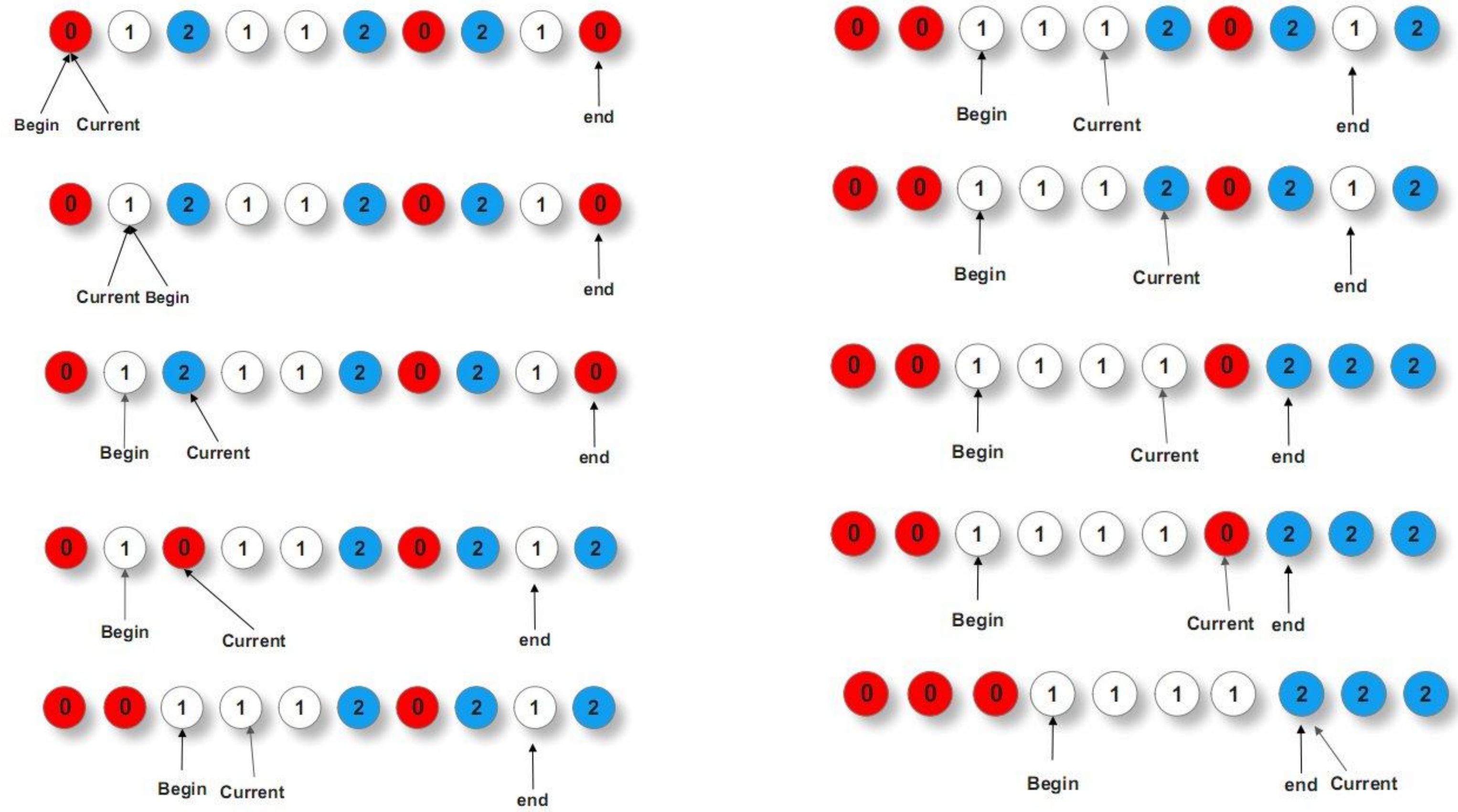
- cur扫过的位置，即：[begin,cur)区间内，一定没有2
  - 在前面的A[cur]==2中，已经被替换到数组后面了
- 因此：A[begin]要么是0，要么是1，不可能是2
- 考察begin指向的元素的值：
  - 归纳法：若begin≠cur，则必有A[begin]=1
- 因此，当A[cur]==0时，
  - 若begin≠cur，因为A[begin]==1，则交换后，A[cur]==1，此时，可以**cur++**；

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≠cur，则A[cur]与A[begin]交换，begin++，cur++

```
void Holland2(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++;
            }
        }
    }
}
```

优化后的图示演示



0 1 2 1 1 2 0 2 1 0
Begin Current
end
0 1 2 1 1 2 0 2 1 0
Current Begin
end
0 1 2 1 1 2 0 2 1 0
Begin Current
end
0 1 0 1 1 2 0 2 1 2
Begin Current
end
0 0 1 1 1 2 0 2 1 2
Begin Current
end
0 0 1 1 1 2 0 2 1 2
Begin Current
end
0 0 1 1 1 2 0 2 1 2
Begin Current
end
0 0 1 1 1 1 0 2 2 2
Begin Current
end
0 0 1 1 1 1 0 2 2 2
Begin Current end
0 0 0 1 1 1 1 2 2 2
Begin end Current

Code3

```
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begin/cur/end的循环中，cur遇到0和遇到2，begin和end的处理方式不对称
  - 遇到0：begin++,cur++
  - 遇到2：end--，cur不变
  - 网络流行的荷兰国旗算法的版本，往往直接给出上述结论
- 若初值给定如下：
  - begin=0、cur=N-1、end=N-1，如何完成代码？

Code4

```
void Hollandr(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++;
        }
    }
}
```

循环不变式

- 循环不变式：如果某命题初始为真，且每次更改后仍然保持该命题为真，则若干次更改后该命题仍然为真。

循环不变式的应用

- 三个变量begin、cur、end，将数组分成4个区域：
  - [0,begin)：所有数据都是0
  - [begin,cur)：所有数据都是1
  - (end,size-1]：所有数据都是2
  - [cur,end]：未知
- 循环不变式：
  - 初值begin=cur=0，end=size-1，前三个区间都为空集，满足以上4个条件。
  - 遍历cur，根据a[cur]的值做相应处理，直到区间[cur,end)为空，即cur==end时退出。
  - 得代码如Code3所示。

“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
  - 在子集和数等需要分支限界搜索的问题中，往往可以作为预处理，方便限界条件的给出。

荷兰国旗问题的其他方案

- 思路1
  - 将0、1、2分别计数，根据计数值c_0、c_1、c_2：前c_0个元素赋值为0，中间c_1个元素赋值为1，最后c_2个元素赋值为2
  - 是否可行？
- 思路2
  - 将(0,1)(2)根据快速排序的Partition，划分为两部分
    - 如PivotKey选择1.5
  - 将(0)(1)根据快速排序的Partition，分成两部分
    - 如PivotKey选择0.5
  - "两次Partition == 一次荷兰国旗"

荷兰国旗问题的实践应用

- 优化快速排序的Partition过程
- 快速排序根据PivotKey分成大于、小于等于两部分
  - 或者大于等于、小于两部分
- 根据与PivotKey的大小，将Partition过程改造成大于、等于、小于三部分
  - 优点：对于待排序的等于PivotKey的数值，可以在执行下一次Partition时直接跳过，利于数据规模的降低

数组的最大间隔

- 给定整数数组A[0…N-1]，求这N个数排序后最大间隔。如：1,7,14,9,4,13的最大间隔为4。
  - 排序后：1,4,7,9,13,14，最大间隔是13-9=4
  - 显然，对原数组排序，然后求后项减前项的最大值，即为解。
  - 可否有更好的方法？

问题分析

- 假定N个数的最大最小值为max，min，则这N个数形成N-1个间隔，其最小值是 $\frac{max-min}{N-1}$
  - 如果N个数完全均匀分布，则间距全部是 $\frac{max-min}{N-1}$ 且最小；
  - 如果N个数不是均匀分布，间距不均衡，则最大间距必然大于 $\frac{max-min}{N-1}$

解决思路

- 思路：将N个数用间距$\frac{max-min}{N-1}$分成N-1个区间，则落在同一区间内的数不可能有最大间距。统计后一区间的最小值与前一区间的最大值的差即可。
  - 若没有任何数落在某区间，则该区间无效，不参与统计。
  - 显然，这是借鉴桶排序/Hash映射的思想。

桶的数目

- 同时，N-1个桶是理论值，会造成若干个桶的数目比其他桶大1，从而造成统计误差。
  - 如：7个数，假设最值为10、80，如果适用6个桶，则桶的大小为70/6=11.66，每个桶分别为：[10,21]、[22,33]、[34,44]、[45,56]、[57,68]、[69,80]，存在大小为12的桶，比理论下界11.66大。
- 因此，使用N个桶。

Code

```
typedef struct tagSBucket
{
    bool bValid;
    int nMin;
    int nMax;

    tagSBucket() : bValid(false) {}

    void Add(int n) //将数n加入到桶中
    {
        if(!bValid)
        {
            nMin = nMax = n;
            bValid = true;
        }
        else
        {
            if(nMax < n)
                nMax = n;
            else if(nMin > n)
                nMin = n;
        }
    }
}SBucket;
```

```
int CalcMaxGap(const int* A, int size)
{
    //求最值
    SBucket* pBucket = new SBucket[size];
    int nMax = A[0];
    int nMin = A[0];
    int i;
    for(i = 1; i < size; i++)
    {
        if(nMax < A[i])
            nMax = A[i];
        else if(nMin > A[i])
            nMin = A[i];
    }

    //依次将数据放入桶中
    int delta = nMax - nMin;
    int nBucket;    //某数应该在哪个桶中
    for(i = 0; i < size; i++)
    {
        nBucket = (A[i] - nMin) * size / delta;
        if(nBucket >= size)
            nBucket = size-1;
        pBucket[nBucket].Add(A[i]);
    }

    //计算有效桶的间隔
    i = 0;  //首个桶一定是有效的
    int nGap = delta / size;    //最小间隔
    int gap;
    for(int j = 1; j < size; j++)//i是前一个桶，j是后一个桶
    {
        if(pBucket[j].bValid)
        {
            gap = pBucket[j].nMin - pBucket[i].nMax;
            if(nGap < gap)
                nGap = gap;
            i = j;
        }
    }
    return nGap;
}
```

Cantor数组

```
void CantorExpansion(const int* a, int* b, int size)
{
    int i,j;
    for(i = 0; i < size; i++)
    {
        b[i] = 0;
        for(j = i+1; j < size; j++)
        {
            if(a[j] < a[i])
                b[i]++;
        }
    }
}
```

- 已知数组A[0…N-1]乱序着前N个正整数，现统计后缀数组A[i+1…N-1]中小于元素A[i]的数目，并存放在数组C[i]中。如给定数组A={4,6,2,5,3,1}，得到数组C={3,4,1,2,1,0}。
- 问：给定数组C={3,4,1,2,1,0}，如何恢复数组A？
  - 我们称A为原数组，C为Cantor数组。

Code

- 原数组
- Cantor数组

```
void CantorExpansion(const int* a, int* b, int size)
{
    int i,j;
    for(i = 0; i < size; i++)
    {
        b[i] = 0;
        for(j = i+1; j < size; j++)
        {
            if(a[j] < a[i])
                b[i]++;
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int p[] = {2,6,4,5,3,1};
    int size = sizeof(p) / sizeof(int);
    int* a = new int[size];
    CantorExpansion(p, a, size);
    Print(a, size);
    delete[] a;
    return 0;
}
```

问题分析

- Cantor数组：{3,4,1,2,1,0}
- 原数组 ：{4,6,2,5,3,1}

- 给定顺序数组B={1,2,3…N-1,N}，从0开始数
- 考察Cantor数组的首位C[0]：
  - 小于A[0]的个数为C[0]，则A[0]为B[C[0]]
- 在序列数组B中删除B[C[0]]，仍然满足以上性质。

Code

```
void CantorExpansionR(const int* a, int* result, int size)
{
    int i;
    vector<int> v(size);
    for(i = 0; i < size; i++)
        v[i] = i+1;

    for(i = 0; i < size; i++)
    {
        result[i] = v[a[i]];
        v.erase(v.begin()+a[i]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int c[] = {3,4,1,2,1,0};
    int size = sizeof(c) / sizeof(int);
    int* a = new int[size];
    CantorExpansionR(c, a, size);
    Print(a, size);
    return 0;
}
```

进一步思考

- 以上代码空间复杂度为O(N)，时间复杂度为$O(N^2)$，若允许更改数组C，可否降低空间复杂度？
  - Cantor数组：{3,4,1,2,1,0}
  - 原数组　：{4,6,2,5,3,1}
- 考察Cantor数组中第一个出现0的位置：它表示位于该位置右侧的所有元素都大于该元素，则该元素必然是最小的。
  - 每次找到第一个0后，将0左侧的Cantor值都减一，重复以上操作。
  - 空间复杂度为O(1)。

Code

```
void CantorExpansionR2(int* a, int* result, int size)
{
    memset(result, 0, sizeof(int)*size);    //赋为无效值
    int i,j;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j++)
        {
            if(result[j] != 0)
                continue;
            if(a[j] == 0)
                break;
            a[j]--;
        }
        result[j] = i+1;
    }
}
```

总结与思考

- Cantor数组：{3,4,1,2,1,0}
- 原数组 ：{4,6,2,5,3,1}
- 将Cantor数组的每个元素都指定各自的权重：c[i]的权重为(n-1-i)的阶乘，则Cantor数组与一个整数一一对应，该数称为原数组的Cantor展开数，从Cantor展开数求Cantor数组的过程称为Cantor展开。
- Cantor数组中元素的和，表示原数组中逆序对的个数，问：给定数组A，如果计算A的逆序对个数？
  - 思考时间复杂度为O(NlogN)的算法。

总结与思考

- 数组和字符串这两类问题往往具有一致性。
- 由于计算机体系结构中内存的顺序存储，这种物理存储结构一定意义下决定了几乎所有问题都可以归类为逻辑上的“数组”。
  - 无所不包
  - 逻辑上的非线性结构(如树、图等)往往可以用数组做存储结构。
- 思考：TopK问题在O(N)解法？

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象学院
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

附录

- 子集和数问题 N-Sum
- 完美洗牌问题

子集和数问题 N-Sum

- 已知数组A[0…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…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…N-1]的元素都大于0
- 考察向量x[0…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…i-1]确定的A[0…i-1]的和为has；
- A[i,i+1,…N-1]的和为residue(简记为r)；
  - has+a[i]≤sum并且has+r≥sum：x[i]可以为1；
  - has+(r-a[i])>= 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…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；

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);
        }
    }
}
```

附：完美洗牌算法

- 长度为2n的数组$\{a_1,a_2,a_3,\ldots,a_n,b_1,b_2,b_3,\ldots,b_n\}$，经过整理后变成$\{a_1,b_1,a_2,b_2,a_3,b_3,\ldots\ldots,a_n,b_n\}$，要求时间复杂度O(n)，空间复杂度O(1)。

  - 题目比较“数学”，当做阅读材料即可。

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a1，a2，a3，a4，b1，b2，b3，b4
- 第①步：交换最中间的两个元素a4，b1，序列变成：
  - a1，a2，a3，b1，a4，b2，b3，b4
- 第②步，让最中间的两对元素各自交换：
  - a1，a2，b1，a3，b2，a4，b3，b4
- 第③步，交换最中间的三对元素，序列变成：
  - a1，b1，a2，b2，a3，b3，a4，b4

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

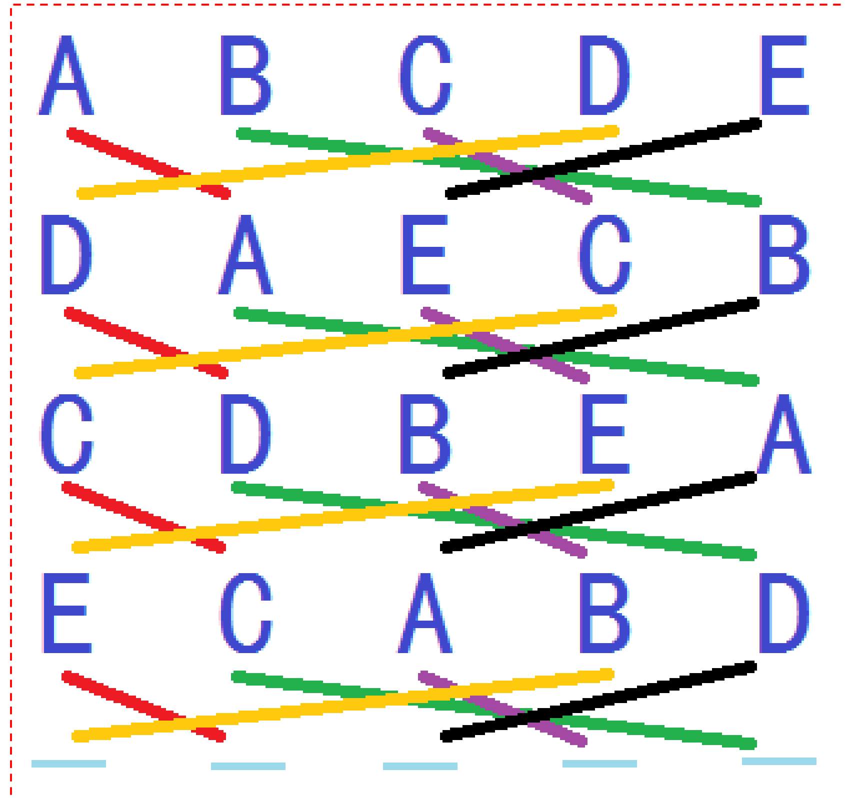
1 6 2 7 3 8 4 9 5 X

中间交换

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

玩乐中带来的算法思维

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



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

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

- a1,a2,a3,...an,b1,b2,b3..bn → b1,a1,b2,a2,b3,a3...bn,an
- 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a1到了原第2个元素a2的位置，即1->2；
  - 第2个元素a2到了原第4个元素a4的位置，即2->4；
  - 第3个元素a3到了原第6个元素b2的位置，即3->6；
  - 第4个元素a4到了原第8个元素b4的位置，即4->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b1到了原第1个元素a1的位置，即5->1；
  - 第6个元素b2到了原第3个元素a3的位置，即6->3；
  - 第7个元素b3到了原第5个元素b1的位置，即7->5；
  - 第8个元素b4到了原第7个元素b3的位置，即8->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
- $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 1$
- $3 \rightarrow 6 \rightarrow 3$

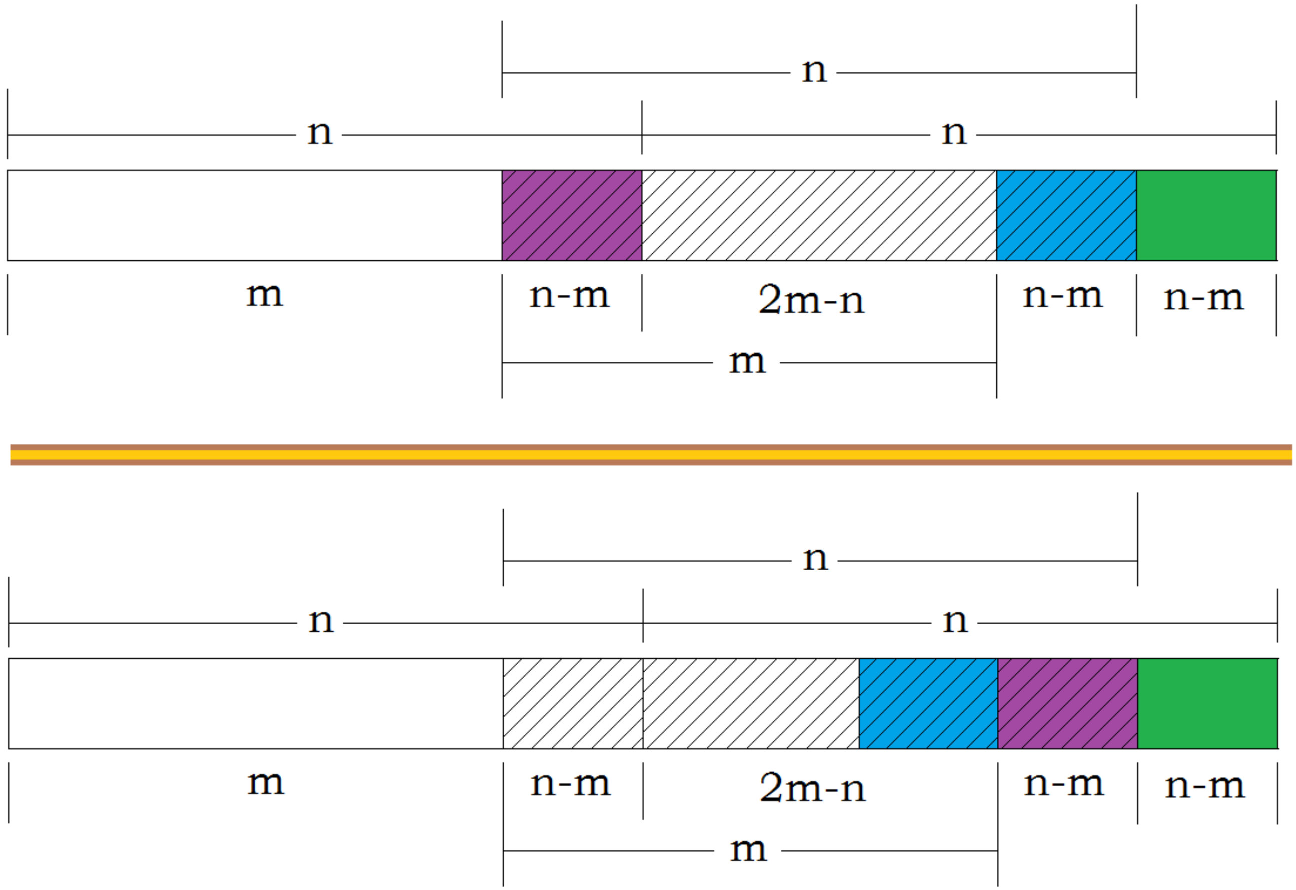
```
//数组下标从1开始，from是圈的头部，mod为 2 * n + 1
void CycleLeader(int *a, int from, int mod)
{
    int t,i;

    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，...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 $A[1, \ldots, 2n]$

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 $A[m+1, \ldots, n+m]$ by a distance m

Step 3. For each $i \in \{0, 1, \ldots, k-1\}$, 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 $A[2m+1, \ldots, 2n]$.

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≤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 + 1)/m >= 3; ++k, m *= 3)
          ;
        m /= 2;
        // 2m = 3^k - 1 , 3^k <= 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;
    }
    if(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}mod 9
- ={1,2,4,8,7,5}
- 而φ(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 $(Z/3^k)^*$ 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 $(Z/3^k)^*$, 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 $3^s 2^t$ (*modulo* 3^k) for $1 \leq t \leq \varphi(3^k)$ (where φ is the Euler-totient function). Since 2 is a generator of $(Z/3^k)^*$, 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 $1, 3, 3^2, \ldots, 3^{k-1}$ 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a1,b1,a2,b2……an,bn，而完美洗牌算法输出是b1,a1,b2,a2,……bn,an,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b1,a1,b2,a2,……bn,an，要求输出a1,a2,a3,……an,b1,b2,b3,……bn。
  - 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a1,a2,……an, b1,b2,……bn, c1,c2,……cn，要求输出是c1,b1,a1,c2,b2,a2,……cn,bn,an
  - 2付牌的结论：2是群(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(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