数组

七月算法 邹博

2015年10月24日

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]。
- 显然，遍历一遍可以找到全局最大值，而全局最大值显然是局部最大值。
- 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]应该位于后面的位置上，则将A[A[i]]和A[i]交换。
    - 若A[i]≥N，由于缺失数据一定小于N，则A[i]丢弃。
  - 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或者A[i] > N，则丢弃A[i]
  - 若A[i] > i，则将A[A[i]]和A[i]交换。
  - 若A[i] = i，i加1，继续比较后面的元素。
- 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) || (a[i] > size))
        {
            a[i] = a[size];
            size--;
        }
        else if(a[i] > i)
        {
            swap(a[a[i]], a[i]);
        }
        else  //if(a[i] == i)
        {
            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

```cpp
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;
}
```

最大子数组和

- 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

- 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)。

进一步的分析

- 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： $0 \leqslant i \leqslant n-1$
- 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;
}
```

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？

参考代码

```
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;
}
```

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

```cpp
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;
```

```cpp
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;
}
```

字符串的全排列

- 给定字符串S[0…N-1]，设计算法，枚举S的全排列。

递归算法

- 以字符串1234为例：
- 1 – 234
- 2 – 134
- 3 – 214
- 4 – 231
- 如何保证不遗漏
  - 保证递归前1234的顺序不变

递归Code

```cpp
void Print(const int* a, int size)
{
	for(int i = 0; i < size; i++)
		cout << a[i] << ' ';
	cout << endl;
}

void Permutation(int* a, int size, int n)
{
	if(n == size-1)
	{
		Print(a, size);
		return;
	}
	for(int i = n; i < size; i++)
	{
		swap(a[i], a[n]);
		Permutation(a, size, n+1);
		swap(a[i], a[n]);
	}
}

int main(int argc,char* argv[])
{
	int a[] = {1,2,3,4};
	Permutation(a, sizeof(a)/sizeof(int), 0);
	return 0;
}
```

```
1234
1243
1324
1342
1432
1423
2134
2143
2314
2341
2431
2413
3214
3241
3124
3142
3412
3421
4231
4213
4321
4312
4132
4123
```

如果字符有重复

- 去除重复字符的递归算法
- 以字符1223为例：
- 1 – 223
- 2 – 123
- 3 – 221

- 带重复字符的全排列就是每个字符分别与它后面非重复出现的字符交换。
- 即：第i个字符与第j个字符交换时，要求[i,j)中没有与第j个字符相等的数。

Code

1:	1223
2:	1232
3:	1322
4:	2123
5:	2132
6:	2213
7:	2231
8:	2321
9:	2312
10:	3221
11:	3212
12:	3122

```
bool IsDuplicate(const int* a, int n, int t)
{
    while(n < t)
    {
        if(a[n] == a[t])
            return false;
        n++;
    }
    return true;
}

void Permutation(int* a, int size, int n)
{
    if(n == size-1)
    {
        Print(a, size);
        return;
    }
    for(int i = n; i < size; i++)
    {
        if(!IsDuplicate(a, n, i))//a[i]是否与[n,i)重复
            continue;
        swap(a[i], a[n]);
        Permutation(a, size, n+1);
        swap(a[i], a[n]);
    }
}

int main(int argc,char* argv[])
{
    int a[] = {1,2,2,3};
    Permutation(a, sizeof(a)/sizeof(int), 0);
    return 0;
}
```

重复字符的全排列递归算法时间复杂度

- $\because f(n) = n*f(n-1) + n^2$
- $\because f(n-1)=(n-1)*f(n-2) + (n-1)^2$
- $\therefore f(n) = n*((n-1)*f(n-2) + (n-1)^2) + n^2$
- $\because f(n-2) = (n-2)*f(n-3) + (n-2)^2$
- $\therefore f(n) = n*(n-1)*((n-2)*f(n-3) + (n-2)^2) + n*(n-1)^2 + n^2$
- $= n*(n-1)*(n-2)*f(n-3) + n*(n-1)*(n-2)^2 + n*(n-1)^2 + n^2$
- $=.......$
- $< n! + n! + n! + n! + ... + n!$
- $= (n+1)*n!$
- 时间复杂度为$O((n+1)!)$
  - 注： 当n足够大时： $n! > n+1$

空间换时间

```
void Permutation(char* a, int size, int n)
{
    if(n == size-1)
    {
        Print(a, size);
        return;
    }
    int dup[256] = {0};
    for(int i = n; i < size; i++)
    {
        if(dup[a[i]] == 1)
            continue;
        dup[a[i]] = 1;
        swap(a[i], a[n]);
        Permutation(a, size, n+1);
        swap(a[i], a[n]);
    }
}

int main(int argc,char* argv[])
{
    char str[] = "abbc";
    Permutation(str, sizeof(str)/sizeof(char)-1, 0);
    return 0;
}
```

空间换时间的方法

- 如果是单字符，可以使用mark[256]；
- 如果是整数，可以遍历整数得到最大值max和最小值min，使用mark[max-min+1]；
- 如果是浮点数或其他结构，考虑使用Hash。
  - 事实上，如果发现整数间变化太大，也应该考虑使用Hash；
  - 可以认为整数/字符的情况是最朴素的Hash。

全排列的非递归算法

- 起点：字典序最小的排列，例如12345
- 终点：字典序最大的排列，例如54321
- 过程：从当前排列生成字典序刚好比它大的下一个排列
- 如：21543的下一个排列是23145
  - 如何计算？

21543的下一个排列的思考过程

- 逐位考察哪个能增大
  - 一个数右面有比它大的数存在，它就能增大
  - 那么最后一个能增大的数是——$x = 1$
- 1应该增大到多少？
  - 增大到它右面比它大的最小的数——$y = 3$
- 应该变为23xxx
- 显然，xxx应由小到大排：145
- 得到23145

全排列的非递归算法：整理成算法语言

- 步骤：后找、小大、交换、翻转——
- 后找：字符串中最后一个升序的位置i，即：S[k]>S[k+1](k>i)，S[i]<S[i+1]；
- 查找(小大)：S[i+1…N-1]中比Ai大的最小值Sj；
- 交换：Si，Sj；
- 翻转：S[i+1…N-1]
  - 思考：交换操作后，S[i+1…N-1]一定是降序的
- 以926520为例，考察该算法的正确性。

非递归算法Code

```
void Reverse(int* from, int* to)
{
    int t;
    while(from < to)
    {
        t = *from;
        *from = *to;
        *to = t;
        from++;
        to--;
    }
}
```

```
bool GetNextPermutation(int* a, int size)
{
    //后找
    int i = size-2;
    while((i >= 0) && (a[i] >= a[i+1]))
        i--;
    if(i < 0)
        return false;

    //小大
    int j = size-1;
    while(a[j] <= a[i])
        j--;

    //交换
    swap(a[j], a[i]);

    //翻转
    Reverse(a+i+1, a+size-1);
    return true;
}

int main(int argc,char* argv[])
{
    int a[] = {1,2,2,3};
    int size = sizeof(a)/sizeof(int);
    Print(a, size);
    while(GetNextPermutation(a, size))
        Print(a, size);
    return 0;
}
```

几点说明

- 下一个排列算法能够天然的解决重复字符的问题！
  - 不妨还是考察926520的下一个字符串
- C++STL已经在Algorithm中集成了next_permutation
- 可以将给定的字符串A[0…N-1]首先升序排序，然后依次调用next_permutation直到返回false，即完成了非递归的全排列算法。

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			

```cpp
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);
        }
    }
}
```

求字符串的最长回文子串

- 回文子串的定义：
  - 给定字符串str，若s同时满足以下条件：
    - s是str的子串
    - s是回文串
  - 则，s是str的回文子串。
- 该算法的要求，是求str中最长的那个回文子串。

解法1 – 枚举中心位置

```
int LongestPalindrome(const char *s, int n)
{
   int i, j, max;
   if (s == 0 || n < 1)
      return 0;
   max = 0;

   for (i = 0; i < n; ++i) { // i is the middle point of the palindrome
      for (j = 0; (i - j >= 0) && (i + j < n); ++j) // if the length of the palindrome is odd
      if (s[i - j] != s[i + j])
         break;
      if (j * 2 + 1 > max)
         max = j * 2 + 1;
      for (j = 0; (i - j >= 0) && (i + j + 1 < n); ++j) // for the even case
      if (s[i - j] != s[i + j + 1])
         break;
      if (j * 2 + 2 > max)
         max = j * 2 + 2;
   }
   return max;
}
```

算法解析 step1——预处理

- 因为回文串有奇数和偶数的不同。判断一个串是否是回文串，往往要分开编写，造成代码的拖沓。
- 一个简单的事实：长度为n的字符串，共有n-1个“邻接”，加上首字符的前面，和末字符的后面，共n+1的“空”(gap)。因此，字符串本身和gap一起，共有2n+1个，必定是奇数；
  - abbc → #a#b#b#c#
  - aba → #a#b#a#
- 因此，将待计算母串扩展成gap串，计算回文子串的过程中，只考虑奇数匹配即可。

数组int P[size]

- 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？)

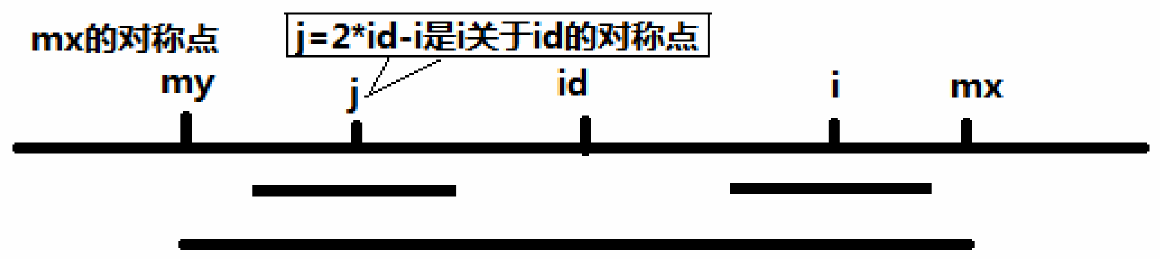
分析算法核心

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

- 我们的任务：假定已经得到了前i个值，考察i+1如何计算
  - 即：在P[0...i-1]已知的前提下，计算P[i]的值。换句话说，算法的核心，是**在P[0...i-1]已知的前提下，能否给P[i]的计算提供一点有用的信息**呢？
- 1、通过简单的遍历，得到i个三元组{k,P[k],k+P[k]}，0≤k≤i-1
  - trick：以k为中心的字符形成的最大回文子串的最右位置是k+P[k]-1
- 2、以k+P[k]为关键字，挑选出这i个三元组中，k+P[k]最大的那个三元组，不妨记做(id, P[id]，P[id]+id)。进一步，为了简化，记mx=P[id]+id，因此，得到三元组为(id,P[id],mx)，这个三元组的含义非常明显：所有i个三元组中，向右到达最远的位置，就是mx；
- 3、在计算P[i]的时候，考察i是否落在了区间[0,mx)中；
  - 若i在mx的右侧，说明[0,mx)没有能够控制住i，P[0..i-1]的已知，无法给P[i]的计算带来有价值信息；
  - 若i在mx的左侧，说明[0,mx)控制(也有可能部分控制)了i，现在以图示来详细考察这种情况。

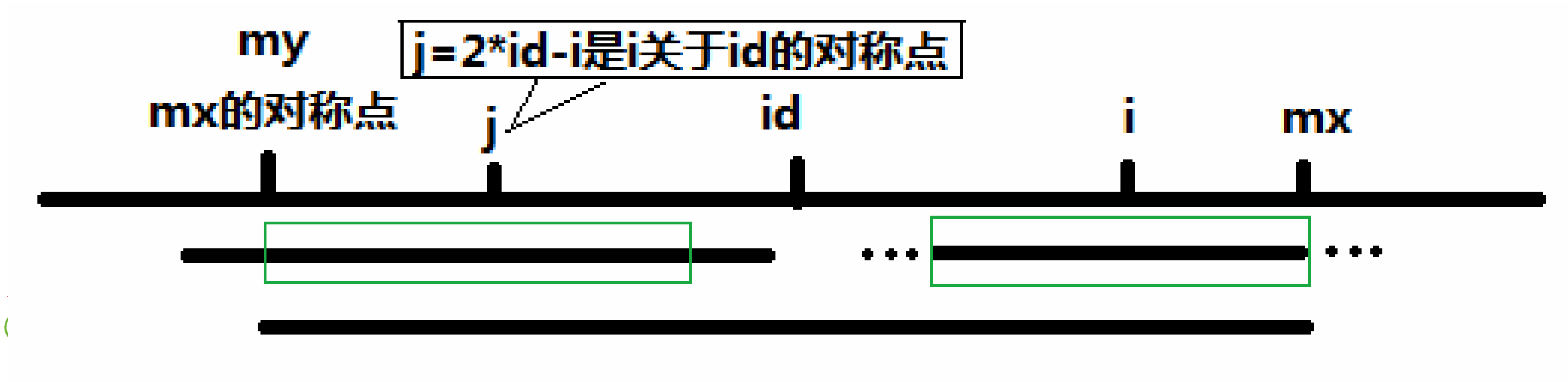
Manacher递推关系

- 记i关于id的对称点为j(=2*id-i)，若此时满足条件mx-i>P[j]：
- 记my为mx关于id的对称点(my=2*id-mx)；
- 由于以S[id]为中心的最大回文子串为S[my+1…id…mx-1]，即：S[my+1…,id]与S[id,…,mx-1]对称，而i和j关于id对称，因此P[i]=P[j](P[j]是已知的)。



Manacher递推关系

- 记i关于id的对称点为j(=2*id-i)，若此时满足条件mx-i＜P[j]：
- 记my为mx关于id的对称点(my=2*id-mx)；
- 由于以S[id]为中心的最大回文子串为S[my+1…id…mx-1]，即：S[my+1…,id]与S[id…,mx-1]对称，而i和j关于id对称，因此P[i]至少等于mx-i(图中绿色框部分)。



Manacher Code

```
void Manacher(char* s, int* P)
{
    int size = strlen(s);
    P[0] = 1;
    int id = 0;
    int mx = 1;
    for(int i = 1; i < size; i++)
    {
        if(mx > i)
        {
            P[i] = min(P[2*id-i], mx-i);
        }
        else
        {
            P[i] = 1;
        }
        for(; s[i+P[i]] == s[i-P[i]]; P[i]++);

        if(mx < i+P[i])
        {
            mx = i + P[i];
            id = i;
        }
    }
}
```

原始算法的个人改进意见

- $P[j] > mx - i$；$P[i] = mx - i$
- $P[j] < mx - i$；$P[i] = P[j]$
- $P[j] = mx - i$；$P[i] \geqslant P[j]$
  - 基本Manacher算法，红色的等号都是$\geqslant$

Manacher改进版

```
void Manacher(char* s, int* P)
{
    int size = strlen(s);
    P[0] = 1;
    int id = 0;
    int mx = 1;
    for(int i = 1; i < size; i++)
    {
        if(mx > i)
        {
            if(P[2*id-i] != mx-i)
            {
                P[i] = min(P[2*id-i], mx-i);
            }
            else
            {
                P[i] = P[2*id-i];
                for(; s[i+P[i]] == s[i-P[i]]; P[i]++);
            }
        }
        else
        {
            P[i] = 1;
            for(; s[i+P[i]] == s[i-P[i]]; P[i]++);
        }

        if(mx < i+P[i])
        {
            mx = i + P[i];
            id = i;
        }
    }
}
```

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)。

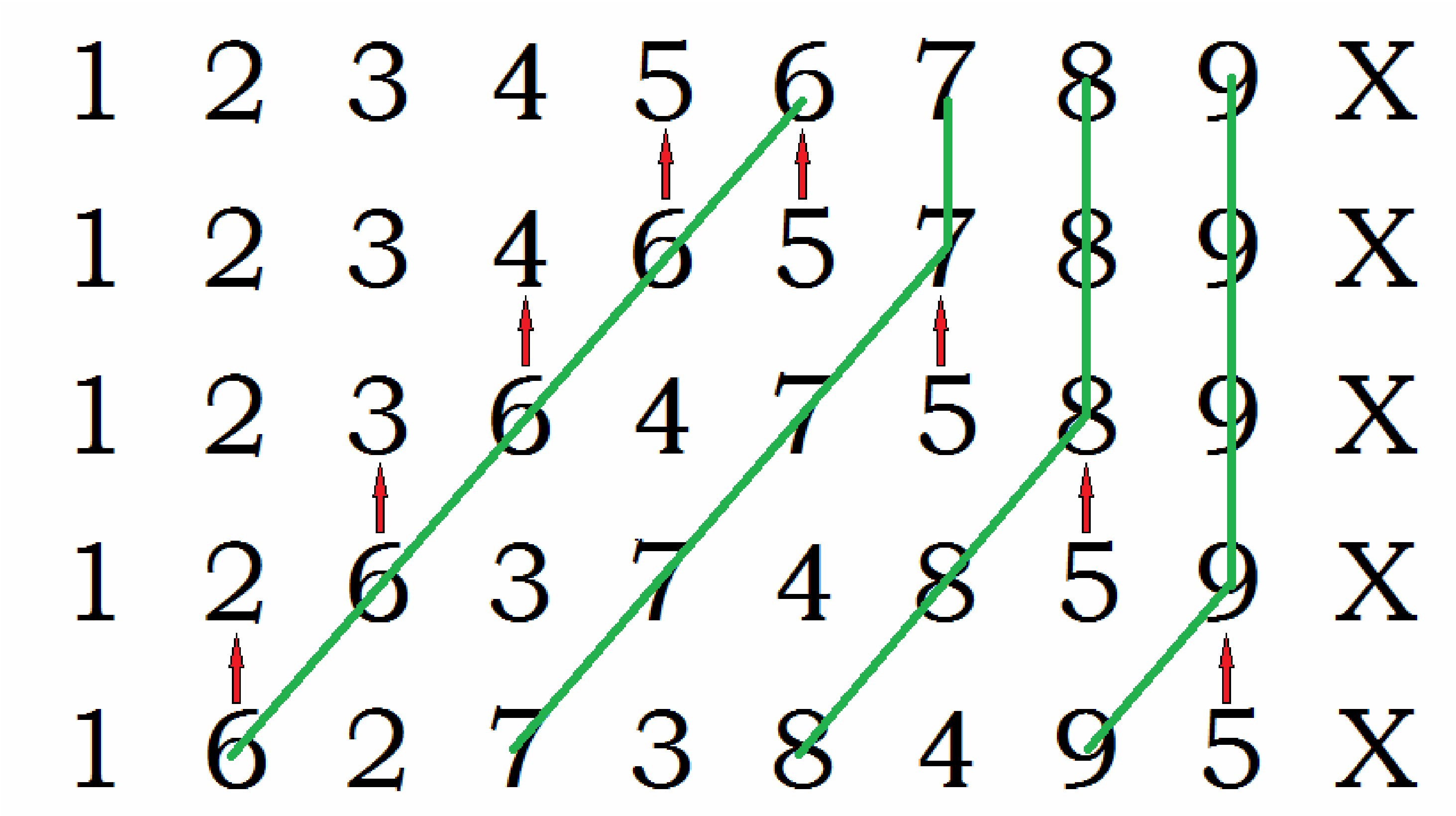
中间交换

- 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

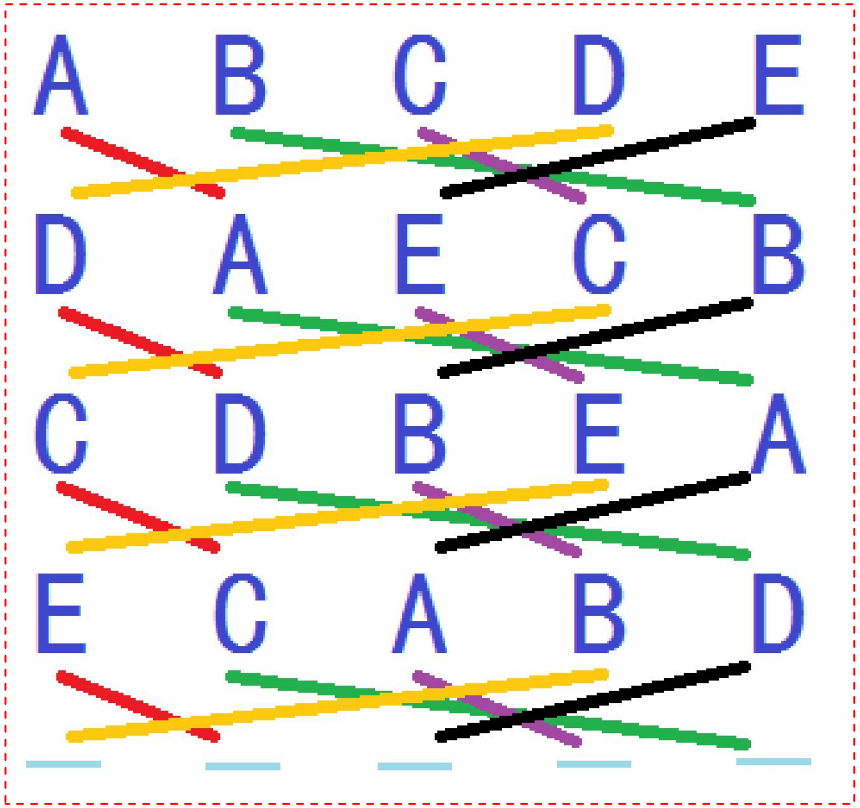
1	2	3	4	5	6	7	8	9	X
1	2	3	4	6	5	7	8	9	X
1	2	3	6	4	7	5	8	9	X
1	2	6	3	7	4	8	5	9	X
1	6	2	7	3	8	4	9	5	X

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

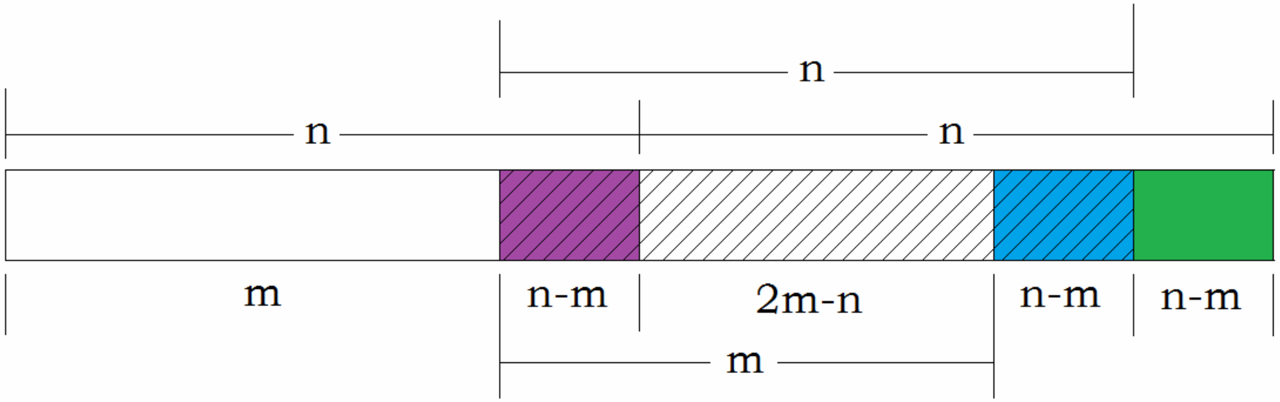
- 我们得到两个圈
- 1 → 2 → 4 → 8 → 7 → 5 → 1
- 3 → 6 → 3

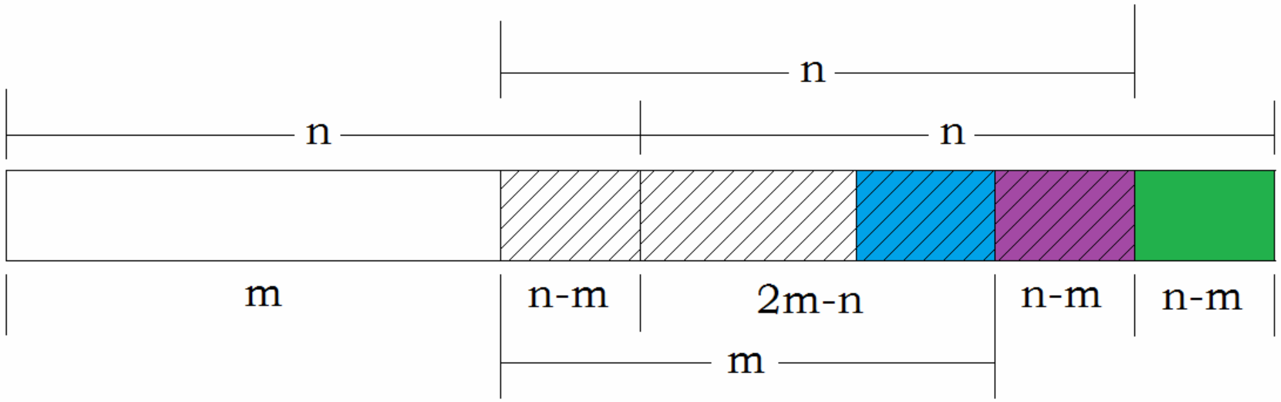
```
//数组下标从1开始，from是圈的头部，mod为 2 * n + 1
void CycleLeader(int *a, int from, int mod)
{
    int t,i;

    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

- $\{2^0,2^1\}=\{1,2\}$
- $\{2^0,2^1,2^2,2^3,2^4,2^5,2^6,2^7,2^8\} \bmod 9$
- $=\{1,2,4,8,7,5\}$
- 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 $(Z/3^k)^*$ 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

我们在这里

7 七月算法 http://www.julyedu.com/

- 视频/课程/社区

七月题库APP：Android/iOS

- http://www.julyapp.com/

微博

- @研究者July
- @七月题库
- @邹博_机器学习

微信公众号

- julyedu

感谢大家

恳请大家批评指正！