数组

七月算法 邹博

2015年10月24日

天平与假币

- 有12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？

- 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：$3^n \geqslant 24$
  - 取对数后计算得到$n \geqslant 2.89$，这表示至少3次才能找到该轻质的假币。

再次思考

- 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。

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

```cpp
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

- 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;
}
```

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？

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

```
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;
}
```

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

```
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(前)与第j个字符(后)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

- $\because f(n) = n*f(n-1) + n^2$
- $\because f(n-1)=(n-1)*f(n-2) + (n-1)^2$
- $\therefore f(n) = n*((n-1)*f(n-2) + (n-1)^2) + n^2$
- $\because f(n-2) = (n-2)*f(n-3) + (n-2)^2$
- $\therefore f(n) = n*(n-1)*((n-2)*f(n-3) + (n-2)^2) + n*(n-1)^2 + n^2$
- $= n*(n-1)*(n-2)*f(n-3) + n*(n-1)*(n-2)^2 + n*(n-1)^2 + n^2$
- $=\ldots\ldots.$
- $< n! + n! + n! + n! + \ldots + n!$
- $= (n+1)*n!$
- 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

- 下排列算法能够天然解决重复字符的问题！
  - 不妨还是考察926520的下一个字符串
- STL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。

我们在这里

- 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！