《编程之美—微软技术面试心得》

IT从业人员“面试真题”。倡导： 思考力等于竞争力。

揭露微软面试内幕：从平常的问题入手，深入挖掘，考察动手能力和独到的思路。 解答面试疑惑： IT 企业重视什么样的能力、需要什么样的技术人才、如何甄别人才。

《编程之美—微软技术面试心得》文章节选：

一摞烙饼的排序

星期五的晚上，一帮同事在希格玛大厦附近的“硬盘酒吧”多喝了几杯。程序员多喝了几

杯之后谈什么呢？自然是算法问题。有个同事说：

“我以前在餐馆打工，顾客经常点非常多的烙饼。店里的饼大小不一，我习惯在到达顾客饭桌前，把一摞饼按照大小次序摆好——小的在上面，大的在下面。由于我一只手托着盘子，只好用另一只手，一次抓住最上面的几块饼，把它们上下颠倒个个儿，反复几次之后，这摞烙饼就排好序了。

我后来想，这实际上是个有趣的排序问题：假设有 *n* 块大小不一的烙饼，那最少要翻几次，才能达到最后大小有序的结果呢？

你能否写出一个程序，对于 *n* 块大小不一的烙饼，输出最优化的翻饼过程呢？

分析与解法

这个排序问题非常有意思，首先我们要弄清楚解决问题的关键操作——“单手每次抓几块饼，全部颠倒”。

具体参看图 1-6：

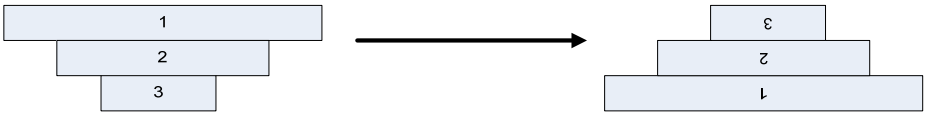

图 1-6　烙饼的翻转过程

每次我们只能选择最上方的一堆饼，一起翻转。而不能一张张地直接抽出来，然后进行插入，也不能交换任意两块饼子。这说明基本的排序办法都不太好用。那么怎么把这 *n* 个烙饼排好序呢？

由于每次操作都是针对最上面的饼，如果最底层的饼已经排序，那我们只用处理上面的 *n*−1 个烙饼。这样，我们可以再简化为 *n*−2、*n*−3，直到最上面的两个饼排好序。

【解法一】

我们用图 1-7 演示一下，为了把最大的烙饼摆在最下面，我们先把最上面的烙饼和最大的烙饼之间的烙饼翻转（1~4 之间），这样，最大的烙饼就在最上面了。接着，我们把所有烙饼翻转（4~5 之间），最大的烙饼就摆在最下面了。

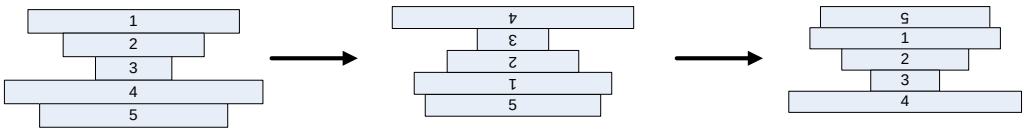

图 1-7　两次翻转烙饼，调整最大的烙饼到最底端

之后，我们对上面 $n-1$、$n-2$ 个饼重复这个过程就可以了。

那么，我们一共需要多少次翻转才能把这些烙饼给翻转过来呢？

首先，经过两次翻转可以把最大的烙饼翻转到最下面。因此，最多需要把上面的 $n-1$ 个烙饼依次翻转两次。那么，我们至多需要 $2(n-1)$ 次翻转就可以把所有烙饼排好序（因为第二小的烙饼排好的时候，最小的烙饼已经在最上面了）。

这样看来，单手翻转的想法是肯定可以实现的。我们进一步想想怎么减少翻转烙饼的次数吧。

怎样才能通过程序来搜索到一个最优的方案呢？

首先，通过每次找出最大的烙饼进行翻转是一个可行的解决方案。那么，这个方案是最好的一个吗？考虑这样一种情况，假如这堆烙饼中有好几个不同的部分相对有序，凭直觉来猜想，我们可以先把小一些的烙饼进行翻转，让其有序。这样会比每次翻转最大的烙饼要更快。

既然如此，有类似的方案可以达到目的吗？比如说，考虑每次翻转的时候，把两个本来应该相邻在烙饼尽可能地换到一起。这样，当所有的烙饼都换到一起之后，实际上就是完成排序了。（从这个意义上来说，每次翻最大烙饼的方案实质上就是每次把最大的和次大的交换到一起。）

在这样的基础之上，本能的一个想法就是穷举。只要穷举出所有可能的交换方案，那么，我们一定能够找到一个最优的方案。

沿着这个思路去考虑，我们自然就会使用动态规划或者递归的方法来进行实现了。可以从不同的翻转策略开始，比如说第一次先翻最小的，然后递归把所有的可能全部翻转一遍。这样，最终肯定是可以找到一个解的。

但是，既然是递归就一定有退出的条件。在这个过程中，第一个退出的条件肯定是所有的烙饼已经排好序。那么，有其他的吗？如果大家仔细想想就会发现到，既然 $2(n-1)$ 是一个最多的翻转次数。如果在算法中，需要翻转的次数多于 $2(n-1)$，那么，我们就应该放弃这个翻转算法，直接退出。这样，就能够减少翻转的次数。

从另外一个层面上来说，既然这是一个排序问题。我们也应该利用到排序的信息来进行处理。同样，在翻转的过程中，我们可以看看当前的烙饼数组的排序情况如何，然后利用这些信息来帮助减少翻转次数的判断过程。

下面是在前面讨论的基础之上形成的一个粗略的搜索最优方案的程序：

代码清单 1-8

```
#include <stdio.h>
/*********************************************************************/
//
// 烙饼排序实现
//
/*********************************************************************/
class CPrefixSorting
{
public:

    CPrefixSorting()
    {
        m_nCakeCnt = 0;
        m_nMaxSwap = 0;
    }

    //
    // 计算烙饼翻转信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Run(int* pCakeArray, int nCakeCnt)
    {
        Init(pCakeArray, nCakeCnt);

        m_nSearch = 0;
        Search(0);
    }

    //
    // 输出烙饼具体翻转的次数
    //
void Output()
{
    for(int i = 0; i < m_nMaxSwap; i++)
    {
        printf("%d ", m_arrSwap[i]);
    }

    printf("\n |Search Times| : %d\n", m_nSearch);
    printf("Total Swap times = %d\n", m_nMaxSwap);
}

private:

    //
    // 初始化数组信息
    // @param
    // pCakeArray   存储烙饼索引数组
    // nCakeCnt 烙饼个数
    //
    void Init(int* pCakeArray, int nCakeCnt)
    {
        Assert(pCakeArray != NULL);
        Assert(nCakeCnt > 0);

        m_nCakeCnt = n;
```

```
// 初始化烙饼数组
    m_CakeArray = new int[m_nCakeCnt];
    Assert(m_CakeArray != NULL);
    for(int i = 0; i < m_nCakeCnt; i++)
    {
        m_CakeArray[i] = pCakeArray[i];
    }

    // 设置最多交换次数信息
    m_nMaxSwap = UpBound(m_nCakeCnt);

    // 初始化交换结果数组
    m_SwapArray = new int[m_nMaxSwap];
    Assert(m_SwapArray != NULL);

    // 初始化中间交换结果信息
    m_ReverseCakeArray = new int[m_nCakeCnt];
    for(i = 0; i < m_nCakeCnt; i++)
    {
        m_ReverseCakeArray[i] = m_CakeArray[i];
    }
    m_ReverseCakeArraySwap = new int[m_nMaxSwap];
}


//
// 寻找当前翻转的上界
//
//
int UpBound(int nCakeCnt)
{
    return nCakeCnt*2;
}

//
// 寻找当前翻转的下界
//
//
int LowerBound(int* pCakeArray, int nCakeCnt)
{
    int t, ret = 0;

    // 根据当前数组的排序信息情况来判断最少需要交换多少次
    for(int i = 1; i < nCakeCnt; i++)
    {
        // 判断位置相邻的两个烙饼，是否为尺寸排序上相邻的
        t = pCakeArray[i] - pCakeArray[i-1];
        if((t == 1) || (t == -1))
        {
        }
        else
        {
            ret++;
        }
    }
    return ret;
}

// 排序的主函数
```

```
void Search(int step)
{
    int i, nEstimate;

    m_nSearch++;

    // 估算这次搜索所需要的最小交换次数
    nEstimate = LowerBound(m_ReverseCakeArray, m_nCakeCnt);
    if(step + nEstimate > m_nMaxSwap)
        return;

    // 如果已经排好序，即翻转完成，输出结果
    if(IsSorted(m_ReverseCakeArray, m_nCakeCnt))
    {
        if(step < m_nMaxSwap)
        {
            m_nMaxSwap = step;
            for(i = 0; i < m_nMaxSwap; i++)
                m_arrSwap[i] = m_ReverseCakeArraySwap[i];
        }
        return;
    }

    // 递归进行翻转
    for(i = 1; i < m_nCakeCnt; i++)
    {
        Revert(0, i);
        m_ReverseCakeArraySwap[step] = i;
        Search(step + 1);
        Revert(0, i);
    }
}

//
// true : 已经排好序
// false : 未排序
//
bool IsSorted(int* pCakeArray, int nCakeCnt)
{
    for(int i = 1; i < nCakeCnt; i++)
    {
        if(pCakeArray[i-1] > pCakeArray[i])
        {
            return false;
        }
    }
    return true;
}

//
// 翻转烙饼信息
//
void Revert(int nBegin, int nEnd)
{
Assert(nEnd > nBegin);
    int i, j, t;

    // 翻转烙饼信息
    for(i = nBegin, j = nEnd; i < j; i++, j--)
    {
```

```
            t = m_ReverseCakeArray[i];
            m_ReverseCakeArray[i] = m_ReverseCakeArray[j];
            m_ReverseCakeArray[j] = t;
        }
    }

private:

    int* m_CakeArray;   // 烙饼信息数组
    int m_nCakeCnt;     // 烙饼个数
    int m_nMaxSwap;     // 最多交换次数。根据前面的推断，这里最多为m_nCakeCnt * 2
    int* m_SwapArray;   // 交换结果数组

    int* m_ReverseCakeArray;        // 当前翻转烙饼信息数组
    int* m_ReverseCakeArraySwap;    // 当前翻转烙饼交换结果数组
    int m_nSearch;                  // 当前搜索次数信息
};
```

当烙饼不多的时候，我们已经可以很快地找出最优的翻转方案。

我们还有办法来对这个程序进行优化，使得能更快地找出最优方案吗？

我们已经知道怎么构造一个可行的翻转方案，所以最优的方案肯定不会比这个差。这个就是我们程序中的上界（UpperBound），就是说，我们感兴趣的最优方案最差也就是我们刚构造出来的方案了。如果我们能够找到一个更好的构造方案，我们的搜索空间就会继续缩小，因为我们一开始就设 m_nMinSwap 为 UpperBound，而程序中有一个剪枝：

```
nEstimate = LowerBound(m_tArr, m_n);
if(step + nEstimate > m_nMinSwap)
    return;
```

m_nMinSwap 越小，那么这个剪枝条件就越容易满足，更多的情况就不需要再去搜索。当然，程序也就能更快地找出最优方案。

仔细分析上面的剪枝条件，在到达 m_tArr 状态之前，我们已经翻转了 step 次，nEstimate 是在当前这个状态我们至少还要翻转多少次才能成功的次数。如果 step+nEstimate 大于 m_nMinSwap，也就说明从当前状态继续下去，m_nMinSwap 次我们也不能排好所有烙饼。那么，当然就没有必要再继续了。因为继续下去得到的方案不可能比我们已经找到的好。

显然，如果 nEstimate 越大，剪枝条件越容易被满足。而这正是我们希望的。

结合上面两点，我们希望 UpperBound 越小越好，而下界（LowerBound）越大越好。假设如果有神仙指点，你只要告诉神仙你当前的状态，他就能告诉你最少需要多少次翻转。这样的话，我们可以花费 $O(N^2)$ 的时间得到最优的方案。但是，现实中，没有这样的神仙。我们只能尽可能地减小 UpperBound，增加 LowerBound，从而减少需要搜索的空间。

利用上面的程序，做一个简单的比较。

对于一个输入，10 个烙饼，从上到下，烙饼半径分别为 3, 2, 1, 6, 5, 4, 9, 8, 7, 0。对应上面程序的输入为：

```
10
3 2 1 6 5 4 9 8 7 0
```

如果 LowerBound 在任何状态都为 0，也就是我们太懒了，不想考虑那么多。当然任意状态下，你至少需要 0 次翻转才能排好序。这样，上面的程序 Search 函数被调用了 575 225 200 次。

但是如果把 LowerBound 稍微改进一下（如上面程序中所计算的方法估计），程序则只需要调用 172 126 次 Search 函数便可以得到最优方案：

```
6
4 8 6 8 4 9
```

程序中的下界是怎么估计出来的呢？

每一次翻转我们最多使得一个烙饼与大小跟它相邻的烙饼排到一起。如果当前状态 *n* 个烙饼中，有 *m* 对相邻的烙饼它们的半径不相邻，那么我们至少需要 *m* 次才能排好序。

从上面的例子，大家都会发现改进上界和下界，好处可不少。我想不用多说，大家肯定想继续优化上界和下界的估计吧。

除了上界和下界的改进，还有什么办法可以提高搜索效率吗？如果我们翻了若干次之后，又回到一个已经出现过的状态，我们还值得继续从这个状态开始搜索吗？我们怎样去检测一个状态是否出现过呢？

读者也许不相信，比尔盖茨在上大学的时候也研究过这个问题，并且发表过论文。你不妨跟盖茨的结果[1]比比吧。

[1] Gates, W. and Papadimitriou, C. "Bounds for Sorting by Prefix Reversal." Discrete Mathematics. 27, 47~57, 1979. 据说这是 Bill Gates 发表的唯一学术论文。

轻松一刻：看了《编程之美》后面试微软的经历

今天我去面试，前面答得还不错，最后面试官问：看了《编程之美》了么？

我回答：看了。

问：怪不得，书带来了么？

我从书包里拿出皱巴巴的书。

问：为什么书这么皱？你在上面乱画了什么？好像还被水泡过。。。

我想起来面试的时候要诚实，就鼓起勇气说：我有一次做题的时候趴在上面睡着了，然后流了很多口水。。。

面试官想了想，说：比尔开始写程序的时候，也是趴在电脑上睡着了。。。 你明天就来上班吧。

好书推荐《移山之道——VSTS 软件开发指南》

《编程之美——微软技术面试心得》作者之一邹欣 力作。

经典的项目管理之道，讲述在中国的产业背景下，

中小型 IT 企业项目开发和管理的实际案例

不管你用不用 VSTS，这本书都对你很有用！

《移山之道——VSTS 软件开发指南》社区网站 http://yishan.cc/

China-pub 地址：http://www.china-pub.com/35373

《移山之道——VSTS 软件开发指南》读者评论：

全书内容写的很不错 ,以一种别具匠心的风格展示软件生命周期各个阶段管理以及应用。

内容也很简洁没有那么多的理论 ,比较适合要步入 PM 或者其他技术人员学习参考 , 尤其是里面的一些看似很短的对话却体现了一个很深刻的道理。希望作者能多写些这方面的著作。支持原创作品。　　读者： thirston_bill

其实里面讲 VSTS 部分并不比我们公司内部讲师讲的详细 ,但作者说明了为什么要这么用，另外作者让我弄明白了什么是 MSF，也让我有马上将自己项目移至 VSTS 服务器上去的冲动。

目前项目还是处于很松弛的管理管理状态 , 代码管理也不正规 , 而该书还提出了一个在微软敏捷 4 之上的新的方法——移山方法。我就打算用这个管理我自己的项目。　　读者： kiciro

因为在书店中看了一看这书, 感觉很不错, 所以买了它。

当然我买这本书的目的并不是在学习 VSTS 的用法， 因为我觉得在这本书中有很多的技巧与道理， 出于学习这些技巧与道理的角度，我觉得不管你用不用 VSTS 都可以来看看这本书。　　读者：ooyuan

经典的一本项目管理之道： 故事化的情节，实战型的经历，让你不知不觉中了解整个项目开发的过程 , 就当看小说一样 , 只要你能翻到书的最后一页 , 你就受益非浅。　　豆瓣读者