计算数组的小和

【题目】

数组小和的定义如下:

例如数组s=[1,3,5,2,4,6]，在s[0]的左边小于等于s[0]的数的和为0，在s[1]的左边小于等于s[1]的数的和为1，在s[2]的左边小于等于s[2]的数的和为1+3=4，在s[3]的左边小于等于s[3]的数的和为1，在s[4]的左边小于等于s[4]的数的和为1+3+2=6，在s[5]的左边小于等于s[5]的数的和为1+3+5+2+4=15,所以s的小和=0+1+4+1+6+15=27。

给定一个数组s，实现函数返回s的小和。

【难度】

校 ★★★☆

【解答】

时间复杂度O(N^2)的方法比较简单，按照题目例子描述的求小和的方法求解即可，本书不再详述。下面介绍一种时间复杂度O(N*logN)，额外空间复杂度O(N)的方法，这是一种在归并排序的过程中，利用组间在进行合并时产生小和的过程:

1，假设左组为l[]，右组为r[]，左右两个组的组内都已经有序，现在要利用外排序合并成一个大组，并假设当前外排序是l[i]与r[j]在进行比较。

2，如果l[i]<=r[j]，那么产生小和。假设从r[j]往右一直到r[]结束，元素的个数为m，那么产生的小和为l[i]*m。

3，如果l[i]>r[j]，不产生任何小和。

4，整个归并排序的过程该怎么进行就怎么进行，排序过程没有任何变化，只是利用步骤1~3，也就是在组间合并的过程中累加所有产生的小和，总共的累加和就是结果。

还是以题目的例子来说明计算过程:

1，归并排序的过程中会进行拆组再合并的过程。[1,3,5,2,4,6]拆分成左组[1,3,5]和右组[2,4,6]，[1,3,5]再拆分成[1,3]和[5]，[2,4,6]再拆分成[2,4]和[6]，[1,3]再拆分成[1]和[3]。[2,4]再拆分成[2]和[4]，如图1-1。

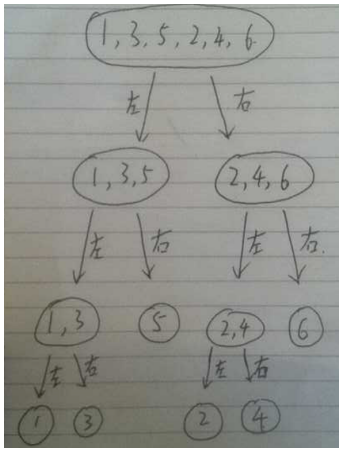

图1-1

2，[1]与[3]合并。1和3比较，左组的数小，右组从3开始到最后一共只有1个数，所以产生小和为1*1=1。合并为[1,3]。

3，[1,3]与[5]合并。1和5比较，左组的数小，右组从5开始到最后一共只有1个数，所以产生小和为1*1=1。3和5比较同理，产生小和为3*1=3。合并为[1,3,5]。

4，[2]与[4]合并。2和4比较，左组的数小，右组从4开始到最后一共只有1个数，所以产生小和为2*1=2。合并为[2,4]。

5，[2,4]与[6]合并。与步骤3同理，产生小和为6，合并为[2,4,6]。

6，[1,3,5]与[2,4,6]合并。1和2比较，左组的数小，右组从2开始到最后一共有3个数，所以

产生小和为1*3=3。3和2比较，右组的数小，不产生小和。3和4比较，左组的数小，右组从4开始到最后一共有2个数，所以产生小和为3*2=6。5和4比较，右组的数小，不产生小和。5和6比较，左组的数小，右组从6开始到最后一共有1个数，所以产生小和为5。合并为[1,2,3,4,5,6]。
7，归并过程结束，总共的小和为1+1+3+2+6+3+6+5=27。合并全部过程如图1-2。

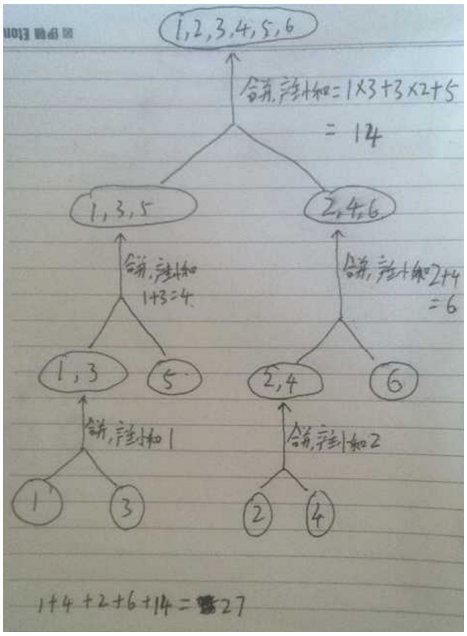

图1-2

在归并排序中，尤其是在组与组之间进行外排合并的过程中，按照如上的方式把小和一点一点榨出来，最后收集到所有的小和。具体过程请参看如下代码中的getSmallSum方法。

```
public int getSmallSum(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        return func(arr, 0, arr.length - 1);
}

public int func(int[] s, int l, int r) {
        if (l == r) {
                return 0;
        }
        int mid = (l + r) / 2;
        return func(s, l, mid) + func(s, mid + 1, r) + merge(s, l, mid, r);
}

public int merge(int[] s, int left, int mid, int right) {
        int[] h = new int[right - left + 1];
        int hi = 0;
        int i = left;
        int j = mid + 1;
        int smallSum = 0;
        while (i <= mid && j <= right) {
                if (s[i] <= s[j]) {
                        smallSum += s[i] * (right - j + 1);
                        h[hi++] = s[i++];
                } else {
                        h[hi++] = s[j++];
                }
```

```
        }
        for (; (j < right + 1) || (i < mid + 1); j++, i++) {
                h[hi++] = i > mid ? s[j] : s[i];
        }
        for (int k = 0; k != h.length; k++) {
                s[left++] = h[k];
        }
        return smallSum;
}
```

数组排序之后相邻数的最大差值

【题目】

给定一个整型数组arr，返回如果排序之后，相邻两数的最大差值。

【举例】

arr=[9,3,1,10]。如果排序，结果为[1,3,9,10]，9和3的差为最大差值，故返回6。

arr=[5,5,5,5]。返回0。

【要求】

如果arr的长度为N，请做到时间复杂度为O(N)。

【难度】

尉 ★★☆☆

【解答】

本题如果用排序来做，时间复杂度是O(N*logN)，而如果利用桶排序的思想(不是直接进行桶排序)，可以做到时间复杂度O(N)，额外空间复杂度O(N)。遍历arr找到最小值和最大值，分别记为min和max。如果arr的长度为N，那么我们准备N+1个桶，把max单独放在第N+1号桶里。arr中在[min,max)范围上的数放在1~N号桶里，对于1~N号桶中的每一个桶来说，负责的区间大小为(max-min)/N。比如长度为10的数组arr中，最小值为10，最大值为110。那么就准备11个桶，arr中等于110的数全部放在第11号桶里。区间[10,20)的数全部放在1号桶里，区间[20,30)的数全部放在2号桶里...，区间[100,110)的数全部放在10号桶里。那么如果一个数为num，它应该分配进(num - min) * len / (max - min)号桶里。

arr一共有N个数，min一定会放进1号桶里，max一定会放进最后的桶里，所以如果把所有的数放入N+1个桶中，必然有桶是空的。如果arr经过排序，相邻的数有可能此时在同一个桶中，也可能在不同的桶中。在同一个桶中的任何两个数的差值都不会大于区间值，而在空桶左右两边不空的桶里，相邻数的差值肯定大于区间值。所以产生最大差值的两个相邻数肯定来自不同的桶。所以只要计算桶之间数的间距就可以了，也就是只用记录每个桶的最大值和最小值，最大差值只可能来自某个非空桶的最小值减去前一个非空桶的最大值。

具体过程请参看如下代码中的maxGap方法。

```
public int maxGap(int[] nums) {
        if (nums == null || nums.length < 2) {
                return 0;
        }
        int len = nums.length;
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < len; i++) {
            min = Math.min(min, nums[i]);
            max = Math.max(max, nums[i]);
        }
        if (min == max) {
            return 0;
        }
        boolean[] hasNum = new boolean[len + 1];
        int[] maxs = new int[len + 1];
        int[] mins = new int[len + 1];
        int bid = 0;
        for (int i = 0; i < len; i++) {
            bid = bucket(nums[i], len, min, max); // 算出桶号
            mins[bid] = hasNum[bid] ? Math.min(mins[bid], nums[i]) : nums[i];
            maxs[bid] = hasNum[bid] ? Math.max(maxs[bid], nums[i]) : nums[i];
            hasNum[bid] = true;
        }
        int res = 0;
        int lastMax = 0;
```

```
        int i = 0;
        while (i <= len) {
            if (hasNum[i++]) { // 找到第一个不空的桶
                lastMax = maxs[i - 1];
                break;
            }
        }
        for (; i <= len; i++) {
            if (hasNum[i]) {
                res = Math.max(res, mins[i] - lastMax);
                lastMax = maxs[i];
            }
        }
        return res;
}

// 使用long类型是为了防止相乘时溢出
public int bucket(long num, long len, long min, long max) {
        return (int) ((num - min) * len / (max - min));
}
```

数组中未出现的最小正整数

【题目】

给定一个无序整型数组arr，找到数组中未出现的最小正整数。

【举例】

arr=[-1,2,3,4]。返回1。

arr=[1,2,3,4]。返回5。

【难度】

尉 ★★☆☆

【解答】

原问题。如果arr长度为N，本题的最优解可以做到时间复杂度O(N)，额外空间复杂度O(1)。
具体过程如下：
1，在遍历arr之前先生成两个变量。变量l表示遍历到目前为止，数组arr已经包含的正整数范围是[1,l]，所以没有开始遍历之前令l=0，表示arr目前没有包含任何正整数。变量r表示遍历到目前为止，在后续出现最优状况的情况下，arr可能包含的正整数范围是[1,r]，所以没有开始遍历之前令r=N，因为还没有开始遍历，所以后续出现的最优状况是arr包含1~N所有的整数。r同时表示arr当前的结束位置。
2，从左到右遍历arr，遍历到位置l，位置l的数为arr[l]。
3，如果arr[l]==l+1。没有遍历arr[l]之前，arr已经包含的正整数范围是[1,l]，此时出现了arr[l]==l+1的情况，所以arr包含的正整数范围可以扩到[1,l+1]，即令l++。然后重复步骤2。
4，如果arr[l]<=l。没有遍历arr[l]之前，arr在后续最优的情况下可能包含的正整数范围是[1,r]，已经包含的正整数范围是[1,l]，所以需要[l+1,r]上的数。而此时出现了arr[l]<=l，说明[l+1,r]范围上的数少了一个，所以arr在后续最优情况下，可能包含的正整数范围缩小了，变为[1,r-1]，此时把arr最后位置的数(arr[r-1])放在位置l上，下一步考察这个数。然后令r--。重复步骤2。
5，如果arr[l]>r，与步骤4同理，把arr最后位置的数(arr[r-1])放在位置l上，下一步考察这个数。然后令r--。重复步骤2。
6，如果arr[arr[l]-1]==arr[l]。如果步骤4和步骤5没中，说明arr[l]是在[l+1,r]范围上的数，而且这个数应该放在arr[l]-1位置上。可是此时发现arr[l]-1位置上的数已经是arr[l]了，说明出现了两个arr[l]，既然在[l+1,r]上出现了重复值，那么[l+1,r]范围上的数又少了一个，所以和步骤4和步骤5一样，把arr最后位置的数(arr[r-1])放在位置l上，下一步考察这个数。然后令r--。重复步骤2。
7，如果步骤4、步骤5和步骤6都没中，说明发现了[l+1,r]范围上的数，并且此时并未发现重复。那么arr[l]应该放到arr[l]-1位置上，所以把l位置上的数和arr[l]-1位置上的数交换，下一步继续遍历l位置上的数。重复步骤2。
8，最终l位置和r位置会碰在一起(l==r)，arr已经包含的正整数范围是[1,l]，返回l+1即可。
具体过程请参看如下代码中的missNum方法。

```
public int missNum(int[] arr) {
    int l = 0;
    int r = arr.length;
    while (l < r) {
        if (arr[l] == l + 1) {
                l++;
        } else if (arr[l] <= l || arr[l] > r || arr[arr[l] - 1] == arr[l]) {
                arr[l] = arr[--r];
        } else {
                swap(arr, l, arr[l] - 1);
        }
    }
    return l + 1;
}
```

矩阵的最小路径和

【题目】

给定一个矩阵m，从左上角开始每次只能向右或者向下走，最后到达右下角位置，路径上所有的数字累加起来就是路径和，返回所有路径中最小的路径和。

【举例】

如果给定的m如下：

1　3　5　9

8　1　3　4

5　0　6　1

8　8　4　0

路径1，3，1，0，6，1，0是所有路径中路径和最小的，所以返回12。

【难度】

尉　★★☆☆

【解答】

经典动态规划方法。假设矩阵m的大小为M*N，行数为M，列数为N。先生成大小和m一样的矩阵dp，dp[i][j]的值表示从左上角即(0,0)位置走到(i,j)位置的最小路径和。对于m的第一行的所有位置来说即(0,j)(0<=j<N)，从(0,0)位置走到(0,j)位置只能向右走，所以(0,0)位置到(0,j)位置的路径和就是m[0][0..j]这些值的累加结果。同理，对于m的第一列的所有位置来说即(i,0)(0<=i<M)，从(0,0)位置走到(i,0)位置只能向下走，所以(0,0)位置到(i,0)位置的路径和就是m[0..i][0]这些值的累加结果。以题目中的例子来说dp第一行和第一列的值如下：

1　4　9　18

9

14

22

除了第一行和第一列的其他位置(i,j)，都有左边位置(i-1,j)和上边位置(i,j-1)。从(0,0)到(i,j)的路径必然经过位置(i-1,j)或位置(i,j-1)，所以dp[i][j]=Min{dp[i-1][j],dp[i][j-1]}+m[i][j]，含义是比较从(0,0)位置开始，经过(i-1,j)位置最终到达(i,j)的最小路径和经过(i,j-1)位置最终到达(i,j)的最小路径之间，哪条路径的路径和更小。那么更小的路径和就是dp[i][j]的值。以题目的例子来说，最终生成的dp矩阵如下：

1　4　9　18

9　5　8　12

14　5　11　12

22　13　15　12

除了第一行和第一列之外，每一个位置都考虑从左边到达自己的路径和更小还是从上边达到自己的路径和更小。最右下角的值就是整个问题的答案。具体过程请参看如下代码中的minPathSum1方法。

```
public int minPathSum1(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
                return 0;
        }
        int row = m.length;
```

```
        int col = m[0].length;
        int[][] dp = new int[row][col];
        dp[0][0] = m[0][0];
        for (int i = 1; i < row; i++) {
            dp[i][0] = dp[i - 1][0] + m[i][0];
        }
        for (int j = 1; j < col; j++) {
            dp[0][j] = dp[0][j - 1] + m[0][j];
        }
        for (int i = 1; i < row; i++) {
            for (int j = 1; j < col; j++) {
                dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + m[i][j];
            }
        }
        return dp[row - 1][col - 1];
    }
```

矩阵中一共有M*N个位置，每个位置都计算一次从(0,0)位置达到自己的最小路径和，计算的时候只是比较一下上边位置的最小路径和和左边位置的最小路径和哪个更小，所以时间复杂度为O(M*N)，dp矩阵的大小为M*N所以额外空间复杂度为O(M*N)。

动态规划经过空间压缩后的方法。这道题的经典动态规划方法在经过空间压缩之后，时间复杂度依然是O(M*N)，但是额外空间复杂度可以从O(M*N)减小至O(min{M,N})，也就是不使用大小为M*N的dp矩阵，而仅仅使用大小为min{M,N}的arr数组。具体过程如下，以题目的例子来举例说明：
1，生成长度为4的数组arr，初始时arr=[0,0,0,0]，我们知道从(0,0)位置到达m中第一行的每个位置，最小路径和就是从(0,0)位置的值开始依次累加的结果，所以依次把arr依次设置为arr=[1,4,9,18]，此时arr[j]的值代表从(0,0)位置达到(0,j)位置的最小路径和。
2，步骤1中arr[j]的值代表从(0,0)位置达到(0,j)位置的最小路径和，在这一步中想把arr[j]的值更新成从(0,0)位置达到(1,j)位置的最小路径和。首先来看arr[0]，更新之前arr[0]的值代表(0,0)位置到达(0,0)位置的最小路径和(dp[0][0])，如果想把arr[0]更新成从(0,0)位置达到(1,0)位置的最小路径和(dp[1][0])，令arr[0]=arr[0]+m[1][0]=9即可。然后来看arr[1]，更新之前arr[1]的值代表(0,0)位置到达(0,1)位置的最小路径和(dp[0][1])，更新之后想让arr[1]代表(0,0)位置到达(1,1)位置的最小路径和(dp[1][1])。根据动态规划的求解过程，到达(1,1)位置有两种选择，一种是从(1,0)位置到达(1,1)位置(dp[1][0]+m[1][1])，另一种是从(0,1)位置到达(1,1)位置(dp[0][1]+m[1][1])，应该选择路径和最小的那个。此时arr[0]的值已经更新成了dp[1][0]，arr[1]目前还没有更新呢，所以arr[1]还是dp[0][1]。所以，arr[1]=Min{arr[0],arr[1]}+m[1][1]=5。更新之后arr[1]的值成了dp[1][1]的值。同理，arr[2]=Min{arr[1],arr[2]}+m[1][2]，...。最终arr可以更新成[9,5,8,12]。
3，重复步骤2的更新过程，一直到arr彻底变成dp矩阵的最后一行。整个过程其实就是不断滚动更新arr数组，让arr依次变成dp矩阵每一行的值，最终变成dp矩阵最后一行的值。
本题的例子是矩阵m的行数等于列数，如果给定的矩阵列数小于行数(N<M)，依然可以用上面的方法令arr更新成dp矩阵每一行的值。但是如果给定的矩阵行数小于列数(M<N)，那么就生成长度为M的arr，然后令arr更新成dp矩阵每一列的值，然后从左向右滚动过去，以本例来说，如果按列来更新，arr首先更新成[1,9,14,22]，然后向右滚动更新成[4,5,5,13]，继续向右滚动更新成[9,8,11,15]，最后是[18,12,12,12]。总之是根据给定矩阵行和列的大小关系决定滚动的方式，始终生成最小长度(Min{M,N})的arr数组。具体过程请参看如下代码中的minPathSum2方法。

```
    public int minPathSum2(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
            return 0;
        }
        int more = Math.max(m.length, m[0].length); // 行数与列数较大的那个为more
        int less = Math.min(m.length, m[0].length); // 行数与列数较小的那个为less
        boolean rowmore = more == m.length; // 行数是不是大于等于列数
        int[] arr = new int[less]; // 辅助数组的长度仅为行数与列数中的最小值
        arr[0] = m[0][0];
        for (int i = 1; i < less; i++) {
            arr[i] = arr[i - 1] + (rowmore ? m[0][i] : m[i][0]);
```

```
                }
                for (int i = 1; i < more; i++) {
                        arr[0] = arr[0] + (rowmore ? m[i][0] : m[0][i]);
                        for (int j = 1; j < less; j++) {
                                arr[j] = Math.min(arr[j - 1], arr[j])
                                                + (rowmore ? m[i][j] : m[j][i]);
                        }
                }
                return arr[less - 1];
        }
```

【扩展】

本题压缩空间的方法几乎可以应用到所有需要二维动态规划表的面试题目，通过一个数组滚动更新的方式无疑节省了大量的空间。没有优化之前，取得某个位置动态规划值的过程是在矩阵中进行两次寻址，优化后这一过程只需要一次寻址，程序的常数时间也得到了一定程度的加速。但是空间压缩的方法是有局限性的，本题如果改成“打印具有最小路径和的路径”，那么就不能使用空间压缩的方法。如果类似本题目这种需要二维表的动态规划题目，最终目的是想求最优解的具体路径，往往需要完整的动态规划表，但如果只是想求最优解的值，则可以使用空间压缩的方法。因为空间压缩的方法是滚动更新的，会覆盖掉之前求解的值，让求解轨迹变的不可回溯。希望读者好好研究这种空间压缩的实现技巧，本书还有许多动态规划题目会涉及空间压缩方法的实现。