第四章 查找匹配

# 4.1有序数组的查找

**题目描述**

给定一个有序的数组，查找某个数是否在数组中，请编程实现。

**分析与解法**

一看到数组本身已经有序，我想你可能反应出了要用二分查找，毕竟二分查找的适用条件就是有序的。那什么是二分查找呢？

二分查找可以解决（预排序数组的查找）问题：只要数组中包含T（即要查找的值），那么通过不断缩小包含T的范围，最终就可以找到它。其算法流程如下：

* 一开始，范围覆盖整个数组。
* 将数组的中间项与T进行比较，如果T比数组的中间项要小，则到数组的前半部分继续查找，反之，则到数组的后半部分继续查找。
* 如此，每次查找可以排除一半元素，范围缩小一半。就这样反复比较，反复缩小范围，最终就会在数组中找到T，或者确定原以为T所在的范围实际为空。

对于包含N个元素的表，整个查找过程大约要经过log(2)N次比较。

此时，可能有不少读者心里嘀咕，不就二分查找么，太简单了。

然《编程珠玑》的作者Jon Bentley曾在贝尔实验室做过一个实验，即给一些专业的程序员几个小时的时间，用任何一种语言编写二分查找程序（写出高级伪代码也可以），结果参与编写的一百多人中：90%的程序员写的程序中有bug（我并不认为没有bug的代码就正确）。

也就是说：在足够的时间内，只有大约10%的专业程序员可以把这个小程序写对。但写不对这个小程序的还不止这些人：而且高德纳在《计算机程序设计的艺术 第3卷 排序和查找》第6.2.1节的“历史与参考文献”部分指出，虽然早在1946年就有人将二分查找的方法公诸于世，但直到1962年才有人写出没有bug的二分查找程序。

你能正确无误的写出二分查找代码么？不妨一试，关闭所有网页，窗口，打开记事本，或者编辑器，或者直接在本文评论下，不参考上面我写的或其他任何人的程序，给自己十分钟到N个小时不等的时间，立即编写一个二分查找程序。

要准确实现二分查找，首先要把握下面几个要点：

* 关于right的赋值
  + right = n-1 => while(left <= right) => right = middle-1;
  + right = n => while(left < right) => right = middle;
* middle的计算不能写在while循环外，否则无法得到更新。

以下是一份参考实现：

int BinarySearch(int array[], int n, int value)

{

int left = 0;

int right = n - 1;

//如果这里是int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：

//1、下面循环的条件则是while(left < right)

//2、循环内当 array[middle] > value 的时候，right = mid

while (left <= right) //循环条件，适时而变

{

int middle = left + ((right - left) >> 1); //防止溢出，移位也更高效。同时，每次循环都需要更新。

if (array[middle] > value)

{

right = middle - 1; //right赋值，适时而变

}

else if(array[middle] < value)

{

left = middle + 1;

}

else

return middle;

//可能会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的情况更多

//如果每次循环都判断一下是否相等，将耗费时间

}

return -1;

}

**总结**

编写二分查找的程序时

* 如果令 `left <= right，则right = middle - 1;
* 如果令left < right，则 right = middle;`

换言之，算法所操作的区间,是左闭右开区间,还是左闭右闭区间,这个区间,需要在循环初始化。且在循环体是否终止的判断中,以及每次修改left, right区间值这三个地方保持一致,否则就可能出错。

# 4.2行列递增矩阵的查找

## 题目描述

在一个m行n列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

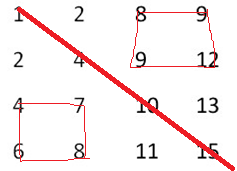
例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字6，则返回true；如果查找数字5，由于数组不含有该数字，则返回false。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/23~24/23.1.gif)

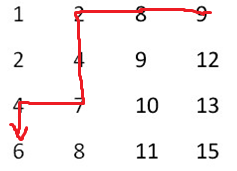
## 分析与解法

### 解法一、分治法

这种行和列分别递增的矩阵，有一个专有名词叫做杨氏矩阵，由剑桥大学数学家杨表在1900年推提出，在这个矩阵中的查找，俗称杨氏矩阵查找。

以查找数字6为例，因为矩阵的行和列都是递增的，所以整个矩阵的对角线上的数字也是递增的，故我们可以在对角线上进行二分查找，如果要找的数是6介于对角线上相邻的两个数4、10，可以排除掉左上和右下的两个矩形，而在左下和右上的两个矩形继续递归查找，如下图所示：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/23~24/23.2.gif)

### 解法二、定位法

首先直接定位到最右上角的元素，再配以二分查找，比要找的数（6）大就往左走，比要找数（6）的小就往下走，直到找到要找的数字（6）为止，这个方法的时间复杂度O（m+n）。如下图所示：  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/23~24/23.3.gif)

关键代码如下所示：

#define ROW 4

#define COL 4

bool YoungMatrix(int array[][COL], int searchKey){

int i = 0, j = COL - 1;

int var = array[i][j];

while (true){

if (var == searchKey)

return true;

else if (var < searchKey && i < ROW - 1)

var = array[++i][j];

else if (var > searchKey && j > 0)

var = array[i][--j];

else

return false;

}

}

## 举一反三

1、给定 n×n 的实数矩阵，每行和每列都是递增的，求这 n^2 个数的中位数。

2、我们已经知道杨氏矩阵的每行的元素从左到右单调递增，每列的元素从上到下也单调递增的矩阵。那么，如果给定从1-n这n个数，我们可以构成多少个杨氏矩阵呢？

例如n = 4的时候，我们可以构成1行4列的矩阵：

1 2 3 4

2个2行2列的矩阵:

1 2

3 4

和

1 3

2 4

还有一个4行1列的矩阵

1

2

3

4

因此输出4。

# 4.3出现次数超过一半的数字

## 题目描述

题目：数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字。

## 分析与解法

一个数组中有很多数，现在我们要找出其中那个出现次数超过总数一半的数字，怎么找呢？大凡当我们碰到某一个杂乱无序的东西时，我们人的内心本质期望是希望把它梳理成有序的。所以，我们得分两种情况来讨论，无序和有序。

### 解法一

如果**无序**，那么我们是不是可以先把数组中所有这些数字**先进行排序**（至于排序方法可选取最常用的快速排序）。排完序后，直接遍历，在遍历整个数组的同时统计每个数字的出现次数，然后把那个出现次数超过一半的数字直接输出，题目便解答完成了。总的时间复杂度为O(nlogn + n)。

但**如果是有序的数组呢**，或者经过排序把无序的数组变成有序后的数组呢？是否在排完序O(nlogn)后，还需要再遍历一次整个数组？

我们知道，既然是数组的话，那么我们可以根据数组索引支持直接定向到某一个数。我们发现，一个数字在数组中的出现次数超过了一半，那么在已排好序的数组索引的N/2处（从零开始编号），就一定是这个数字。自此，我们只需要对整个数组排完序之后，然后直接输出数组中的第N/2处的数字即可，这个数字即是整个数组中出现次数超过一半的数字，总的时间复杂度由于少了最后一次整个数组的遍历，缩小到O(n\*logn)。

然时间复杂度并无本质性的改变，我们需要找到一种更为有效的思路或方法。

### 解法二

既要缩小总的时间复杂度，那么可以用查找时间复杂度为O(1)的**hash表**，即以空间换时间。哈希表的键值（Key）为数组中的数字，值（Value）为该数字对应的次数。然后直接遍历整个**hash表**，找出每一个数字在对应的位置处出现的次数，输出那个出现次数超过一半的数字即可。

### 解法三

Hash表需要O(n)的空间开销，且要设计hash函数，还有没有更好的办法呢？我们可以试着这么考虑，如果**每次删除两个不同的数**（不管是不是我们要查找的那个出现次数超过一半的数字），那么，在剩下的数中，我们要查找的数（出现次数超过一半）出现的次数仍然超过总数的一半。通过不断重复这个过程，不断排除掉其它的数，最终找到那个出现次数超过一半的数字。这个方法，免去了排序，也避免了空间O(n)的开销，总得说来，时间复杂度只有O(n)，空间复杂度为O(1)，貌似不失为最佳方法。

举个简单的例子，如数组a[5] = {0, 1, 2, 1, 1};

很显然，若我们要找出数组a中出现次数超过一半的数字，这个数字便是1，若根据上述思路4所述的方法来查找，我们应该怎么做呢？通过一次性遍历整个数组，然后每次删除不相同的两个数字，过程如下简单表示：

0 1 2 1 1 =>2 1 1=>1

最终1即为所找。

但是数组如果是{5, 5, 5, 5, 1}，还能运用上述思路么？很明显不能，咱们得另寻良策。

### 解法四

更进一步，考虑到这个问题本身的特殊性，我们可以在遍历数组的时候保存两个值：一个candidate，用来保存数组中遍历到的某个数字；一个nTimes，表示当前数字的出现次数，其中，nTimes初始化为1。当我们遍历到数组中下一个数字的时候：

* 如果下一个数字与之前candidate保存的数字相同，则nTimes加1；
* 如果下一个数字与之前candidate保存的数字不同，则nTimes减1；
* 每当出现次数nTimes变为0后，用candidate保存下一个数字，并把nTimes重新设为1。 直到遍历完数组中的所有数字为止。

举个例子，假定数组为{0, 1, 2, 1, 1}，按照上述思路执行的步骤如下：

* 1.开始时，candidate保存数字0，nTimes初始化为1；
* 2.然后遍历到数字1，与数字0不同，则nTimes减1变为0；
* 3.因为nTimes变为了0，故candidate保存下一个遍历到的数字2，且nTimes被重新设为1；
* 4.继续遍历到第4个数字1，与之前candidate保存的数字2不同，故nTimes减1变为0；
* 5.因nTimes再次被变为了0，故我们让candidate保存下一个遍历到的数字1，且nTimes被重新设为1。最后返回的就是最后一次把nTimes设为1的数字1。

思路清楚了，完整的代码如下：

//a代表数组，length代表数组长度

int FindOneNumber(int\* a, int length)

{

int candidate = a[0];

int nTimes = 1;

for (int i = 1; i < length; i++)

{

if (nTimes == 0)

{

candidate = a[i];

nTimes = 1;

}

else

{

if (candidate == a[i])

nTimes++;

else

nTimes--;

}

}

return candidate;

}

即针对数组{0, 1, 2, 1, 1}，套用上述程序可得：

i=0，candidate=0，nTimes=1；

i=1，a[1] != candidate，nTimes--，=0；

i=2，candidate=2，nTimes=1；

i=3，a[3] != candidate，nTimes--，=0；

i=4，candidate=1，nTimes=1；

如果是0，1，2，1，1，1的话，那么i=5，a[5] == candidate，nTimes++，=2；......

## 举一反三

加强版水王：找出出现次数刚好是一半的数字

分析：我们知道，水王问题：有N个数，其中有一个数出现超过一半，要求在线性时间求出这个数。那么，我的问题是，加强版水王：有N个数，其中有一个数刚好出现一半次数，要求在线性时间内求出这个数。

因为，很明显，如果是刚好出现一半的话，如此例： 0，1，2，1 ：

遍历到0时，candidate为0，times为1

遍历到1时，与candidate不同，times减为0

遍历到2时，times为0，则candidate更新为2，times加1

遍历到1时，与candidate不同，则times减为0；我们需要返回所保存candidate（数字2）的下一个数字，即数字1。

# 第五章 动态规划

# 5.0本章导读

学习一个算法，可分为3个步骤：首先了解算法本身解决什么问题，然后学习它的解决策略，最后了解某些相似算法之间的联系。例如图算法中，

* 广搜是一层一层往外遍历，寻找最短路径，其策略是采取队列的方法。
* 最小生成树是最小代价连接所有点，其策略是贪心，比如Prim的策略是贪心+权重队列。
* Dijkstra是寻找单源最短路径，其策略是贪心+非负权重队列。
* Floyd是多结点对的最短路径，其策略是动态规划。

而贪心和动态规划是有联系的，贪心是“最优子结构+局部最优”，动态规划是“最优独立重叠子结构+全局最优”。一句话理解动态规划，则是枚举所有状态，然后剪枝，寻找最优状态，同时将每一次求解子问题的结果保存在一张“表格”中，以后再遇到重叠的子问题，从表格中保存的状态中查找（俗称记忆化搜索）。

# 5.1最大连续乘积子串

### 题目描述

给一个浮点数序列，取最大乘积连续子串的值，例如 -2.5，4，0，3，0.5，8，-1，则取出的最大乘积连续子串为3，0.5，8。也就是说，上述数组中，3 0.5 8这3个数的乘积30.58=12是最大的，而且是连续的。

### 分析与解法

此最大乘积连续子串与最大乘积子序列不同，请勿混淆，前者子串要求连续，后者子序列不要求连续。也就是说，最长公共子串（Longest CommonSubstring）和最长公共子序列（LongestCommon Subsequence，LCS）是：

* 子串（Substring）是串的一个连续的部分，
* 子序列（Subsequence）则是从不改变序列的顺序，而从序列中去掉任意的元素而获得的新序列；

更简略地说，前者（子串）的字符的位置必须连续，后者（子序列LCS）则不必。比如字符串“ acdfg ”同“ akdfc ”的最长公共子串为“ df ”，而它们的最长公共子序列LCS是“ adf ”，LCS可以使用动态规划法解决。

#### 解法一

或许，读者初看此题，可能立马会想到用最简单粗暴的方式：两个for循环直接轮询。

double maxProductSubstring(double \*a, int length)

{

double maxResult = a[0];

for (int i = 0; i < length; i++)

{

double x = 1;

for (int j = i; j < length; j++)

{

x \*= a[j];

if (x > maxResult)

{

maxResult = x;

}

}

}

return maxResult;

}

但这种蛮力的方法的时间复杂度为O(n^2)，能否想办法降低时间复杂度呢？

#### 解法二

考虑到乘积子序列中有正有负也还可能有0，我们可以把问题简化成这样：数组中找一个子序列，使得它的乘积最大；同时找一个子序列，使得它的乘积最小（负数的情况）。因为虽然我们只要一个最大积，但由于负数的存在，我们同时找这两个乘积做起来反而方便。也就是说，不但记录最大乘积，也要记录最小乘积。

假设数组为a[]，直接利用动态规划来求解，考虑到可能存在负数的情况，我们用maxend来表示以a[i]结尾的最大连续子串的乘积值，用minend表示以a[i]结尾的最小的子串的乘积值，那么状态转移方程为：

maxend = max(max(maxend \* a[i], minend \* a[i]), a[i]);

minend = min(min(maxend \* a[i], minend \* a[i]), a[i]);

初始状态为maxend = minend = a[0]。

参考代码如下：

double MaxProductSubstring(double \*a, int length)

{

double maxEnd = a[0];

double minEnd = a[0];

double maxResult = a[0];

for (int i = 1; i < length; ++i)

{

double end1 = maxEnd \* a[i], end2 = minEnd \* a[i];

maxEnd = max(max(end1, end2), a[i]);

minEnd = min(min(end1, end2), a[i]);

maxResult = max(maxResult, maxEnd);

}

return maxResult;

}

动态规划求解的方法一个for循环搞定，所以时间复杂度为O(n)。

### 举一反三

1、给定一个长度为N的整数数组，只允许用乘法，不能用除法，计算任意（N-1）个数的组合中乘积最大的一组，并写出算法的时间复杂度。  
分析：我们可以把所有可能的（N-1）个数的组合找出来，分别计算它们的乘积，并比较大小。由于总共有N个（N-1）个数的组合，总的时间复杂度为O（N2），显然这不是最好的解法。

# 5.2字符串编辑距离

## 题目描述

给定一个源串和目标串，能够对源串进行如下操作：  
1. 在给定位置上插入一个字符  
2. 替换任意字符  
3. 删除任意字符

写一个程序，返回最小操作数，使得对源串进行这些操作后等于目标串，源串和目标串的长度都小于2000。

### 分析与解法

此题常见的思路是动态规划，假如令dp[i][j] 表示源串S[0…i] 和目标串T[0…j] 的最短编辑距离，其边界：dp[0][j] = j，dp[i][0] = i，那么我们可以得出状态转移方程：

* dp[i][j] =min{
  + dp[i-1][j] + 1 , S[i]不在T[0…j]中
  + dp[i-1][j-1] + 1/0 , S[i]在T[j]
  + dp[i][j-1] + 1 , S[i]在T[0…j-1]中

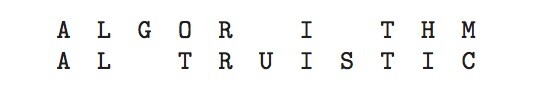
}

接下来，咱们重点解释下上述3个式子的含义

* 关于dp[i-1][j] + 1, s.t. s[i]不在T[0…j]中的说明
  + s[i]没有落在T[0…j]中，即s[i]在中间的某一次编辑操作被删除了。因为删除操作没有前后相关性，不妨将其在第1次操作中删除。除首次操作时删除外，后续编辑操作是将长度为i-1的字符串，编辑成长度为j的字符串：即dp[i-1][j]。
  + 因此：dp[i][j] = dp[i-1][j] + 1。
* 关于dp[i-1][j-1] + 0/1, s.t. s[i] 在T[j]的说明
  + 若s[i]经过编辑，最终落在T[j]的位置。
  + 则要么s[i] == t[j]，s[i]直接落在T[j]。这种情况，编辑操作实际上是将长度为i-1的S’串，编辑成长度为j-1的T’串：即dp[i-1][j-1]；
  + 要么s[i] ≠ t[j]，s[i] 落在T[j]后，要将s[i]修改成T[j]，即在上一种情况的基础上，增加一次修改操作：即dp[i-1][j-1] + 1。
* 关于dp[i][j-1] + 1, s.t. s[i]在T[0…j-1]中的说明
  + 若s[i]落在了T[1…j-1]的某个位置，不妨认为是k，因为最小编辑步数的定义，那么，在k+1到j-1的字符，必然是通过插入新字符完成的。因为共插入了(j-k)个字符，故编辑次数为(j-k)次。而字符串S[1…i]经过编辑，得到了T[1…k]，编辑次数为dp[i][k]。故： dp[i][j] = dp[i][k] + (j-k)。
  + 由于最后的(j-k)次是插入操作，可以讲(j-k)逐次规约到dp[i][k]中。即：dp[i][k]+(j-k)=dp[i][k+1] + (j-k-1)规约到插入操作为1次，得到dp[i][k]+(j-k) =dp[i][k+1] + (j-k-1) =dp[i][k+2] + (j-k-2)=…=dp[i][k+(j-k-1)] + (j-k)-(j-k-1) =dp[i][j-1] + 1。

上述的解释清晰规范，但为啥这样做呢？

换一个角度，其实就是字符串对齐的思路。例如把字符串“ALGORITHM”，变成“ALTRUISTIC”，那么把相关字符各自对齐后，如下图所示：

[](https://camo.githubusercontent.com/675036529d5495952deba4069ec4f73aaff8e6e7/687474703a2f2f696d672e626c6f672e6373646e2e6e65742f3230313430363136313134333234323936)

把图中上面的源串S[0…i] = “ALGORITHM”编辑成下面的目标串T[0…j] = “ALTRUISTIC”，我们枚举字符串S和T最后一个字符s[i]、t[j]对应四种情况：（字符-空白）（空白-字符）(字符-字符)（空白-空白）。

由于其中的（空白-空白）是多余的编辑操作。所以，事实上只存在以下3种情况：

* 下面的目标串空白，即S + 字符X，T + 空白，S变成T，意味着源串要删字符
  + dp[i - 1, j] + 1
* 上面的源串空白，S + 空白，T + 字符，S变成T，最后，在S的最后插入“字符”，意味着源串要添加字符
  + dp[i, j - 1] + 1
* 上面源串中的的字符跟下面目标串中的字符不一样，即S + 字符X，T + 字符Y，S变成T，意味着源串要修改字符
  + dp[i - 1, j - 1] + (s[i] == t[j] ? 0 : 1)

综上，可以写出简单的DP状态方程：

//dp[i,j]表示表示源串S[0…i] 和目标串T[0…j] 的最短编辑距离

dp[i, j] = min { dp[i - 1, j] + 1, dp[i, j - 1] + 1, dp[i - 1, j - 1] + (s[i] == t[j] ? 0 : 1) }

//分别表示：删除1个，添加1个，替换1个（相同就不用替换）。

参考代码如下：

//dp[i][j]表示源串source[0-i)和目标串target[0-j)的编辑距离

int EditDistance(char \*pSource, char \*pTarget)

{

int srcLength = strlen(pSource);

int targetLength = strlen(pTarget);

int i, j;

//边界dp[i][0] = i，dp[0][j] = j

for (i = 1; i <= srcLength; ++i)

{

dp[i][0] = i;

}

for (j = 1; j <= targetLength; ++j)

{

dp[0][j] = j;

}

for (i = 1; i <= srcLength; ++i)

{

for (j = 1; j <= targetLength; ++j)

{

if (pSource[i - 1] == pTarget[j - 1])

{

dp[i][j] = dp[i - 1][j - 1];

}

else

{

dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1]);

}

}

}

return dp[srcLength][targetLength];

}

## 举一反三

1、传统的编辑距离里面有三种操作，即增、删、改，我们现在要讨论的编辑距离只允许两种操作，即增加一个字符、删除一个字符。我们求两个字符串的这种编辑距离，即把一个字符串变成另外一个字符串的最少操作次数。假定每个字符串长度不超过1000，只有大写英文字母组成。

2、有一亿个数，输入一个数，找出与它编辑距离在3以内的数，比如输入6（0110），找出0010等数，数是32位的。

## 问题扩展

实际上，关于这个“编辑距离”问题在搜索引擎中有着重要的作用，如搜索引擎关键字查询中拼写错误的提示，如下图所示，当你输入“[Jult](https://www.google.com.hk/search?hl=zh-CN&newwindow=1&safe=strict&site=&source=hp&q=Jult&btnK=Google+%E6%90%9C%E7%B4%A2)”后，因为没有这个单词“Jult”，所以搜索引擎猜测你可能是输入错误，进而会提示你是不是找“July”：[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/28~29/29.7.jpg)

当然，面试官还可以继续问下去，如请问，如何设计一个比较这篇文章和上一篇文章相似性的算法？

# 5.3格子取数问题

## 题目描述

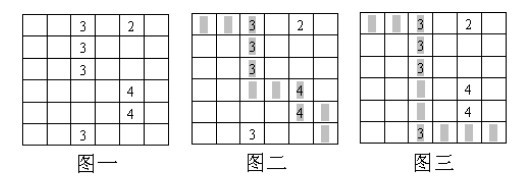
有n\*n个格子，每个格子里有正数或者0，从最左上角往最右下角走，只能向下和向右，一共走两次（即从左上角走到右下角走两趟），把所有经过的格子的数加起来，求最大值SUM，且两次如果经过同一个格子，则最后总和SUM中该格子的计数只加一次。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/34/34.1.jpg)

## 分析与解法

初看到此题，因为要让两次走下来的路径总和最大，读者可能最初想到的思路可能是让每一次的路径都是最优的，即不顾全局，只看局部，让第一次和第二次的路径都是最优。

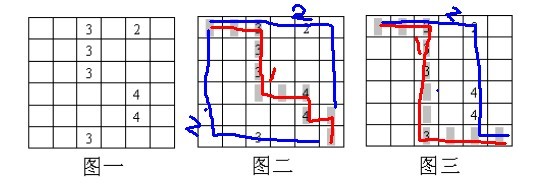
但问题马上就来了，虽然这一算法保证了连续的两次走法都是最优的，但却不能保证总体最优，相应的反例也不难给出，请看下图：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/34/34.2.jpg)

上图中，图一是原始图，那么我们有以下两种走法可供我们选择：

* 如果按照上面的局部贪优走法，那么第一次势必会如图二那样走，导致的结果是第二次要么取到2，要么取到3，
* 但若不按照上面的局部贪优走法，那么第一次可以如图三那样走，从而第二次走的时候能取到2 4 4，很显然，这种走法求得的最终SUM值更大；

为了便于读者理解，我把上面的走法在图二中标记出来，而把应该正确的走法在上图三中标示出来，如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/34/34.3.jpg)

也就是说，上面图二中的走法太追求每一次最优，所以第一次最优，导致第二次将是很差；而图三第一次虽然不是最优，但保证了第二次不差，所以图三的结果优于图二。由此可知不要只顾局部而贪图一时最优，而丧失了全局最优。

局部贪优不行，我们可以考虑穷举，但最终将导致复杂度过高，所以咱们得另寻良策。

为了方便讨论，我们先对矩阵做一个编号，且以5\*5的矩阵为例（给这个矩阵起个名字叫M1）：

M1

0 1 2 3 4

1 2 3 4 5

2 3 4 5 6

3 4 5 6 7

4 5 6 7 8

从左上(0)走到右下(8)共需要走8步（2\*5-2）。我们设所走的步数为s。因为限定了只能向右和向下走，因此无论如何走，经过8步后（s = 8)都将走到右下。而DP的状态也是依据所走的步数来记录的。

再来分析一下经过其他s步后所处的位置，根据上面的讨论，可以知道：

* 经过8步后，一定处于右下角(8)；
* 那么经过5步后(s = 5)，肯定会处于编号为5的位置；
* 3步后肯定处于编号为3的位置；
* s = 4的时候，处于编号为4的位置，此时对于方格中，共有5（相当于n）个不同的位置，也是所有编号中最多的。

故推广来说，对于n\*n的方格，总共需要走2n - 2步，且当s = n - 1时，编号为n个，也是编号数最多的。

如果用DP[s,i,j]来记录2次所走的状态获得的最大值，其中s表示走s步，i和j分别表示在s步后第1趟走的位置和第2趟走的位置。

为了方便描述，再对矩阵做一个编号（给这个矩阵起个名字叫M2）：

M2

0 0 0 0 0

1 1 1 1 1

2 2 2 2 2

3 3 3 3 3

4 4 4 4 4

把之前定的M1矩阵也再贴下：

M1

0 1 2 3 4

1 2 3 4 5

2 3 4 5 6

3 4 5 6 7

4 5 6 7 8我们先看M1，在经过6步后，肯定处于M1中编号为6的位置。而M1中共有3个编号为6的，它们分别对应M2中的2 3 4。故对于M2来说，假设第1次经过6步走到了M2中的2，第2次经过6步走到了M2中的4，DP[s,i,j] 则对应 DP[6,2,4]。由于s = 2n - 2,0 <= i <= j <= n，所以这个DP共有O(n^3)个状态。

M1

0 1 2 3 4

1 2 3 4 5

2 3 4 5 6

3 4 5 6 7

4 5 6 7 8再来分析一下状态转移，以DP[6,2,3]为例(就是上面M1中加粗的部分)，可以到达DP[6,2,3]的状态包括DP[5,1,2]，DP[5,1,3]，DP[5,2,2]，DP[5,2,3]。

下面，我们就来看看这几个状态：DP[5,1,2]，DP[5,1,3]，DP[5,2,2]，DP[5,2,3]，用加粗表示位置DP[5,1,2] DP[5,1,3] DP[5,2,2] DP[5,2,3] （加红表示要达到的状态DP[6,2,3]）

0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4

1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

2 3 4 5 6 2 3 4 5 6 2 3 4 5 6 2 3 4 5 6

3 4 5 6 7 3 4 5 6 7 3 4 5 6 7 3 4 5 6 7

4 5 6 7 8 4 5 6 7 8 4 5 6 7 8 4 5 6 7 8

因此：

DP[6,2,3] = Max(DP[5,1,2] ，DP[5,1,3]，DP[5,2,2]，DP[5,2,3]) + 6,2和6,3格子中对应的数值 （式一）

上面（式一）所示的这个递推看起来没有涉及：“如果两次经过同一个格子，那么该数只加一次的这个条件”，讨论这个条件需要换一个例子，以DP[6,2,2]为例：DP[6,2,2]可以由DP[5,1,1]，DP[5,1,2]，DP[5,2,2]到达，但由于i = j，也就是2次走到同一个格子，那么数值只能加1次。 所以当i = j时，

DP[6,2,2] = Max(DP[5,1,1]，DP[5,1,2]，DP[5,2,2]) + 6,2格子中对应的数值 （式二）

故，综合上述的（式一），（式二）最后的递推式就是

if(i != j) DP[s, i ,j] = Max(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j - 1], DP[s - 1, i, j]) + W[s,i] + W[s,j] else DP[s, i ,j] = Max(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j]) + W[s,i]其中W[s,i]表示经过s步后，处于i位置，位置i对应的方格中的数字。下一节我们将根据上述DP方程编码实现。

为了便于实现，我们认为所有不能达到的状态的得分都是负无穷，参考代码如下：

//copyright@caopengcs 2013

const int N = 202;

const int inf = 1000000000; //无穷大

int dp[N \* 2][N][N];

bool IsValid(int step, int x1, int x2, int n) //判断状态是否合法

{

int y1 = step - x1, y2 = step - x2;

return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n) && (y2 >= 0) && (y2 < n));

}

int GetValue(int step, int x1, int x2, int n) //处理越界 不存在的位置 给负无穷的值

{

return IsValid(step, x1, x2, n) ? dp[step][x1][x2] : (-inf);

}

//状态表示dp[step][i][j] 并且i <= j, 第step步 两个人分别在第i行和第j行的最大得分 时间复杂度O(n^3) 空间复杂度O(n^3)

int MinPathSum(int a[N][N], int n)

{

int P = n \* 2 - 2; //最终的步数

int i, j, step;

//不能到达的位置 设置为负无穷大

for (i = 0; i < n; ++i)

{

for (j = i; j < n; ++j)

{

dp[0][i][j] = -inf;

}

}

dp[0][0][0] = a[0][0];

for (step = 1; step <= P; ++step)

{

for (i = 0; i < n; ++i)

{

for (j = i; j < n; ++j)

{

dp[step][i][j] = -inf;

if (!IsValid(step, i, j, n)) //非法位置

{

continue;

}

//对于合法的位置进行dp

if (i != j)

{

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j - 1, n));

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));

dp[step][i][j] += a[i][step - i] + a[j][step - j]; //不在同一个格子，加两个数

}

else

{

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j - 1, n));

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i - 1, j, n));

dp[step][i][j] = max(dp[step][i][j], GetValue(step - 1, i, j, n));

dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次

}

}

}

}

return dp[P][n - 1][n - 1];

}

复杂度分析：状态转移最多需要统计4个变量的情况，看做是O(1)的，共有O(n^3)个状态，所以总的时间复杂度是O(n^3)的，且dp数组开了N^3大小，故其空间复杂度亦为O(n^3)。

事实上，空间上可以利用滚动数组优化，由于每一步的递推只跟上1步的情况有关，因此可以循环利用数组，将空间复杂度降为O(n^2)。

即我们在推算dp[step]的时候，只依靠它上一次的状态dp[step - 1]，所以dp数组的第一维，我们只开到2就可以了。即step为奇数时，我们用dp[1][i][j]表示状态，step为偶数我们用dp[0][i][j]表示状态，这样我们只需要O(n^2)的空间，这就是滚动数组的方法。滚动数组写起来并不复杂，只需要对上面的代码稍作修改即可，感兴趣的读者可以自己写代码实现下。

## 举一反三

1、给定m\*n的矩阵，每个位置是一个非负整数，从左上角开始，每次只能朝右和下走，走到右下角，但只走一次，求总和最小的路径。

提示：因为只走一次，所以相对来说比较简单，dp[0, 0]=a[0, 0]，且dp[x, y] = min(dp[x-1, y] + a[x, y]dp[x, y-1] + a[x, y])。

2、给定m\*n的矩阵，每个位置是一个整数，从左上角开始，每次只能朝右、上和下走，并且不允许两次进入同一个格子，走到右上角，最小和。

分析：@cpcs ：我们按列dp，假设前一列的最优值已经算好了，一旦往右就回不去了。枚举我们从对固定的(y-1)列，我们已经算好了最优值，我们枚举行x，朝右走到(x,y),然后再从(x,y)朝上走到(x,0)，再从(x,y)朝下走到(x,n-1)，所有这些第y列的值，作为第y列的候选值，取最优。 实际上，我们枚举了进入第y列的位置和在最终停在第y列的位置。这样保证我们不重复经过一个格子，也能保证我们不会往“左”走。

# 5.4交替字符串

**题目描述**

输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序，例如当s1 = “aabcc”，s2 = “dbbca”，s3 = “aadbbcbcac”时，则输出true，但如果s3=“accabdbbca”，则输出false。

**分析与解法**

此题不能简单的排序，因为一旦排序，便改变了s1或s2中各个字符原始的相对顺序，既然不能排序，咱们可以考虑下用动态规划的方法，令dp[i][j]代表s3[0...i+j-1]是否由s1[0...i-1]和s2[0...j-1]的字符组成

* 如果s1当前字符（即s1[i-1]）等于s3当前字符（即s3[i+j-1]），而且dp[i-1][j]为真，那么可以取s1当前字符而忽略s2的情况，dp[i][j]返回真；
* 如果s2当前字符等于s3当前字符，并且dp[i][j-1]为真，那么可以取s2而忽略s1的情况，dp[i][j]返回真，其它情况，dp[i][j]返回假

参考代码如下：

public boolean IsInterleave(String s1, String 2, String 3){

int n = s1.length(), m = s2.length(), s = s3.length();

//如果长度不一致，则s3不可能由s1和s2交错组成

if (n + m != s)

return false;

boolean[][]dp = new boolean[n + 1][m + 1];

//在初始化边界时，我们认为空串可以由空串组成，因此dp[0][0]赋值为true。

dp[0][0] = true;

for (int i = 0; i < n + 1; i++){

for (int j = 0; j < m + 1; j++){

if ( dp[i][j] || (i - 1 >= 0 && dp[i - 1][j] == true &&

//取s1字符

s1.charAT(i - 1) == s3.charAT(i + j - 1)) ||

(j - 1 >= 0 && dp[i][j - 1] == true &&

//取s2字符

s2.charAT(j - 1) == s3.charAT(i + j - 1)) )

dp[i][j] = true;

else

dp[i][j] = false;

}

}

return dp[n][m]

}

理解本题及上段代码，对真正理解动态规划有一定帮助。

# 5.10本章动态规划的习题

##### 1.子序列个数

子序列的定义：对于一个序列a=a[1],a[2],......a[n]，则非空序列a'=a[p1],a[p2]......a[pm]为a的一个子序列 其中1<=p1<p2<.....<pm<=n。 例如：4,14,2,3和14,1,2,3都为4,13,14,1,2,3的子序列。

* 对于给出序列a，有些子序列可能是相同的，这里只算做1个。
* 要求输出a的不同子序列的数量。

##### 2.数塔取数问题

一个高度为N的由正整数组成的三角形，从上走到下，求经过的数字和的最大值。 每次只能走到下一层相邻的数上，例如从第3层的6向下走，只能走到第4层的2或9上。

5

8 4

3 6 9

7 2 9 5

例子中的最优方案是：5 + 8 + 6 + 9 = 28。

##### 3.最长公共子序列

什么是最长公共子序列呢?好比一个数列 S，如果分别是两个或多个已知数列的子序列，且是所有符合此条件序列中最长的，则S 称为已知序列的最长公共子序列。

举个例子，如：有两条随机序列，如 1 3 4 5 5 ，and 2 4 5 5 7 6，则它们的最长公共子序列便是：4 5 5。

提示：最容易想到的算法是穷举搜索法，但考虑到最长公共子序列问题也有最优子结构性质，可以用动态规划解决。

##### 4.最长递增子序列

给定一个长度为N的数组a0,a1,a2...,an-1，找出一个最长的单调递增子序列（注：递增的意思是对于任意的i<j，都满足ai<aj，此外子序列的意思是不要求连续，顺序不乱即可）。例如：给定一个长度为6的数组A{5， 6， 7， 1， 2， 8}，则其最长的单调递增子序列为{5，6，7，8}，长度为4。

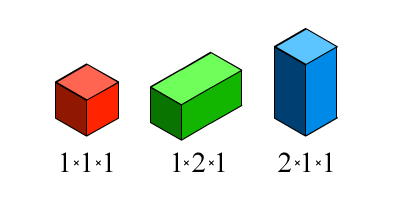
提示：一种解法是转换为最长公共子序列问题，另外一种解法则是动态规划。当我们考虑动态规划解决时，可以定义dp[i]为以ai为末尾的最长递增子序列的长度，故以ai结尾的递增子序列

* 要么是只包含ai的子序列
* 要么是在满足j<i并且aj<ai的以ai为结尾的递增子序列末尾，追加上ai后得到的子序列

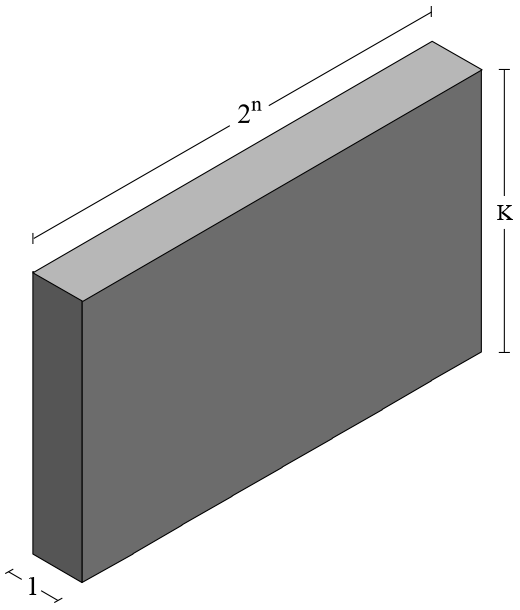
如此，便可建立递推关系，在O(N^2)时间内解决这个问题。

##### 5.木块砌墙

用 1×1×1, 1×2×1以及2×1×1的三种木块（横绿竖蓝，且绿蓝长度均为2），

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/32~33/33.1.png)

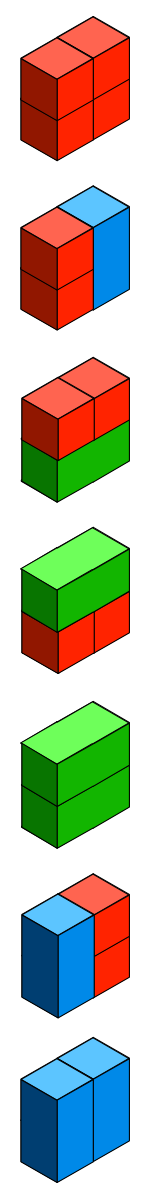
搭建高长宽分别为K × 2^N × 1的墙，不能翻转、旋转（其中，0<=N<=1024，1<=K<=4）

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/32~33/33.2.png)

有多少种方案，输出结果

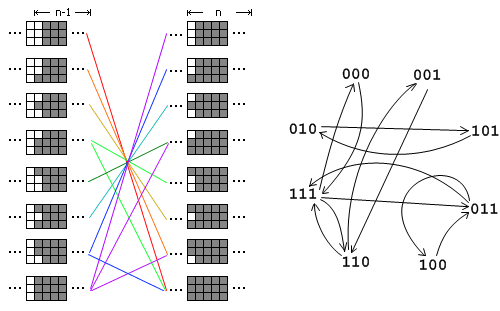
对1000000007取模。

举个例子如给定高度和长度：N=1 K=2，则答案是7，即有7种搭法，如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/32~33/33.3.png)

提示：此题很有意思，涉及的知识点也比较多，包括动态规划，快速矩阵幂，状态压缩，排列组合等等都一一考察了个遍。

而且跟一个比较经典的矩阵乘法问题类似：即用1 x 2的多米诺骨牌填满M x N的矩形有多少种方案，M<=5，N<2^31，输出答案mod p的结果

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/32~33/33.4.gif)