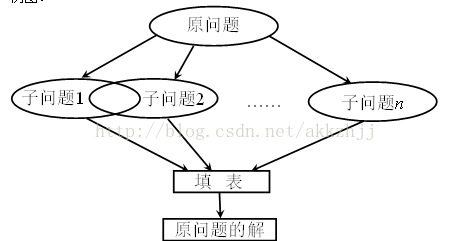
# 有关动态规划的解释

首先,我们看一下官方定义:  
定义:  
动态规划算法是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。  
动态规划算法的基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。  
**基本思想与策略编辑:**  
由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。  
(来自百度百科)  
说实话,没有动态规划的基础很难看懂,但是也能从中看出一些信息,下面我翻译成**人话**:  
**首先是拆分问题**,我的理解就是根据问题的可能性把问题划分成一步一步这样就可以通过递推或者递归来实现.  
关键就是这个步骤,动态规划有一类问题就是从后往前推到,有时候我们很容易知道:**如果只有一种情况时,最佳的选择应该怎么做**.然后根据这个最佳选择往前一步推导,得到前一步的最佳选择  
**然后就是定义问题状态和状态之间的关系**,我的理解是前面拆分的步骤之间的关系,用一种**量化的形式**表现出来,类似于高中学的推导公式,因为这种式子很容易用程序写出来,也可以说对程序比较亲和(也就是最后所说的状态转移方程式)  
**我们再来看定义的下面的两段**,我的理解是比如我们找到最优解,我们应该讲最优解保存下来,为了往前推导时能够使用前一步的最优解,在这个过程中难免有一些相比于最优解差的解,此时我们应该放弃,**只保存最优解**,这样我们每一次都把最优解保存了下来,大大降低了时间复杂度

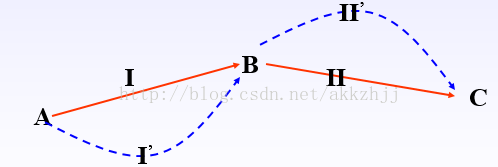
动态规划法和分治法类似，它也是将大问题分解成子问题求解，不同的是，如果分解的子问题有很多是相同的，采用分治法相同的子问题会求解多次，是不是很影响效率；动态规划法呢，它会保存已解决的子问题的答案，再有相同的子问题直接用保存的答案就行了，节省了很多计算时间。



使用动态规划法的情况，对于一个问题具有的性质可以总结为：最优子结构，重叠子问题

**最优性问题**

动态规划法通常用于求解最优性问题，这类问题中，可能会有许多可行解，我们希望找出最优的那个。动态规划法就能找出其中的最优解（最优解可能不止一个）。如下图：



从A到C，经过I和II，肯定是AC的最优解。

例：

0-1背包问题

　　有5个物品，其重量分别是X{2, 2, 6, 5, 4}，价值分别为W{6, 3, 5, 4, 6}，背包的容量为10。求选择装入的物品，使装入的物品总价值最大？

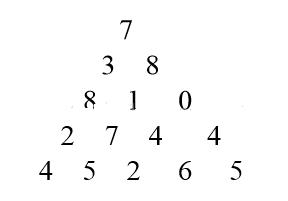
　　根据问题要求，有一个约束条件是总容量必须<=10，需求解决的问题就是在不超过总容量的情况下使总价值最大。

# c语言动态规划——入门篇

动态规划相信大家都知道，动态规划算法也是新手在刚接触算法设计时很苦恼的问题，有时候觉得难以理解，但是真正理解之后，就会觉得动态规划其实并没有想象中那么难。网上也有很多关于讲解动态规划的文章，大多都是叙述概念，讲解原理，让人觉得晦涩难懂，即使一时间看懂了，发现当自己做题的时候又会觉得无所适从。我觉得，理解算法最重要的还是在于练习，只有通过自己练习，才可以更快地提升。

首先，我们看一下这道题（此题目来源于北大POJ）：

## 从问题入手：数字三角形(POJ1163)



在上面的数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字之和最大。路径上的每一步都只能往左下或右下走。只需要求出这个最大和即可，不必给出具体路径。三角形的行数大于1小于等于100，数字为0-99

### 输入与输出：

5//表示三角形的行数接下来输入三角形

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

要求输出最大和

### 解题思路：

首先，肯定得用二维数组来存放数字三角形

然后我们用D(r,j)来表示第r行第j个数字(r,j从1开始算)

我们用MaxSum(r,j)表示从D(r,j)到底边的各条路径中，最佳路径的数字之和。

因此，此题的最终问题就变成了求MaxSum(1,1)

当我们看到这个题目的时候，首先想到的就是可以用简单的递归来解题：

D(r,j)出发，下一步只能走D(r+1,j)或者D(r+1,j+1)。故对于N行的三角形，我们可以写出如下的递归式：if(r==N) MaxSum(r,j)=D(r,j)else MaxSum(r,j)=Max{MaxSum(r＋1,j),MaxSum(r+1,j+1)}+D(r,j)

根据上面这个简单的递归式，我们就可以很轻松地写出完整的递归代码：

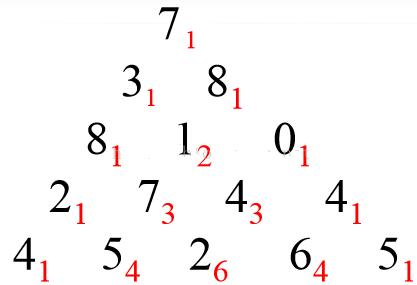
#include<iostream>#include<algorithm>#define MAX 101using namespace std;int D[MAX][MAX];int n;int MaxSum(int i,int j){ if(i==n) return D[i][j]; int x=MaxSum(i+1,j); int y=MaxSum(i+1,j+1); return max(x,y)+D[i][j];}int main(){ int i,j; cin>>n; for(i=1;i<=n;i++) for(j=1;j<=i;j++) cin>>D[i][j]; cout<<MaxSum(1,1)<<endl;}

对于如上这段递归的代码，当我提交到POJ时，会显示如下结果：

教你彻底学会c语言动态规划——入门篇

对的，代码运行超时了，为什么会超时呢？

答案很简单，因为我们重复计算了，当我们在进行递归时，计算机帮我们计算的过程如下图：



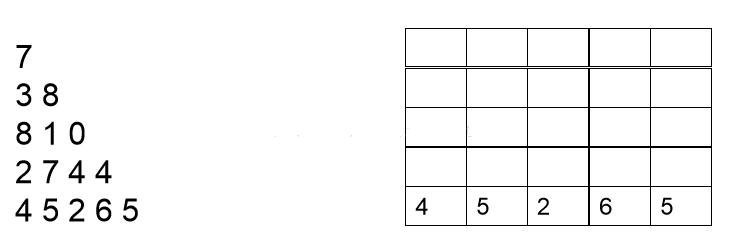
就拿第三行数字1来说，当我们计算从第2行的数字3开始的MaxSum时会计算出从1开始的MaxSum，当我们计算从第二行的数字8开始的MaxSum的时候又会计算一次从1开始的MaxSum，也就是说有重复计算。这样就浪费了大量的时间。也就是说如果采用递规的方法，深度遍历每条路径，存在大量重复计算。则时间复杂度为2的n次方,对于n=100行，肯定超时。

接下来，我们就要考虑如何进行改进，我们自然而然就可以想到如果每算出一个MaxSum(r,j)就保存起来，下次用到其值的时候直接取用，则可免去重复计算。那么可以用n方的时间复杂度完成计算。因为三角形的数字总数是n(n+1)/2

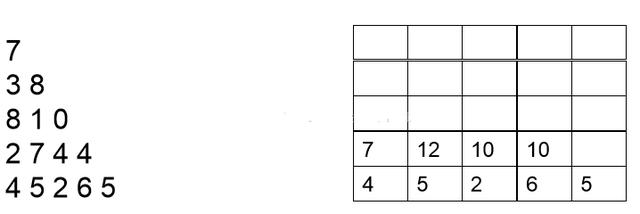
根据这个思路，我们就可以将上面的代码进行改进，使之成为记忆递归型的动态规划程序：#include<iostream>#include<algorithm>using namespace std;#define MAX 101int D[MAX][MAX];int n;int maxSum[MAX][MAX];int MaxSum(int i,int j){ if(maxSum[i][j]!=-1) return maxSum[i][j]; if(i==n) maxSum[i][j]=D[i][j]; else{ int x=MaxSum(i+1,j); int y=MaxSum(i+1,j+1); maxSum[i][j]=max(x,y)+D[i][j]; } return maxSum[i][j];}int main(){ int i,j; cin>>n; for(i=1;i<=n;i++) for(j=1;j<=i;j++){ cin>>D[i][j]; maxSum[i][j]=-1; } cout<<MaxSum(1,1)<<endl;}

当我们提交如上代码时，结果就是一次AC虽然在短时间内就AC了。但是，我们并不能满足于这样的代码，因为递归总是需要使用大量堆栈上的空间，很容易造成栈溢出，我们现在就要考虑如何把递归转换为递推，让我们一步一步来完成这个过程。

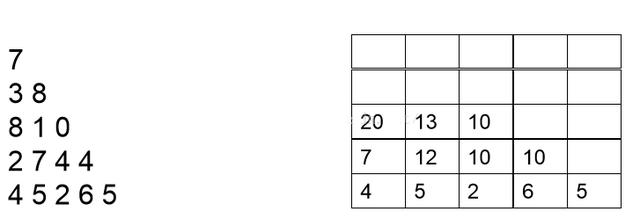
我们首先需要计算的是最后一行，因此可以把最后一行直接写出，如下图：

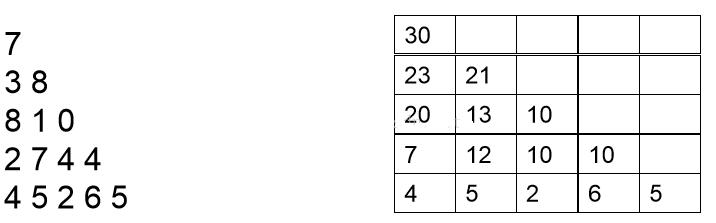


现在开始分析倒数第二行的每一个数，现分析数字2，2可以和最后一行4相加，也可以和最后一行的5相加，但是很显然和5相加要更大一点，结果为7，我们此时就可以将7保存起来，然后分析数字7，7可以和最后一行的5相加，也可以和最后一行的2相加，很显然和5相加更大，结果为12，因此我们将12保存起来。以此类推。。我们可以得到下面这张图：



然后按同样的道理分析倒数第三行和倒数第四行，最后分析第一行，我们可以依次得到如下结果：



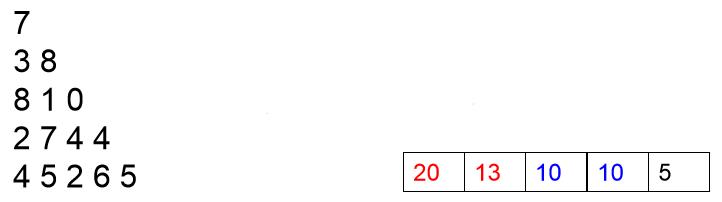
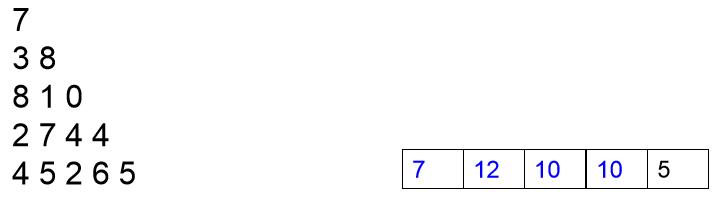
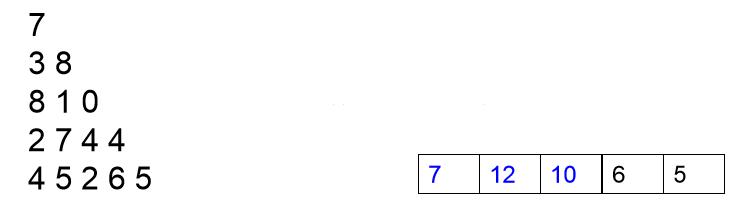
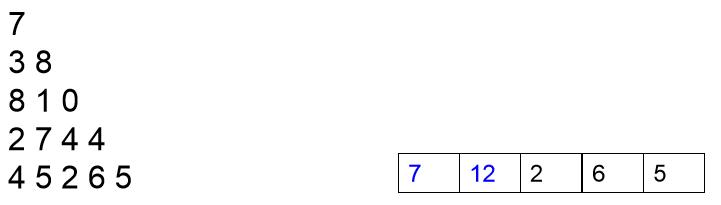
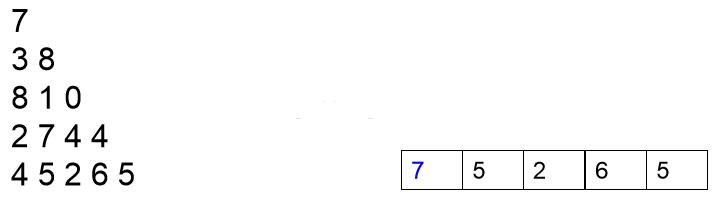


上面的推导过程相信大家不难理解，理解之后我们就可以写出如下的递推型动态规划程序：

#include<iostream>#include<algorithm>using namespace std;#define MAX 101int D[MAX][MAX];int n;int maxSum[MAX][MAX];int main(){ int i,j; cin>>n; for(i=1;i<=n;i++) for(j=1;j<=i;j++) cin>>D[i][j]; for(int i=1;i<=n;++i) maxSum[n][i]=D[n][i]; for(int i=n-1;i>=1;--i) for(int j=1;j<=i;++j) maxSum[i][j]=max(maxSum[i+1][j],maxSum[i+1][j+1])+D[i][j]; cout<<maxSum[1][1]<<endl;}

我们的代码仅仅是这样就够了吗？当然不是，我们仍然可以继续优化，而这个优化当然是对于空间进行优化，其实完全没必要用二维maxSum数组存储每一个MaxSum(r,j),只要从底层一行行向上递推，那么只要一维数组maxSum[100]即可,即只要存储一行的MaxSum值就可以。

对于空间优化后的具体递推过程如下：



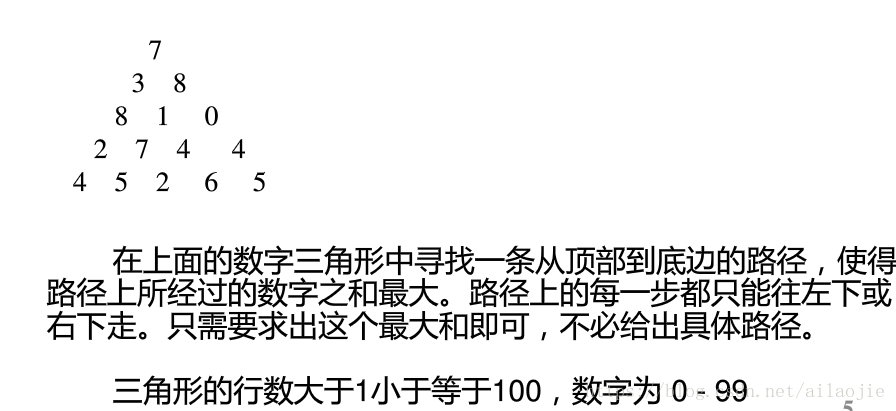
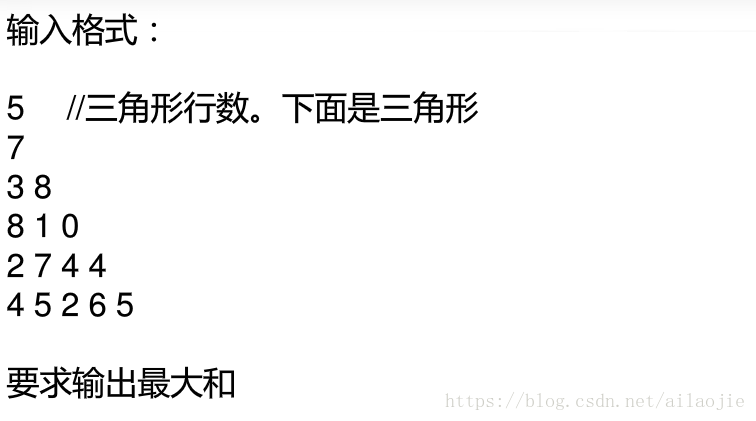
接下里的步骤就按上图的过程一步一步推导就可以了。进一步考虑，我们甚至可以连maxSum数组都可以不要，直接用D的第n行直接替代maxSum即可。但是这里需要强调的是：虽然节省空间，但是时间复杂度还是不变的。

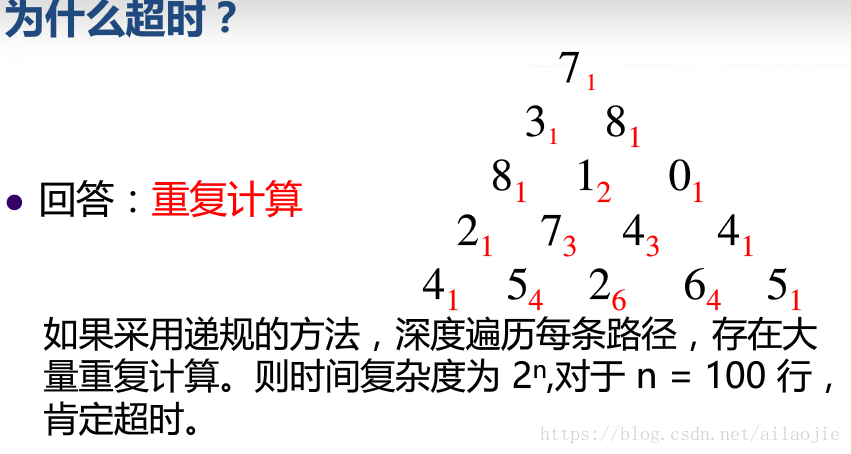
依照上面的方式，我们可以写出如下代码：

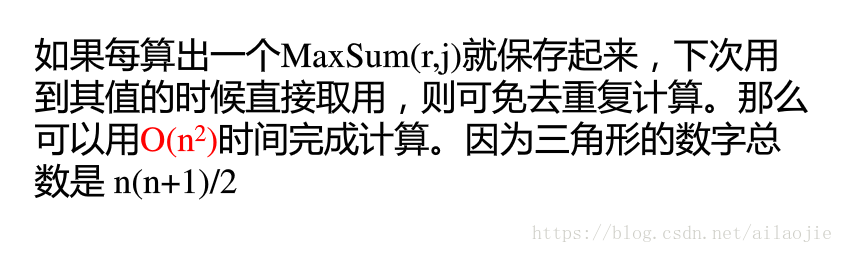
#include<iostream>#include<algorithm>using namespace std;#define MAX 101int D[MAX][MAX];int n;int\*maxSum;int main(){ int i,j; cin>>n; for(i=1;i<=n;i++) for(j=1;j<=i;j++) cin>>D[i][j]; maxSum=D[n];//maxSum指向第n行 for(int i=n-1;i>=1;--i) for(int j=1;j<=i;++j) maxSum[j]=max(maxSum[j],maxSum[j+1])+D[i][j]; cout<<maxSum[1]<<endl;}

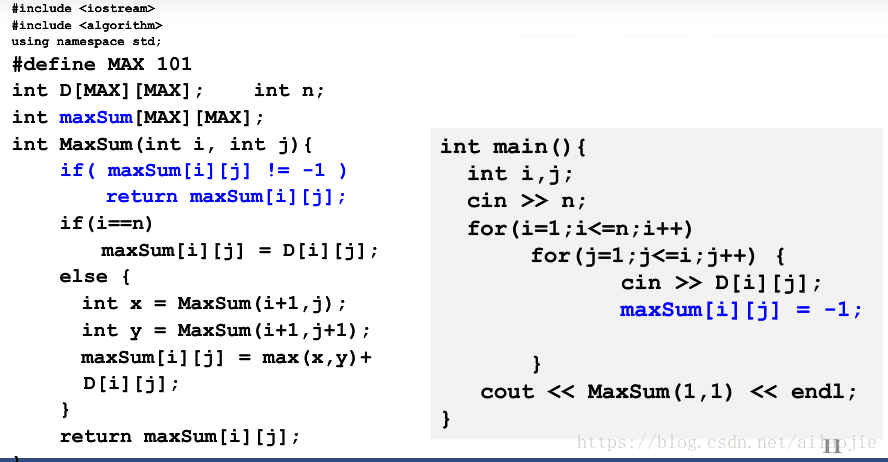
接下来，我们就进行一下总结：

## 经典的数字三角形问题解法

题目:  
  
可以看出每走第n行第m列时有两种后续:**向下或者向右下**  
由于最后一行可以确定,当做边界条件,所以我们自然而然想到递归求解

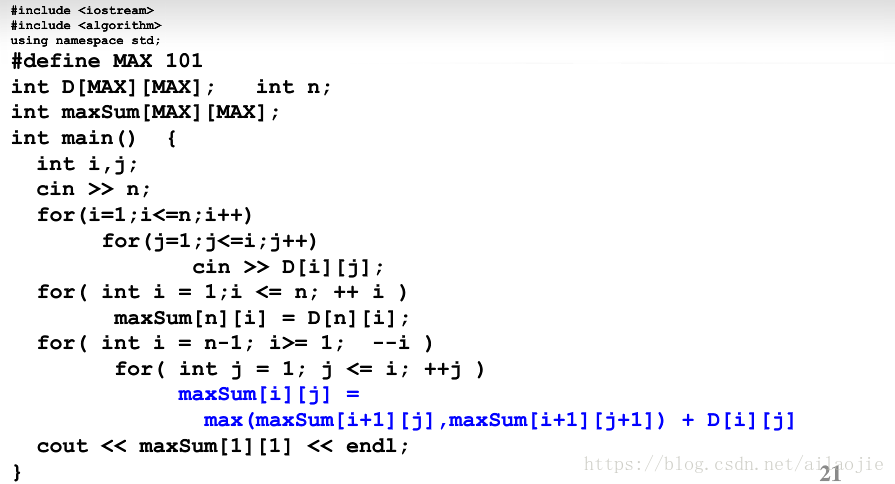
其实仔细观察,上面的解答过程时间复杂度难以想象的大,那是因为他对有的数字的解进行了多次的重复计算,具体如下图:  
  
如果不明白上图,可以把每条路径都画出来,观察每个数字有多少条路径经过了他,就会一目了然  
然后我们就可以自然而然的想到,如果我们每次都把结果保存下来,复杂度就会大大降低

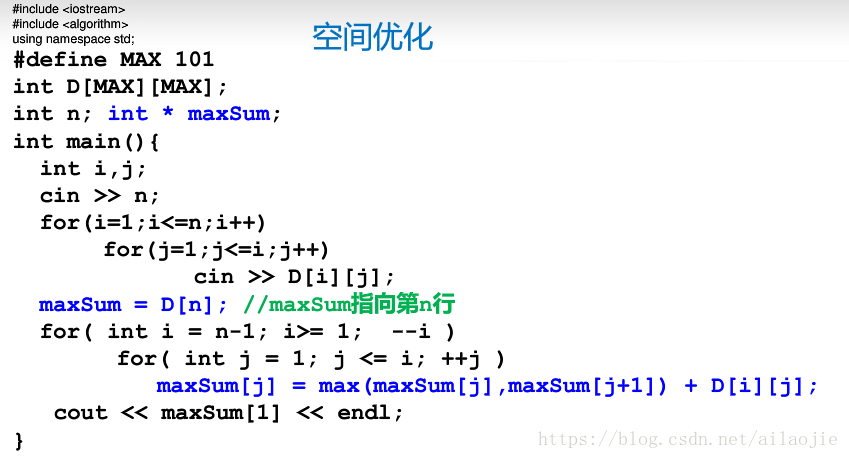




其实答案很简单:

其实这是动态规划很精髓的一部分,是减少复杂度的主要原因  
我们都知道,递归一般情况下是可以转化为递推的,不详细解释了,贴上答案:



其实,仔细观察该解题过程,该过程就是标准的动态规划解题过程,如果把该过程画出来(找到每一步的最优解,其他的舍弃)对动态规划会有更深刻的解法  
还有就是,递推的另一个好处是可以进行空间优化,如图:

下面总结一下动态规划的解题一般思路:  
首先递归应该是我们解决动态规划问题最常用的方法,帅,速度不算太慢  
**那么递归到动规的一般转化方法为:**  
如果该递归函数有n个参数,那么就定义一个n维数组,数组下标是递归函数参数的取值范围(也就是数组每一维的大小).数组元素的值就是递归函数的返回值(初始化为一个标志值,表明还未被填充),这样就可以从边界值开始逐步的填充数组,相当于计算递归函数的逆过程(这和前面所说的推导过程应该是相同的).  
原文链接:https://blog.csdn.net/ailaojie/article/details/83014821

动规解题的一般思路(标准官方,不过经过前边讲解应该就能理解了):

1. **将原问题分解为子问题**(开头已经介绍了怎么分解) (**注意**:1,子问题与原问题形式相同或类似,只是问题规模变小了,从而变简单了; 2,子问题一旦求出就要保存下来,保证每个子问题只求解一遍)
2. **确定状态**(状态:在动规解题中,我们将和子问题相关的各个变量的一组取值,称之为一个"状态",一个状态对应一个或多个子问题所谓的在某个状态的值,这个就是状态所对应的子问题的解,所有状态的集合称为"状态空间".我的理解就是状态就是某个问题某组变量,状态空间就是该问题的所有组变量) **另外:整个问题的时间复杂度就是状态数目乘以每个状态所需要的时间**
3. 确定一些初始状态(边界条件)的值 (这个视情况而定,千万别以为就是最简单的那个子问题解,上面只是例子,真正实践动规千变万化)
4. 确定状态转移方程 (这一步和第三步是最关键的 记住"人人为我"递推,由已知推未知)

适合使用动规求解的问题:  
1,问题具有最优子结构  
2,无后效性 说的花里胡哨的,其实一般遇到求最优解问题一般适合使用动态规划

## 递归到动规的一般转化方法

递归函数有n个参数，就定义一个n维的数组，数组的下标是递归函数参数的取值范围，数组元素的值是递归函数的返回值，这样就可以从边界值开始，逐步填充数组，相当于计算递归函数值的逆过程。

## 动规解题的一般思路

### 1.将原问题分解为子问题

* 把原问题分解为若干个子问题，子问题和原问题形式相同或类似，只不过规模变小了。子问题都解决，原问题即解决(数字三角形例）。
* 子问题的解一旦求出就会被保存，所以每个子问题只需求解一次。

### 2.确定状态

* 在用动态规划解题时，我们往往将和子问题相关的各个变量的一组取值，称之为一个“状态”。一个“状态”对应于一个或多个子问题，所谓某个“状态”下的“值”，就是这个“状态”所对应的子问题的解。
* 所有“状态”的集合，构成问题的“状态空间”。“状态空间”的大小，与用动态规划解决问题的时间复杂度直接相关。在数字三角形的例子里，一共有N×(N+1)/2个数字，所以这个问题的状态空间里一共就有N×(N+1)/2个状态。

整个问题的时间复杂度是状态数目乘以计算每个状态所需时间。在数字三角形里每个“状态”只需要经过一次，且在每个状态上作计算所花的时间都是和N无关的常数。

### 3.确定一些初始状态（边界状态）的值

以“数字三角形”为例，初始状态就是底边数字，值就是底边数字值。

### 4.确定状态转移方程

定义出什么是“状态”，以及在该“状态”下的“值”后，就要找出不同的状态之间如何迁移――即如何从一个或多个“值”已知的“状态”，求出另一个“状态”的“值”(递推型)。状态的迁移可以用递推公式表示，此递推公式也可被称作“状态转移方程”。

数字三角形的状态转移方程:



### 能用动规解决的问题的特点

1)问题具有最优子结构性质。如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质。

2)无后效性。当前的若干个状态值一旦确定，则此后过程的演变就只和这若干个状态的值有关，和之前是采取哪种手段或经过哪条路径演变到当前的这若干个状态，没有关系。

# c语言动态规划——进阶篇

在我的上一篇文章中已经详细讲解了动态规划的原理和如何使用动态规划解题。本篇文章，我将继续通过例子来让大家更加熟练地使用动态规划算法。

话不多说，来看如下例题，也是在动态规划里面遇到过的最频繁的一个题，本题依然来自于北大POJ：

## 从问题入手：最长公共子序列（POJ1458)

给出两个字符串，求出这样的一个最长的公共子序列的长度：子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

### Sample Input

abcfbc abfcab

programming contest

abcd mnp

Sample Output

4

2

0

### 解题思路：

设输入的两个串为s1,s2,设MaxLen(i,j)表示:：s1的左边i个字符形成的子串，与s2左边的j个字符形成的子串的最长公共子序列的长度(i,j从0开始算），则MaxLen(i,j)就是本题的“状态”（如果还不懂“状态”是什么意思，可以参考我上一篇文章）

假定len1=strlen(s1),len2=strlen(s2），那么题目就是要求MaxLen(len1,len2)

显然：

MaxLen(n,0)=0(n=0…len1）

MaxLen(0,n)=0(n=0…len2）

于是，我们可以得到如下的递推公式：

1. if(s1[i-1]==s2[j-1])//s1的最左边字符是s1[0]
2. MaxLen(i,j)=MaxLen(i-1,j-1)+1;
3. else
4. MaxLen(i,j)=Max(MaxLen(i,j-1),MaxLen(i-1,j));

时间复杂度O(mn)m,n是两个字串长度



S1[i-1]!=s2[j-1]时，MaxLen(S1,S2)不会比MaxLen(S1,S2j-1)和MaxLen(S1i-1,S2)两者之中任何一个小，也不会比两者都大。

通过上面的分析，我们很简单就可以写出如下的代码：

1. #include<iostream>
2. #include<cstring>
3. using namespace std;
4. char sz1[1000];
5. char sz2[1000];
6. int maxLen[1000][1000];
7. int main(){
8. while(cin>>sz1>>sz2){
9. int length1=strlen(sz1);
10. int length2=strlen(sz2);
11. int nTmp;
12. int i,j;
13. for(i=0;i<=length1;i++)
14. maxLen[i][0]=0;
15. for(j=0;j<=length2;j++)
16. maxLen[0][j]=0;
17. for(i=1;i<=length1;i++){
18. for(j=1;j<=length2;j++){
19. if(sz1[i-1]==sz2[j-1])
20. maxLen[i][j]=maxLen[i-1][j-1]+1;
21. else
22. maxLen[i][j]=max(maxLen[i][j-1],maxLen[i-1][j]);
23. }
24. }
25. cout<<maxLen[length1][length2]<<endl;
26. }
27. return 0;
28. }

然后提交我们的代码，一次AC。

教你彻底学会c语言动态规划——进阶篇

如果想学c++并想学好，可以加这个群，首先是玖四捌，中间是玖伍四，最后是四捌四，里面有大量的学习资料可以下载。

接下来我们再来看一道典型的例题：

## 从问题入手：最长上升子序列(百练2757)

一个数的序列ai，当a1<a2<...<aS的时候，我们称这个序列是上升的。对于给定的一个序列(a1,a2,...,aN)，我们可以得到一些上升的子序列(ai1,ai2,...,aiK)，这里1<=i1<i2<...<iK<=N。比如，对于序列(1,7,3,5,9,4,8)，有它的一些上升子序列，如(1,7),(3,4,8)等等。这些子序列中最长的长度是4，比如子序列(1,3,5,8).。

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

### 输入数据

输入的第一行是序列的长度N(1<=N<=1000)。第二行给出序列中的N个整数，这些整数的取值范围都在0到10000。

### 输出要求

最长上升子序列的长度。

输入样例

7

1 7 3 5 9 4 8

输出样例

4

### 解题思路

#### 1.找子问题

“求序列的前n个元素的最长上升子序列的长度”是个子问题，但这样分解子问题，不具有“无后效性”，因为假设F(n)=x,但可能有多个序列满足F(n)=x。有的序列的最后一个元素比an+1小，则加上an+1就能形成更长上升子序列；有的序列最后一个元素不比an+1小……以后的事情受如何达到状态n的影响，不符合“无后效性”，因此我们必须换一种思路来解决此问题。

“求以ak（k=1,2,3…N）为终点的最长上升子序列的长度”，一个上升子序列中最右边的那个数，称为该子序列的“终点”。虽然这个子问题和原问题形式上并不完全一样，但是只要这N个子问题都解决了，那么这N个子问题的解中，最大的那个就是整个问题的解。

#### 2.确定状态

子问题只和一个变量——数字的位置相关。因此序列中数的位置k就是“状态”，而状态k对应的“值”，就是以ak做为“终点”的最长上升子序列的长度。状态一共有N个。

#### 3.找出状态转移方程

maxLen(k)表示以ak做为“终点”的

最长上升子序列的长度那么：

初始状态：maxLen(1)=1

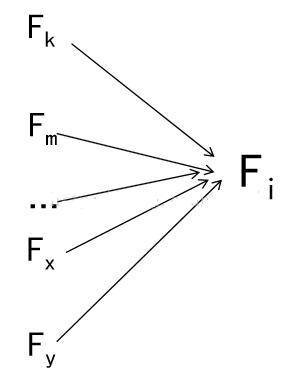
maxLen(k)=max{maxLen(i)：1<=i<k且ai<ak且k≠1}+1若找不到这样的i,则maxLen(k)=1

maxLen(k)的值，就是在ak左边，“终点”数值小于ak，且长度最大的那个上升子序列的长度再加1。因为ak左边任何“终点”小于ak的子序列，加上ak后就能形成一个更长的上升子序列。

有了这个思路，我们就可以很轻松地写出代码了。然而，即使到了这里，我们依然还能从两个方向解决这道题，我们可以将它们分别称为“人人为我”递推型动归和“我为人人”递推型动归。请看下面的讲解：

##### “人人为我”递推型动归

状态i的值Fi由若干个值已知的状态值Fk,Fm,..Fy推出，如求和，取最大值



根据这个方向，我们不难写出如下代码：

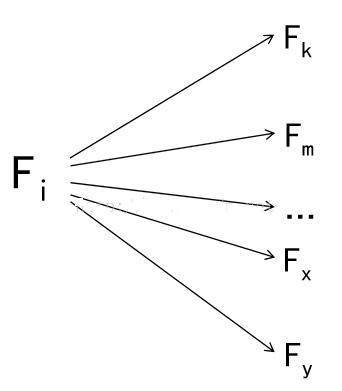
1. #include<iostream>
2. #include<cstring>
3. #include<algorithm>
4. using namespace std;
5. const int MAXN=1010;
6. int a[MAXN];int maxLen[MAXN];
7. int main(){
8. int N;
9. cin>>N;
10. for(int i=1;i<=N;++i){
11. cin>>a[i];
12. maxLen[i]=1;
13. }
14. for(int i=2;i<=N;++i){//每次求以第i个数为终点的最长上升子序列的长度
15. for(int j=1;j<i;++j)//察看以第j个数为终点的最长上升子序列
16. if(a[i]>a[j])
17. maxLen[i]=max(maxLen[i],maxLen[j]+1);
18. }
19. cout<<\*max\_element(maxLen+1,maxLen+N+1);
20. return 0;
21. }//时间复杂度O(N2)

下面是我这段代码的提交结果：



##### “我为人人”递推型动归

状态i的值Fi在被更新（不一定是最终求出）的时候，依据Fi去更新（不一定是最终求出）和状态i相关的其他一些状态的值Fk,Fm,..Fy



根据这个方向，我们又可以写出如下代码：

1. #include<iostream>
2. #include<cstring>
3. #include<algorithm>
4. using namespace std;
5. const int MAXN=1010;
6. int a[MAXN];
7. int maxLen[MAXN];
8. int main(){
9. int N;
10. cin>>N;
11. for(int i=1;i<=N;++i){
12. cin>>a[i];
13. maxLen[i]=1;
14. }
15. for(int i=1;i<=N;++i)
16. for(int j=i+1;j<=N;++j)//看看能更新哪些状态的值
17. if(a[j]>a[i])
18. maxLen[j]=max(maxLen[j],maxLen[i]+1);
19. cout<<\*max\_element(maxLen+1,maxLen+N+1);
20. return 0;
21. }//时间复杂度O(N2)

下面是我这段代码的提交结果：



## 动规的三种形式

#### 1）记忆递归型

优点：只经过有用的状态，没有浪费。递推型会查看一些没用的状态，有浪费。

缺点：可能会因递归层数太深导致栈溢出，函数调用带来额外时间开销。总体来说，比递推型慢。

#### 2)“我为人人”递推型

没有什么明显的优势，有时比较符合思考的习惯。个别特殊题目中会比“人人为我”型节省空间。

#### 3）“人人为我”递推型

在选取最优备选状态的值Fm,Fn,…Fy时，有可能有好的算法或数据结构可以用来显著降低时间复杂度。