**动态规划**

**动态规划**与分治法类似，都是把大问题拆分成小问题，通过寻找大问题与小问题的递推关系，解决一个个小问题，最终达到解决原问题的效果。但不同的是，分治法在子问题和子子问题等上被重复计算了很多次，而动态规划则具有记忆性，通过填写表把所有已经解决的子问题答案纪录下来，在新问题里需要用到的子问题可以直接提取，避免了重复计算，从而节约了时间，所以在问题满足最优性原理之后，用动态规划解决问题的核心就在于填表，表填写完毕，最优解也就找到。

# **[动态规划解决01背包问题](https://www.cnblogs.com/Christal-R/p/Dynamic_programming.html)**

 寻找递推关系式，面对当前商品有两种可能性：

　　　　第一，包的容量比该商品体积小，装不下，此时的价值与前i-1个的价值是一样的，即V(i,j)=V(i-1,j)；

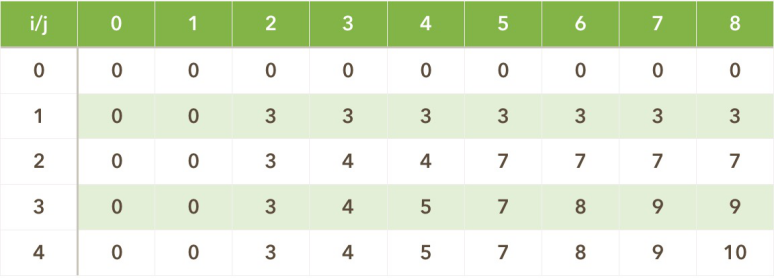
　　　　第二，还有足够的容量可以装该商品，但装了也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即V(i,j)=max｛ V(i-1,j)，V(i-1,j-w(i))+v(i) ｝

　　　　　　　其中V(i-1,j)表示不装，V(i-1,j-w(i))+v(i) 表示装了第i个商品，背包容量减少w(i)但价值增加了v(i)；

　　　　由此可以得出递推关系式：

　　　　1) **j<w(i)      V(i,j)=V(i-1,j)**

　　　　2) **j>=w(i)     V(i,j)=max｛ V(i-1,j)，V(i-1,j-w(i))+v(i) ｝**



void FindMax()//动态规划

{

int i,j;

//填表

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

{

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

{

if(j<w[i])//包装不进

{

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

}

else//能装

{

if(V[i-1][j]>V[i-1][j-w[i]]+v[i])//不装价值大

{

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

}

else//前i-1个物品的最优解与第i个物品的价值之和更大

{

V[i][j]=V[i-1][j-w[i]]+v[i];

}

}

}

}

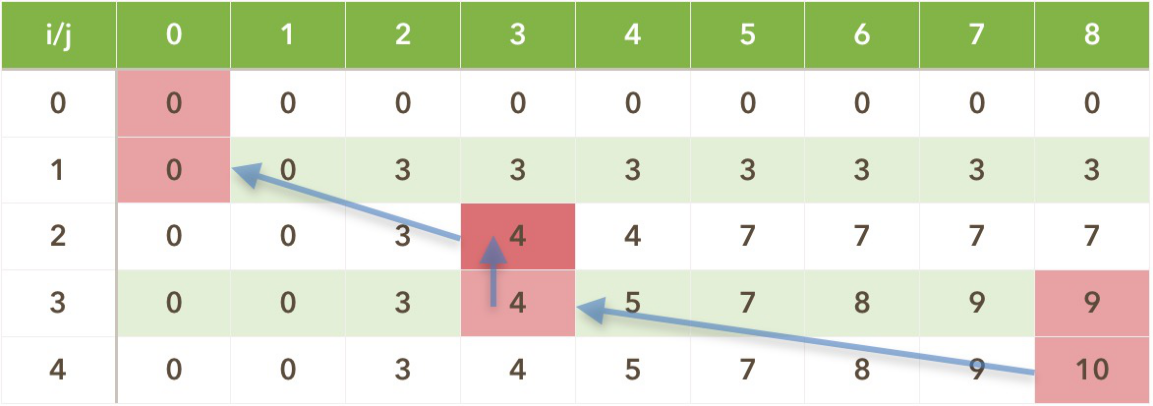
}

1) 最优解为V(4,8)=10，而V(4,8)!=V(3,8)却有V(4,8)=V(3,8-w(4))+v(4)=V(3,3)+6=4+6=10，所以第4件商品被选中，并且回到V(3,8-w(4))=V(3,3)；

　　　　2) 有V(3,3)=V(2,3)=4，所以第3件商品没被选择，回到V(2,3)；

　　　　3) 而V(2,3)!=V(1,3)却有V(2,3)=V(1,3-w(2))+v(2)=V(1,0)+4=0+4=4，所以第2件商品被选中，并且回到V(1,3-w(2))=V(1,0)；

　　　　4) 有V(1,0)=V(0,0)=0，所以第1件商品没被选择；



void FindWhat(int i,int j)//寻找解的组成方式

{

if(i>=0)

{

if(V[i][j]==V[i-1][j])//相等说明没装

{

item[i]=0;//全局变量，标记未被选中

FindWhat(i-1,j);

}

else if( j-w[i]>=0 && V[i][j]==V[i-1][j-w[i]]+v[i] )

{

item[i]=1;//标记已被选中

FindWhat(i-1,j-w[i]);//回到装包之前的位置

}

}

}

**最大子数组和问题**

一个有N个整数元素的一维数组（A[0],A[1],…A[N-1]），这个数组有很多子数组，求子数组和的最大值？注意：子数组必须是连续的、不需要返回子数组的具体位置、数组中包含：正、负、零整数、子数组不能空。

int A[5] = {-1,2,3,-4,2};

符合条件的子数组为2，3，即答案为5

我们不妨考虑一个这样的序列：1，-3，5，-2，4

a[i]表示这个序列的第 i 个元素，dp[i]表示最后一个元素是a[i]的最大连续和（此乃状态，是不是跟LIS的DP解法有点类似），于是：

dp[0] : a[0] ; ( 1 )

dp[1] : max(dp[0] + a[1] , a[1]) ; ( -2 )

dp[2] : max(dp[1] + a[2] , a[2]) ; ( 5 )

dp[3] : max(dp[2] + a[3] , a[3]) ; ( 3 )

dp[4] : max(dp[3] + a[4] , a[4]) ; ( 7 )

所以：ans = 7 （dp数组的最大值）

于是，我们可以得到状态转移方程：dp[i+1] = max(dp[i]+a[i+1] , a[i+1])

状态方程 ： max( dp[ i ] )  = getMax(  max( dp[ i -1 ] ) + arr[ i ] ,arr[ i ] ) 。上面式子的意义是：我们从头开始遍历数组，遍历到数组元素 arr[ i ] 时，连续的最大的和 可能为 max( dp[ i -1 ] ) + arr[ i ] ，也可能为 arr[ i ] ，做比较即可得出哪个更大，取最大值。时间复杂度为 n

int GetMax(int a, int b) //得到两个数的最大值

{

return (a) > (b) ? (a) : (b);

}

int GetMaxAddOfArray(int\* arr, int sz)

{

if (arr == NULL || sz <= 0)

return 0;

int Sum = arr[0]; //临时最大值

int MAX = arr[0]; //比较之后的最大值

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

{

Sum = GetMax(Sum + arr[i], arr[i]); //状态方程

if (Sum >= MAX)

MAX = Sum;

}

return MAX;

}

int main()

{

int array[] = { 2, 3, -6, 4, 6, 2, -2, 5, -9 };

int sz = sizeof(array) / sizeof(array[0]);

int MAX = GetMaxAddOfArray(array, sz);

cout << MAX << endl;

return 0;

}

**将n个相同的小球放入m个相同的盒子，问有多少种方法**。

5，1，1和1，5，1 是同一种分法。

①    当放入小球后，球数最少的盒子为空，那么就相当于，把i个小球放入j-1个盒子里面，即a[i][j-1]。

②    当放入小球后，球数最少的盒子不为空，那么说明每个盒子里面都至少有一个小球，那么就预先把每个盒子里面放入一个小球好了，有 a[i-j] [j]。

所以 ，递推公式为 a[i][j]= a[i][j-1]+a[i-j][j]，这里是i>=j的。

当i<j的时候，无论怎么放，肯定会存在空盒，不妨先拿走一个空盒（保证有空盒）,然后把i个相同的小球放入剩余的j-1个相同的盒子中，有a[i][j-1]种方法，所以a[i][j]=a[i][j-1].

综上：将n个相同的小球放入m个相同的盒子有

当i>=j时，a[i][j]=a[i][j-1]+a[i-j][j];

当i<时，a[i][j]=a[i][j-1],（仔细想想，这里也可以写成a[i][j]=a[i][i]，因为a[i][i]=a[i][i+1]=a[i][i+2]=a[i][i+3]…….

#include <iostream>

using namespace std;

const int maxn=11;

int a[maxn][maxn];

void init()

{

for(int i=1;i<=10;i++)

a[0][i]=1,a[i][1]=1;

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

for(int j=2;j<=maxn;j++)

if(j<=i)

a[i][j]=a[i][j-1]+a[i-j][j];

else

a[i][j]=a[i][i];

}

int main()

{

init();

int t,n,m;

cin>>t;

while(t--)

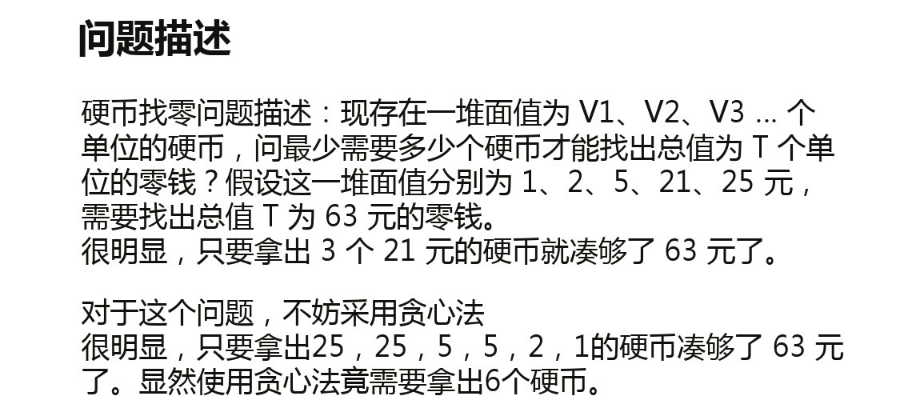
{

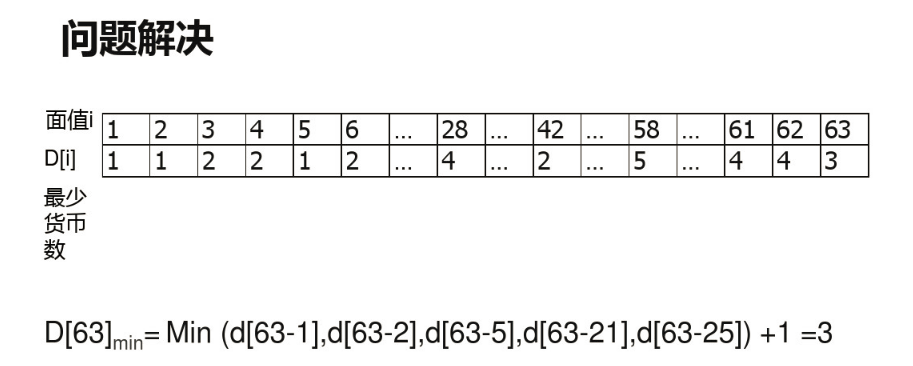
cin>>n>>m;

cout<<a[n][m]<<endl;

}

return 0;}







也可以c[i,j]= min{c[i-1,j] , c[i, j-coinsValues[i]] + 1}

**考虑该问题的最优子结构：设 c[i,j]表示 可用第 0，1，.... i 枚硬币 对 金额为 j 的钱 进行找钱 所需要的最少硬币数。**

**i 表示可用的硬币种类数， j 表示 需要找回的零钱**

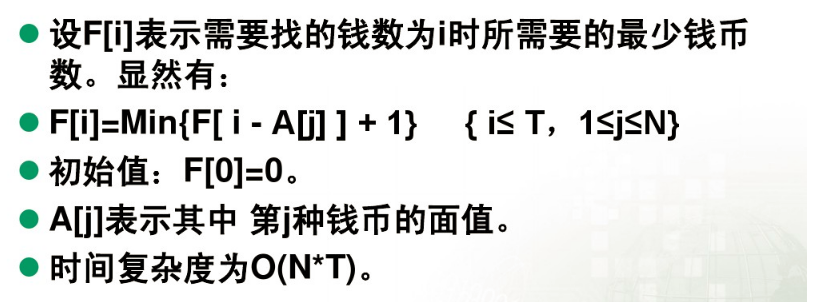
第 i 枚硬币有两种选择：用它来找零 和 不用它找零。因此，c[i,j]的最优解如下：

c[i,j]= min{c[i-1,j] , c[i, j-coinsValues[i]] + 1}   其中，

c[i-1,j] 表示 **不使用**第 i 枚硬币找零时，对金额为 j 进行找钱所需要的**最少**硬币数

c[i, j-coinsValues[i]] + 1 表示**使用** 第 i 枚硬币找零时，对金额为 j 进行找钱所需要的**最少**硬币数。由于用了第 i 枚硬币，故使用的硬币数量要增1

c[i,j] 取二者的较小值的那一个。



**有n级台阶，一个人每次上一级或者两级，问有多少种走完n级台阶的方法。**

分析：动态规划的实现的关键在于能不能准确合理的用动态规划表来抽象出 实际问题。在这个问题上，我们让f(n)表示走上n级台阶的方法数。

那么当n为1时，f(n) = 1,n为2时，f(n) =2,就是说当台阶只有一级的时候，方法数是一种，台阶有两级的时候，方法数为2。那么当我们要走上n级台阶，必然是从n-1级台阶迈一步或者是从n-2级台阶迈两步，所以到达n级台阶的方法数必然是到达n-1级台阶的方法数加上到达n-2级台阶的方法数之和。即**f(n) = f(n-1)+f(n-2)**，我们用dp[n]来表示动态规划表，dp[i],i>0,i<=n,表示到达i级台阶的方法数。

/\*dp是全局数组，大小为n,全部初始化为0，是题目中的动态规划表\*/

int fun(int n){

if (n==1||n==2)

return n;

/\*判断n-1的状态有没有被计算过\*/

if (!dp[n-1])

dp[n-1] = fun(n-1);

if(!dp[n-2])

dp[n-2]=fun(n-2);

return dp[n-1]+dp[n-2];

}

**给定数组arr，返回arr的最长递增子序列的长度，比如arr=[2,1,5,3,6,4,8,9,7]，最长递增子序列为[1,3,4,8,9]返回其长度为5.**

首先生成dp[n]的数组，dp[i]表示以必须arr[i]这个数结束的情况下产生的最大递增子序列的长度。对于第一个数来说，很明显dp[0]为1，当我们计算dp[i]的时候，我们去考察i位置之前的所有位置，找到i位置之前的最大的dp值，记为dp[j](0=<j<i),dp[j]代表以arr[j]结尾的最长递增序列，而dp[j]又是之前计算过的最大的那个值，我们在来判断arr[i]是否大于arr[j],如果大于dp[i]=dp[j]+1.计算完dp之后，我们找出dp中的最大值，即为这个串的最长递增序列。

2, 11, 4, 12, 6, 1

在我们不知道答案的情况下，当我们扫描到6的时候，应该怎样找出2, 4呢？  
为了找出能与6组合的****LIS****，我们要依次检查****结尾元素在**6**左边且比**6**小****的LIS：

* 以2结尾的****LIS****
* 以4结尾的****LIS****

#include <iostream>

#include <algorithm>

using namespace std;

/\*动态规划表\*/

int dp[5] = {};

int main(){

int arr[5] = {2,4,5,3,1};

dp[0] = 1;

const int oo = 0;

for (int i = 1;i<5;i++){

int \_max = oo;

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

if(dp[j]>\_max&&arr[i]>arr[j])

\_max = dp[j];

dp[i] = \_max+1;

}

int maxlist=0;

for (int i = 0; i < 5;i++)

if (dp[i] > maxlist)

maxlist=dp[i];

cout << maxlist << endl;

}

**给定两个字符串str1和str2，返回两个字符串的最长公共子序列**，例如：str1="1A2C3D4B56",str2="B1D23CA45B6A","123456"和"12C4B6"都是最长公共子序列，返回哪一个都行

分析：本题是非常经典的动态规划问题，假设str1的长度为M，str2的长度为N，则生成M\*N的二维数组dp，dp[i][j]的含义是str1[0..i]与str2[0..j]的最长公共子序列的长度。

dp值的求法如下：

dp[i][j]的值必然和dp[i-1][j],dp[i][j-1],dp[i-1][j-1]相关，结合下面的代码来看，我们实际上是从第1行和第1列开始计算的，而把第0行和第0列都初始化为0，这是为了后面的取最大值在代码实现上的方便，dp[i][j]取三者之间的最大值。

int findLCS(string A, int n, string B, int m) {

// n表示字符串A的长度，m表示字符串B的长度

int dp[500][500] = {};

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

{

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

{

if (A[i]==B[j])

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

else

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

}

}

return dp[n][m];

}

**LIS（Longest Increasing Subsequence）最长上升子序列 或者 最长不下降子序列**。很基础的题目，有两种算法，复杂度分别为O(n\*logn)和O(n^2)

先回顾经典的O(n^2)的动态规划算法:

设a[t]表示序列中的第t个数，dp[t]表示从1到t这一段中以t结尾的最长上升子序列的长度，初始时设dp[t] = 0(t = 1, 2, ..., len(A))。则有动态规划方程：dp[t] = max{1, dp[j] + 1} (j = 1, 2, ..., t - 1, 且a[j] < a[t])。

一般若从a[t]开始，此时最长不下降子序列应该是按下列方法求出的：

 在a[t+1],a[t+2],...a[n]中，找出一个比a[t]大的且最长的不下降子序列，作为它的后继。

#include<iostream>

using namespace std;

#define max(a,b) a>b?a:b

int main()

{

int n, i, j, dp[101], x[101], max\_len;

while (cin >> n)

{

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

cin >> x[i];

dp[0] = 1;//表示以x[0]为子序列最右边的长度位1

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

{

dp[i] = 1;//初始化每种情况最小值为1

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

{

if (x[i]>x[j] && dp[j] + 1>dp[i])//从0-i进行扫描,查找边界小于当前最优解长度相等的解优化最优解

dp[i] = dp[j] + 1;//如果允许子序列相邻元素相同 x[i]>=x[j]&&dp[j]+1>dp[i];

}

}

for (i = max\_len = 0; i < n; i++)

max\_len = max(max\_len, dp[i]);//等到最大子序列长度

cout << max\_len << endl;

}

return 0;

}

**最长不下降子序列nlogn算法详解**

 定义：a[1..n]为原始序列，d[k]表示**长度为k的不下降子序列末尾元素的最小值**，len表示当前已知的最长子序列的长度。

初始化：d[1]=a[1]; len=1; （0个元素的时候特判一下）

//最长不下降子序列nlogn Song

#include<cstdio>

#include<algorithm>

using namespace std;

int a[40005];

int d[40005];

int main()

{

int n;

scanf("%d",&n);

for (int i=1;i<=n;i++) scanf("%d",&a[i]);

if (n==0) //0个元素特判一下

{

printf("0\n");

return 0;

}

d[1]=a[1]; //初始化

int len=1;

for (int i=2;i<=n;i++)

{

if (a[i]>=d[len]) d[++len]=a[i]; //如果可以接在len后面就接上，如果是最长上升子序列，这里变成>

else //否则就找一个最该替换的替换掉

{

int j=upper\_bound(d+1,d+len+1,a[i])-d; //找到第一个大于它的d的下标，如果是最长上升子序列，这里变成lower\_bound

d[j]=a[i];

}

}

printf("%d\n",len);

return 0;

}

**打家劫舍**

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

**输入:** [2,7,9,3,1]**输出:** 12**解释:** 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。

  偷窃到的最高金额 = 2 + 9 + 1 = 12 。

对于此题，如果只有两家或者以下，我们选择金额最大的。如果2家以上，那我们打劫到第 i 家的时候，就要考虑，要不要打劫这一家，也就是（这一家的价值+打劫到 i - 2家的最大价值）和（打劫到上一家（i - 1）的最大价值），比较这两个值，选较大值作为打劫到第 i 家的最大价值。最后输出最后一家就可以了。

class Solution {

public int rob(int[] nums) {

int n = nums.length;

if (n == 0) {

return 0;

}

int[] f = new int[n];

if (n == 1) {

return nums[0];

}

int a = Math.max(nums[0], nums[1]);

if (n == 2) {

return a;

}

f[0] = nums[0];

f[1] = a;

for (int i = 2; i < f.length; i++) {

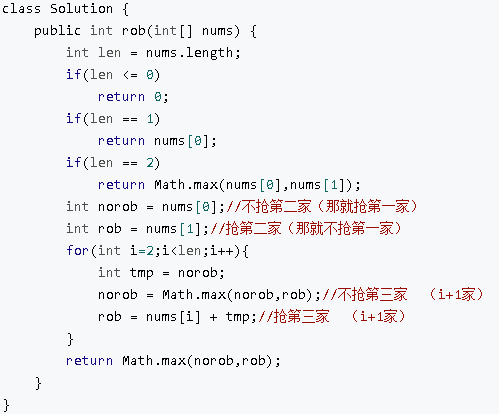
f[i] = Math.max(f[i - 2] + nums[i], f[i -1]);

}

return f[n - 1];

}

}



不抢第三家 ==》就是抢和不抢第二家的最大值。   
抢第三家===》就是第三家的值加上不抢第二家的值。需要临时变量保存一个变量。

**打家劫舍 II**

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

* 先求出第一家到倒数第二家的最大钱财数量
* 然后求出第二家到最后一家的最大钱财数量
* 最后求两者的较大值

class Solution {

public int robCycle(int[] nums,int low,int high){

int len = high - low + 1;

if(len == 1)

return nums[low];

if(len == 2)

return Math.max(nums[low],nums[high]);

int norob = nums[low];

int rob = nums[low+1];

for(int i=low+2;i<=high;i++){//注意：i的初始值和最后的值

int tmp = norob;

norob = Math.max(norob,rob);

rob = nums[i] + tmp;//注意：这里是tmp

}

return Math.max(norob,rob);

}

public int rob(int[] nums) {

if(nums.length == 0)

return 0;

if(nums.length == 1)

return nums[0];

int res1 = robCycle(nums,0,nums.length-2);

int res2 = robCycle(nums,1,nums.length-1);

return Math.max(res1,res2);

}

}

**母牛生产**（今年的数量，去年的数量，新增的数量）

题目描述：假设农场中成熟的母牛每年都会生 1 头小母牛，并且永远不会死。第一年有 1 只小母牛，从第二年开始，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。给定整数 N，求 N 年后牛的数量。

新增的数量是距今3年前的牛的数量

f[i]代表第i年母牛的数量，那么f[i]=f[i-1]+f[i-3]，意思是：

第i年的母牛头数 = 去年的木牛头数 + 新生的小牛头数(只有距第i年3年前的牛才能生小牛)。

写代码的时候i=1开始比较方便！

**问题描述： 给定数组，找出连续乘积最大值的子序列。**  
例如 0，-1，-3，-2，则最大连续乘积为6= (-3) \* (-2)

实现思路

此题与最大连续和的子序列问题类似，也可通过找到递推公式然后用DP来解。

关键在于求公式的过程要考虑到元素可能为负的情况。假设元素都为正数的话，DP公式为：

dp[i] = max(a[i],dp[i-1]\*a[i]) ,乘或不乘，取最大的那个

可元素可能为负数，因此可以使用min和max分别存当前最大值和最小值，如果当前元素为负数的话，当前最小值就成了最大值。这样一来，dp公式为：

max = Max(Max(max \* a[i], min \* a[i]), a[i]) ，先从max \* a[i]和min \* a[i]找出最大值，在与a[i]进行比较

min = Min(Min(max \* a[i], min \* a[i]), a[i]) ，同上，只是取的是最小值

public class Solution {

public int MaxProduct(int[] nums) {

var len = nums.Length;

if(len == 0)

{

return 0;

}

var max = nums[0];

var min = nums[0];

var result = nums[0];

for(var i = 1;i < len; i++){

var tmpMax = Math.Max(Math.Max(max \* nums[i] , min \* nums[i]), nums[i]);

var tmpMin = Math.Min(Math.Min(max \* nums[i] , min \* nums[i]), nums[i]);

max = tmpMax;

min = tmpMin;

result = Math.Max(result , max);

}

return result;

}

}

**动态规划之矩阵连乘问题**

**当i<j m[i][j]=m[i[k]+m[k+1][j] + Pi-1\*Pk\*Pj**

**当i=j m[i][j] = 0**

#include<iostream>

#define LEN 6 // 连乘的矩阵个数

using namespace std;

//矩阵的维数分别是 30x35,35x15,15x5,5x10,10x20,20x25

int p[LEN + 1] ={ 30,35,15,5,10,20,25};

int m[LEN + 1][LEN + 1] = { 0,0,0};

int s[LEN + 1][LEN + 1] = { 0,0,0};

void MatrixChain(int \*p,int m[][LEN + 1], int s[][LEN + 1]){

for(int i = 0; i <= LEN; ++i){

m[i][i] = 0;

}//初始化，即链长为1

for(int r = 2; r <= LEN; ++r){//链长从2到LEN

for(int i = 1; i <= LEN - r + 1; ++i){对i位置的矩阵计算链长为r的最优解

int j = i + r - 1;

m[i][j] = m[i + 1][j] + p[i-1]\*p[i]\*p[j];

s[i][j] = i;

for(int k = i + 1; k < j; ++k){//k 是i,j 之间的断点

int t = m[i][k] + m[k + 1][j] + p[i-1]\*p[k]\*p[j];//m[i][k] + m[k + 1][j]链长短于当前链，因此已经计算出

if(m[i][j] > t){

s[i][j] = k;//断开点

m[i][j] = t;//最小计算量

}

}

}

}

}

void TrackBack(int i, int j,int s[][LEN + 1]){

if(i == j || i == LEN){

cout<<"A"<<i;

return;

}

cout<<"(";

TrackBack(i,s[i][j],s);

cout<<")(";

TrackBack(s[i][j] + 1,j,s);

cout<<")";

}

int main()

{

MatrixChain(p,m,s);

TrackBack(1,LEN,s);

cout<<endl;

}

**动态规划——整数划分**

根据本题要求，要将整数n划分为k部分，而且在不考虑顺序的情况下每种划分互不相同。

在此可以把问题抽象为一下模型：

有n个箱子，要把它们垒成k栋，可以假设箱子按某种满足该要求的方式已经摆好，现在从最底层每栋拿掉一个箱子，这样就只有两种情况：

1）之前每栋箱子的数量都多于1，则拿掉箱子后总栋数不变，则之前排列n个箱子成k栋的情况数与之后排列n-k个箱子成k栋的情况数是一样的

2）之前有一栋箱子的数量是1，则之前排列n个箱子成k栋的情况数与之后排列n-1个箱子成k-1栋的情况数相同

于是得出递推方程dp[n][k]=dp[n-j][k]+dp[n-1][k-1];

出口是：dp[i][1]=1;

#include <stdio.h>

const int maxn=200;

const int maxk=6;

long dp[maxn+1][maxk+1];

int k,n;

int main()

{

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

dp[i][1]=1;

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

for(int j=2;j<=maxk;j++)

if(i>=j)

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

while(~scanf("%d%d",&n,&k))

printf("%ld\n",dp[n][k]);

return 0;

}

**[分苹果](http://blog.csdn.net/sunnyyoona/article/details/46787233" \t "https://blog.csdn.net/huzhigenlaohu/article/details/_blank)**

M个相同苹果放到N个相同篮子里有多少种放法,允许有篮子不放。

设f(m,n) 为m个苹果，n个盘子的放法数目：

当n>m：必定有n-m个盘子永远空着，去掉它们对摆放苹果方法数目不产生影响。即if(n>m) f(m,n) = f(m,m)

当n<=m：不同的放法可以分成两类：

（1）有至少一个盘子空着，即相当于f(m,n) = f(m,n-1);

（2）所有盘子都有苹果，相当于可以从每个盘子中拿掉一个苹果，不影响不同放法的数目，即f(m,n) = f(m-n,n).而总的放苹果的放法数目等于两者的和，即 f(m,n) =f(m,n-1)+f(m-n,n)

递归出口条件说明：

当n=1时，所有苹果都必须放在一个盘子里，所以返回１；

当没有苹果可放时，定义为１种放法；

递归的两条路，第一条n会逐渐减少，终会到达出口n==1;

第二条m会逐渐减少，因为n>m时，我们会return f(m,m)　所以终会到达出口m==0

#include <iostream>

#include <string>

#include <vector>

#include <stack>

#include <algorithm>

using namespace std;

// apple 个 苹果 basket 个 篮子

int ShareApple(int apple,int basket){

// 因为我们总是让apple >= basket来求解的，所以apple - basket >= 0,

// 让apple = 0时候结束，如果改为apple = 1，可能得不到正确解

if(apple == 0 || basket == 1){

return 1;

}//if

// 篮子多于苹果 按照苹果个数分

else if(apple < basket){

return ShareApple(apple,apple);

}//else

return ShareApple(apple,basket-1) + ShareApple(apple - basket,basket);

}

int main(){

int apple,basket;

//freopen("C:\\Users\\Administrator\\Desktop\\acm.txt","r",stdin);

while(cin>>apple>>basket){

cout<<ShareApple(apple,basket)<<endl;

}//while

return 0;

}

/\*

   整数划分

   （一）将n划分成若干不同整数之和的划分数

   （二）将n划分成若干正整数之和的划分数

   （三）将n划分成k个正整数之和的划分数

   （四）将n划分成最大数不超过k的划分数

   （五）将n划分成若干个 奇正整数之和的划分数

\*/

#include<iostream>

#include<cstdio>

#include<cstdlib>

#include<cmath>

#include<cstring>

#include<vector>

#include<queue>

#include<set>

#include<map>

#include<algorithm>

#include<sstream>

#define eps 1e-9

#define pi acos(-1)

#define INF 0x7fffffff

#define inf -INF

#define MM 12900

#define N 50

using namespace std;

typedef long long ll;

const int \_max = N + 10;

int dp[\_max][\_max],n,k,out[6];

int main(){

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

#endif // ONLINE\_JUDGE

while(scanf("%d%d",&n,&k)==2){

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*整数划分（二）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

memset(dp,0,sizeof(dp));

dp[0][0] = 1;

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

for(int j = 1; j <= n; ++ j){

if(j>i)dp[i][j]=dp[i][i];

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

}

out[1] = dp[n][n];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*整数划分（四）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

out[3] = dp[n][k];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*整数划分（三）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

memset(dp,0,sizeof(dp));

dp[0][0] = 1;

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

for(int j = 1; j <= i; ++ j){

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

}

out[2] = dp[n][k];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*整数划分（五）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

memset(dp,0,sizeof(dp));

dp[0][0] = 1;

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

for(int j = 1; j <= n; ++ j){

if(j&1){

if(j>i)dp[i][j] = dp[i][i];

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

}

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

}

out[4] = dp[n][n];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***\*整数划分（一）**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

memset(dp,0,sizeof(dp));

dp[0][0] = 1;

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

for(int j = 1; j <= n; ++ j){

if(j>i)dp[i][j]=dp[i][i];

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

}

out[5] = dp[n][n];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*输出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for(int i = 1; i<= 5; ++ i)

printf("%d\n",out[i]);

printf("\n");

}

return 0;

}

/\*

/\*\*\*\*\*（一）**将n划分成若干不同整数之和的划分数**\*\*\*\*\*\*\*\*\*\*\*\*

dp[i][j]表示将整数i划分成不超过j的划分数，分含不含j两种情况

dp[0][0] = 1

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

= dp[i][i] (j >i)

=>ans = dp[n][n]

/\*\*\*\*\*（二）**将n划分成若干正整数之和的划分数**\*\*\*\*\*\*\*\*\*\*\*\*\*

dp[i][j]表示将整数i划分成不超过j的划分数，分含不含j两种情况

与（一）区别，j可重复

dp[0][0] = 1

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

= dp[i][i] (j >i)

=>ans = dp[n][n]

/\*\*\*\*\*（三）**将n划分成k个正整数之和的划分数**\*\*\*\*\*\*\*\*\*\*\*\*\*

dp[i][j]表示将整数i划分成j个正整数的划分数，考虑j组数中含不含1

dp[0][0] = 1

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

如果不包含1，那么每组数至少为2，从每堆数中各拿出1还能够成j堆数dp[i-j][j]

=>ans = dp[n][k]

/\*\*\*\*\*（四）**将n划分成最大数不超过k的划分数**\*\*\*\*\*\*\*\*\*\*\*\*

dp[i][j]表示将整数i划分成不超过j的划分数，分含不含j两种情况

是（二）的特例

dp[0][0] = 1

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

= dp[i][i] (j >i)

=>ans = dp[n][k]

/\*\*\*\*\*（五）**将n划分成若干个 奇正整数之和的划分数**\*\*\*\*\*\*

dp[i][j]表示将整数i划分成不超过j的划分数，分含不含j两种情况

dp[0][0] = 1;

j是奇数，正常判断

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

= dp[i][i] (j >i)

j是偶数,dp[i][j] = dp[i][j-1]//往下递推

=>ans = dp[n][n] \*/

**最大子段和问题**

问题定义：对于给定序列a1,a2,a3……an,寻找它的某个连续子段，使得其和最大。如( -2,11,-4,13,-5,-2 )最大子段是{ 11,-4,13 }其和为20。

思路：辅助数组b[j]用来记录一j为尾的子段和集合中的最大字段和，a[i]为序列的第i个元素。   
那么当b[i-1]>0时,b[i]=b[i-1]+a[i],否则 b[i]=a[i];

#include <iostream>

using namespace std;

int MIS(int n,int a[],int b[],int max)

{

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

{

if (i == 0)

{

b[i] = a[i];

max = b[i];

}

else

{

if (b[i - 1] <= 0)

b[i] = a[i];

else

b[i] = b[i - 1] + a[i];

if (b[i] > max)

max = b[i];

}

}

return max;

}

int main()

{

cout << "请输入数字个数：";

int n;

cin >> n;

cout << "请输入数字，每个数字中间用空格隔开";

int a[10000];

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

cin >> a[i];

int b[10000],max = 0;

max = MIS(n, a, b, max);

cout << "最大字段和为：" << max << endl;

system("pause");

}