动态规划

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解

保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式

应用场景：

适用动态规划的问题必须满足最优化原理、无后效性和重叠性。

**优化原理** 一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

**无后效性** ，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

**子问题的重叠性** 其中的关键在于解决冗余，这是动态规划算法的根本目的

01背包问题

<https://blog.csdn.net/stack_queue/article/details/53544109> （背包九讲内容）

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为f[i-1][v]；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]

优化空间复杂度

以上方法的时间和空间复杂度均为O(N\*V)，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到O(V)。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环i=1..N，每次算出来二维数组f[i][0..V]的所有值。那么，如果只用一个数组f[0..V]，能不能保证第i次循环结束后f[v]中表示的就是我们定义的状态f[i][v]呢？f[i][v]是由f[i-1][v]和f[i-1][v-c[i]]两个子问题递推而来，能否保证在推f[i][v]时（也即在第i次主循环中推f[v]时）能够得到f[i-1][v]和f[i-1][v-c[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的顺序推f[v]，这样才能保证推f[v]时f[v-c[i]]保存的是状态f[i-1][v-c[i]]的值。伪代码如下：

for i=1..N

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因为现在的f[v-c[i]]就相当于原来的f[i-1][v-c[i]]。如果将v的循环顺序从上面的逆序改成顺序的话，那么则成了f[i][v]由f[i][v-c[i]]推知，与本题意不符，但它却是另一个重要的背包问题P02最简捷的解决方案，故学习只用一维数组解01背包问题是十分必要的。

事实上，使用一维数组解01背包的程序在后面会被多次用到，所以这里抽象出一个处理一件01背包中的物品过程，以后的代码中直接调用不加说明。

过程ZeroOnePack，表示处理一件01背包中的物品，两个参数cost、weight分别表明这件物品的费用和价值。

procedure ZeroOnePack(cost,weight)

for v=V..cost

f[v]=max{f[v],f[v-cost]+weight}

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成v=V..0是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为cost的物品不会影响状态f[0..cost-1]，这是显然的。

有了这个过程以后，01背包问题的伪代码就可以这样写：

for i=1..N

ZeroOnePack(c[i],w[i]);

初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了f[0]为0其它f[1..V]均设为-∞，这样就可以保证最终得到的f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将f[0..V]全部设为0。

为什么呢？可以这样理解：初始化的f数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为0的背包可能被价值为0的nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是-∞了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

小结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的空间复杂度。

完全背包问题

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……等很多种。如果仍然按照解01背包时的思路，令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

f[i][v]=max{f[i-1][v-k\*c[i]]+k\*w[i]|0<=k\*c[i]<=v}

这跟01背包问题一样有O(N\*V)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度是超过O(VN)的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品i、j满足c[i]<=c[j]且w[i]>=w[j]，则将物品j去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得j换成物美价廉的i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的O(N^2)地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于V的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以O(V+N)地完成这个优化。这个不太重要的过程就不给出伪代码了，希望你能独立思考写出伪代码或程序。

转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c[i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第i种物品拆成费用为c[i]\*2^k、价值为w[i]\*2^k的若干件物品，其中k满足c[i]\*2^k<=V。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件物品的和。这样把每种物品拆成O(log(V/c[i]))件物品，是一个很大的改进。

但我们有更优的O(VN)的算法。

O(VN)的算法

这个算法使用一维数组，先看伪代码：

for i=1..N

for v=0..V

f[v]=max{f[v],f[v-cost]+weight}

你会发现，这个伪代码与P01的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v=0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：

f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码，以后会用到：

procedure CompletePack(cost,weight)

for v=cost..V

f[v]=max{f[v],f[v-c[i]]+w[i]}

总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法“的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

P03: 多重背包问题

题目

有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第i种物品有n[i]+1种策略：取0件，取1件……取n[i]件。令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值，则有状态转移方程：

f[i][v]=max{f[i-1][v-k\*c[i]]+k\*w[i]|0<=k<=n[i]}

复杂度是O(V\*Σn[i])。

转化为01背包问题

另一种好想好写的基本方法是转化为01背包求解：把第i种物品换成n[i]件01背包中的物品，则得到了物品数为Σn[i]的01背包问题，直接求解，复杂度仍然是O(V\*Σn[i])。

但是我们期望将它转化为01背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

方法是：将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。例如，如果n[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。

分成的这几件物品的系数和为n[i]，表明不可能取多于n[i]件的第i种物品。另外这种方法也能保证对于0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0..2^k-1和2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第i种物品分成了O(log n[i])种物品，将原问题转化为了复杂度为O(V\*Σlog n[i])的01背包问题，是很大的改进。

下面给出O(log amount)时间处理一件多重背包中物品的过程，其中amount表示物品的数量：

procedure MultiplePack(cost,weight,amount)

if cost\*amount>=V

CompletePack(cost,weight)

return

integer k=1

while k<num

ZeroOnePack(k\*cost,k\*weight)

amount=amount-k

k=k\*2

ZeroOnePack(amount\*cost,amount\*weight)

希望你仔细体会这个伪代码，如果不太理解的话，不妨翻译成程序代码以后，单步执行几次，或者头脑加纸笔模拟一下，也许就会慢慢理解了。

O(VN)的算法

多重背包问题同样有O(VN)的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊O(1)的时间求解。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

小结

这里我们看到了将一个算法的复杂度由O(V\*Σn[i])改进到O(V\*Σlog n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)算法。希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并将完整的程序代码写出来

其他的几种待续

动态规划以及记忆化搜索

记忆化搜索：算法上依然是搜索的流程，但是搜索到的一些解用动态规划的那种思想和模式作一些保存。

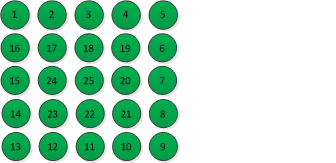
一般说来，动态规划总要遍历所有的状态，而搜索可以排除一些无效状态。

更重要的是搜索还可以剪枝，可能剪去大量不必要的状态，因此在空间开销上往往比动态规划要低很多。

记忆化算法在求解的时候还是按着自顶向下的顺序，但是每求解一个状态，就将它的解保存下来，

比如流水线调度问题，矩阵链乘问题等等都是“一步接着一步解决的”，即规模为 i 的问题需要基于规模 i-1 的问题进行最优解选择，通常的递归模式为DP(i)=optimal{DP(i-1)}。而记忆化搜索本质上也是DP思想，当子问题A和子问题B存在子子问题C时，如果子子问题C的最优解已经被求出，那么子问题A或者是B只需要“查表”获得C的解，而不需要再算一遍C。记忆化搜索的DP模式比普通模式要“随意一些”，通常为DP(i)=optimal(DP(j)), j < i

例题 滑雪问题



上图显示为R\*C的雪场，R是行数，C是列数。圆圈内的数字表示的是雪场的海拔高度h，根据常识我们知道，滑雪只有从上往下滑行才可能滑的动，现在我们想要求出能够滑行的最长距离，上面的例子我们可以很直接判断出25-24-......-1这个顺时针方向螺旋的滑雪方式可以滑的最远。

那么这个问题如何用编程来实现呢？我们发现这是一个典型的递推，DP(i, j)表示从坐标（i，j）出发所能滑行的最大长度，且有：DP(i, j)=optimal{DP(i±1, j±1)}+1

#include<iostream>

#include<cstdio>

#include<cstring>

#include<algorithm>

using namespace std;

const int max\_size=110;

int R,C;

int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};

int h[max\_size][max\_size],dp[max\_size][max\_size];

int inMap(int x,int y){

if(x>=0&&x<=R-1&&y>=0&&y<=C-1) return 1;

return 0;

}

int max2(int a,int b,int c,int d){

return max(max(a,b),max(c,d));

}

int dfs(int i,int j){

int nx,ny,down=0,up=0,left=0,right=0;

if(dp[i][j]) return dp[i][j];

nx=i+dir[0][0]; ny=j+dir[0][1];

if(inMap(nx,ny)){

if(h[i][j]>h[nx][ny]) up=dfs(nx,ny);

}

nx=i+dir[1][0]; ny=j+dir[1][1];

if(inMap(nx,ny)){

if(h[i][j]>h[nx][ny]) right=dfs(nx,ny);

}

nx=i+dir[2][0]; ny=j+dir[2][1];

if(inMap(nx,ny)){

if(h[i][j]>h[nx][ny]) down=dfs(nx,ny);

}

nx=i+dir[3][0]; ny=j+dir[3][1];

if(inMap(nx,ny)){

if(h[i][j]>h[nx][ny]) left=dfs(nx,ny);

}

dp[i][j]=max2(up,down,left,right)+1;

return dp[i][j];

}

int main(){

scanf("%d%d",&R,&C);

memset(h,0,sizeof(h));

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

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

for(int j=0;j<C;j++){

scanf("%d",&h[i][j]);

}

}

int ans=-1;

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

for(int j=0;j<C;j++){

ans=max(ans,dfs(i,j));

}

}

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

}

给你一根长n英尺的棒子和一份关于该棒子的价目表如下（其中 i = 1,2,3,…,n），请问如何将这根棒子卖出最高的价格，可以对棒子进行切割。

http://www.roading.org/images/2012-03/image_thumb.png

这个题同样是可以利用DP记忆化搜索来实现的，递推公为DP(n)=optimal{max{price(i)+DP(n-i)|1≤i≤n}}。

#include<iostream>

#include<cstdio>

#include<cstring>

using namespace std;

const int max\_size=50;

const int inf=1<<30;

int price[max\_size],dp[max\_size];

int n;

int dfs(int n){

if(dp[n]) return dp[n];

if(n==0) return 0;

int mmax=-inf;

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

mmax=max(mmax,price[i]+dfs(n-i));

}

dp[n]=mmax;

return dp[n];

}

int main(){

while(scanf("%d",&n)!=EOF){

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

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

}

}

有N件物品和一个重量为M的背包。（每种物品均只有一件）第i件物品的重量是w[i]，价值是p[i]。求解将哪些物品装入背包可使价值总和最大

#include<iostream>

#include<cstdio>

#include<cstring>

using namespace std;

const int max\_size=50;

const int inf=1<<30;

int p[max\_size],w[max\_size],dp[max\_size][max\_size];

int n,v;

int dfs(int i,int v){

if(dp[i][v]) return dp[i][v];

if(i==0||v<=0) return 0;

if(w[i]>v) dp[i][v]=dfs(i-1,v);

else dp[i][v]=max(dfs(i-1,v),dfs(i-1,v-w[i])+p[i]);

return dp[i][v];

}

int main(){

while(scanf("%d",&n)!=EOF){

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

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

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

scanf("%d",&v);

printf("%d\n",dfs(n,v));

}

}