你有一个包

你有一个包，你想要带着这个包去打劫便利店。便利店里东西太多了，你的包装！不！下！啊！所以你只能挑点又值钱又轻的东西装。这就是 **背包问题**啦♪(^∇^\*)

传说中有背包九讲，今天就讲一讲基础的**01背包和完全背包**。感兴趣的同学可以搜索**《背包九讲》**

**01背包：**

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。

有人肯定会觉得，我算一下物品的价值和重量之比选择最优的不就好了吗。曾经单纯的我也是这样想的，**兰鹅**，两个次优的物品价值相加有可能比最优的要好呀。让我来**举个栗子**，现在有一个可以装103的背包，有3件物品，第一件呢是重100值100，剩下的两件分别重78值77，重25值24。如果要选择最优的，价值只能达到100，但是不选最优的选剩下两件就可以达到101了。所以像这样前面的选择会影响到后面的选择的题目，我们就要用**动态规划**的思想做。用前一种状态进而推出后一种状态的式子我们叫做**状态转移方程**。

01背包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。“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。【这时候我们先忘记这个牵扯了前i-1件物品的包是怎么得到的，假设下面我们用到的f的数值都已知】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]。

伪代码如下：【当然，这个伪代码还可以进行优化，自己思考一下吧】

**for i=1..N**

**for v=V..0**

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

我们还需要注意一下01背包的**初始化问题**：

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。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。

是不是觉得不太难理解，那我们悄悄变个形来看看完全背包。

**完全背包**

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。

和01背包的差别很小，只是现在的物品可以无限次取了。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}**

但是这个做法复杂度要超过O（VN）了，如果时间要求不高，你又有点懒这样做也没问题。让我们来看看能不能优化一下。

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。

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。虽然没有改进复杂度，但是至少想起来思路很简单啊。

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。 这里呢就涉及到了**二进制的思想**啦。用0 和1 能够表示所有的数，所以这样的转化也能表示所有的可能啦。

这里我们还有一种O（VN）的优化。变化这么多，是不是很有意思。

**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01背包的伪代码只有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01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。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。

OK，今天的背包问题就介绍到这里。更多有意思的算法敬请关注JNUACMer联盟。