完全背包问题

题目

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 w[i] ，价值是 v[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][j] 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][j]=max(f[i-1][j],f[i-1][j-k*w[i]]+k*v[i]); //0 <= k*w[i] <= j
```

数据

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行两个整数 w_i, v_i，用空格隔开，分别表示第 i 种物品的体积和价值。

例如输入

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

输出

10

In [1]:

```
#include <fstream>
#include <iostream>
#include <stdlib.h>
using namespace std;
class beibao01 {
    public:
        int getResult() {
            int MAXN = 10000;
            int w[MAXN];    // 体积
            int v[MAXN];    // 价值
            int f[MAXN];    // f[i][j], j体积下前i个物品的最大价值
            try{
                std::ios::sync_with_stdio(false);
                ifstream inFile("dp04beibao02_01.in", ios::in);
                int n;  //物品数量
                int m;  //背包体积
                inFile >> n >> m;
                for(int i = 1; i <= n; ++i)   //读入每一个物品的体积和价值
                    inFile >> w[i] >> v[i];

                for(int i = 1; i <= n; ++i){   //计算每一个物品
                    for(int j = m; j>=0; --j){
                        int amount = m / w[i];  // 物品最多能选的次数
                        for(int k = 1; k <= amount; k++){ // 枚举选择当前物品的个数
                            if(j>=k*w[i])
                                f[j] = max(f[j], f[j-k*w[i]]+k*v[i]);   // 状态转移方程
                        }
                    }
                }
                inFile.close();
                return f[m];
            }catch(...){  //表示捕获所有异常
                cerr<<"Error!";
                return -1;   //计算过程出错，返回-1
            }
        }
};
```

Out[1]:

In [2]:

```
beibao01* bb01 = new beibao01();  //这种方式可以动态的申请和释放内存
bb01->getResult();
```

Out[2]: (int) 20

这跟 01 背包问题一样有$O(VN)$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][j] 的时间是$O\left(\frac{V}{w[i]}\right)$，总的复杂度可以认为是$O\left(N \times \sum\left(\frac{V}{w[i]}\right)\right)$，是比较大的。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 w[i]<=w[j] 且 v[i]>=v[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的 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。

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$\frac{V}{w[i]}$件，于是可以把第i种物品转化为$\frac{V}{w[i]}$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$w[i]*2^k$、价值为$v[i]*2^k$的若干件物品，其中k满足$w[i]*2^k <= V$。这是二进制的思想（为什么是二进制思想？），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$O\left(\log\left(\frac{V}{w[i]}\right)\right)$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)

这个算法使用一维数组，先看代码：

In [3]:
```
#include <fstream>
#include <iostream>
#include <stdlib.h>
using namespace std;
class beibao02 {
    public:
        int getResult() {
            int MAXN = 10000;
            int w[MAXN];    // 重量
            int v[MAXN];    // 价值
            int f[MAXN];    // f[i][j], j重量下前i个物品的最大价值
            try
            {
                ifstream inFile("dp04beibao02_01.in", ios::in);
                int n;  //物品数量
                int m;  //背包重量
                inFile >> n >> m;
                for(int i = 1; i <= n; ++i)   //读入每一个物品的重量和价值
                    inFile >> w[i] >> v[i];

                for(int i = 1;i<=n;i++){
                    for(int j = w[i];j <= m ;j++){
                        f[j] = max(f[j] , f[j-w[i]]+v[i]);
                    }
                }

                inFile.close();
                return f[m];

            }catch(...){  //表示捕获所有异常
                cerr<<"Error!";
                return -1;   //计算过程出错，返回-1
            }
        }
};
```

Out[3]:

In [4]:
```
beibao02* bb02 = new beibao02();  //这种方式可以动态的申请和释放内存
bb02->getResult();
```

Out[4]: (int) 20

细心的读者会发现，这个代码与 01 背包的代码只有 j 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 背包中要按照$j = V \cdots 0$的逆序来循环。这是因为要保证第 i 次循环中的状态 f[i][j] 是由状态 f[i-1][j-w[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][j-w[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][j-w[i]] ，所以就可以并且必须采用$j = 0 \cdots V$的顺序循环。这就是这个简单的程序为何成立的道理。值得一提的是，上面的伪代码中两层 for 循环的次序可以颠倒。这个结论有可能会带来算法时间常数上的优化。 这个算法也可以以另外的思路得出。例如，将基本思路中求解 f[i][j-w[i]] 的状态转移方程显式地写出来，代入原方程中，会发现该方程可以等价地变形成这种形式：

$$f[i][j] = max(f[i-1][j], f[i][j-w[i]] + v[i])$$

小结

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。

例题：

HDU 1114 Piggy-Bank (http://acm.hdu.edu.cn/showproblem.php?pid=1114)
Luogu 1853 投资的最大效益 (https://www.luogu.org/problemnew/show/P1853#sub)

背包问题的倒序枚举与正序枚举

先把各个变量列出来

体积为V的背包，有n个物品，每个物品的体积为w_i，价值为v_i，每个物品装一次，求最大价值。

先给出二维的转移方程

```
f[i][j]=max(f[i-1][j],f[i-1][j-w[i]]+v[i])
```

首先，对于二维数组的背包来说，正序和逆序是无所谓的，因为你把状态都保存了下来，而一维数组的背包是会覆盖之前的状态的

要想知道$f[i][j]$，你要从$f[i-1][j]$和$f[i][j-w[i]]+v[i]$两个状态转移而来，这两个状态可以直接从二维数组中取出

一维数组的转移方程

```
f[j]=max(f[j],f[j-w[i]]+v[i])
```

$f[j]$表示在执行i次循环后(此时已经处理i个物品)，前i个物体放到容量j的背包时的最大价值，即之前的$f[i][j]$。与二维相比较，它把第一维压去了，但是二者表达的含义是相同的，只不过$f[j]$一直在重复使用，所以，也会出现第i次循环覆盖第$i-1$次循环的结果。

按方程来说，其中有许多对应相等的关系，比如$f[i-1][j]$和$f[j]$就是相等的，这是为什么呢？求一下这几个值就好了

1. 前$i-1$个物品放到容量j 的背包中带来的收益($f[i-1][j]$)</br> 由于在执行第i次循环时，$f[j]$存储的是前i个物体放到容量为j的背包时的最大价值，在求前i个物体放到容量j时的最大价值(即之前的$f[i][j]$时，我们正在执行第i次循环，$f[v]$的值还是在第$i-1$次循环时存下的值，在此时取出的$f[j]$就是前$i-1$个物体放到容量j的背包时的最大价值，即$f[i-1][j]$。
2. 前$i-1$件物品放到容量为$j-w[i]$的背包中带来的收益($f[i][j-w[i]]+v[i]$)</br> 由于在执行第i次循环前，$f[0 \cdots V]$中保存的是第$i-1$次循环的结果，即是前$i-1$个物体分别放到容量$0 \ldots V$时的最大价值，即$f[i-1][0 \cdots V]$，则在执行第i次循环前，f数组中$j-w[i]$的位置存储就是我们要找的前$i-1$件物品放到容量为$j-w[i]$的背包中带来的收益(即之前的$f[i][j-w[i]]$)。

具体来说，由于在执行j时，是还没执行到$j-w[i]$的，因此，$f[j-w[i]]$保存的还是第$i-1$次循环的结果。即在执行第i次循环且背包容量为j时，此时的$f[j]$存储的是$f[i-1][j]$，此时$f[j-w[i]]$存储的是$f[i][j-w[i]]$。

相反，如果在执行第i次循环时，背包容量按照$0 \cdots V$的顺序遍历一遍，来检测第i件物品是否能放。此时在执行第i次循环且背包容量为j时，此时的$f[j]$存储的是$f[i-1][j]$，但是，此时$f[j-w[i]]$存储的是$f[i][j-w[i]]$。

因为$j > j-w[i]$，所以第i次循环中，执行背包容量为j时，容量为$j-w[i]$的背包已经计算过了，即$f[j-w[i]]$中存储的是$f[i][j-w[i]]$。它会从一开始就装入某个物品，只是为了价值最大，重复是肯定要存在的。即对于01背包，按照增序枚举背包容量是不对的。如下图

逆序枚举背包

背包容量	0	1	2	3	4	5	
插入物品1(重量3，收益5)	0	0	0	5	5	5	$i=1$时的f数组
插入物品2(重量2，收益10)	0	0	0	5	5	15	$i=2$时的f数组
	枚举背包时先处理大容量，后小容量。则这五种背包容量均还未检测是否放入物品2，数组中这部分值为插入物品1时留下的结果					正在检测是否放入物品2	

我们现在要求 `i=2` 时的 `f[5]`： 蓝色的为数组现在存储的值，这些值是 `i=1` 时(上一次循环)存入数组 `f` 的。相当于 `f[i-1][j]` 。

而黄色的是我们要求的值，在求 `f[5]` 之前， `f[5]=5` ，即 `f[i-1][5]=5` ， 现在要求 `i=2` 时的 `f[5]=f[5-2]+10=5+10=15>f[i-1][5]=5` 故 `f[5]=15` ；

要注意在求 `f[j]` 时，它引用的 `f[j-w[i]]` 和 `f[j]` 都是上一次循环的结果。

顺序枚举背包

背包容量	0	1	2	3	4	5	
插入物品1(重量3，收益5)	0	0	0	5	5	5	$i=1$时的f数组
插入物品2(重量2，收益10)	0	0	10	10	20	20	$i=2$时的f数组
			放入物品2一次	放入物品2一次	放入物品2两次		
	枚举背包时先处理小容量，后大容量。则这五种背包容量均已经检测是否放入物品2，数组中这部分值为检测是否放入物品2后留下的结果					正在检测是否放入物品2	

我们现在要求 `i=2` 时的 `f[5]`

蓝色为数组现在存储的值，这些值是 `i=2` 时(本次循环)存入数组 `f` 的。相当于 `f[i][j]` 。这是由于，我们是增序遍历数组 `f` 的，在求 `f[j]` 时， `j` 之前的值 `(0 ... v-1)` 都已经在第 `i` 次循环中求出。

黄色是我们要求的值，在求 `f[5]` 之前， `f[5]=5` ，即 `f[i-1][5]=5` 现在要求 `i=2` 时的 `f[5]=f[5-2]+10=10+10=20>f[i-1][5]=5` ，故 `f[5]=20` ；

其中引用的 `f[3]` 是 `f[i][3]` 而不是 `f[i-1][3]` 注意一点，在求 `f[j]` 时，它引用的 `f[j-w[i]]` 是本次循环的结果，而 `f[j]` 是上一次循环的结果。

另一方面

在检测背包容量为 `5` 时，看物品 `2` 是否加入

由状态转移方程可知，我们的 `f[5]` 需要引用自己本身和 `f[3]`

由于背包容量为 `3` 时，可以装入物品 `2` ，且收益比之前的大，所以放入背包了。

在检测 `f[5]` 时，肯定要加上物品 `2` 的收益，而 `f[5]` 在引用 `f[3]` 时， `f[3]` 时已经加过一次物品 `2` ，因此，在枚举背包容量时，物品 `2` 加入了多次。

然后我们明确三个问题

1. `j>j-w[i]`
2. 状态 `f[i][j]` 是由 `f[i-1][j]` 和 `f[i][j-w[i]]` 两个状态决定的
3. 对于物品 `i` ，我们在枚举背包容量时，只要背包容量能装下物品 i 且收益比原来的大，就会成功放入物品 i

具体来说，枚举背包容量时，是以递增的顺序的话，由于 `v>j-w[i]` ,则会先计算 `j-w[i]` 。在背包容量为 `j-w[i]` 时，一旦装入了物品 `i` ，由于求 `f[j]` 需要使用 `f[i-1][j-w[i]]` ，而若求 `f[j]` 时也可以装入物品 `i` 的话，那么在背包容量为 `v` 时，容量为 `j` 的背包就装入可两次物品。又若 `j-w[i]` 是由之前的状态推出，它们也成功装入物品 `i` 的话，那么容量为 `j` 的背包就装入了多次物品 `i` 了。 此时，在计算 `f[j]` 时，已经把物品 `i` 能装入的全装入容量为 `j` 的背包了，此时装入物品 `i` 的次数一定是最大的 所以，顺序枚举容量是完全背包问题最简捷的解决方案。

In [ ]:

```

```