**【每日算法DP】背包九讲（其三）多重背包问题**

今天学习多重背包问题。

其实多重背包问题与完全背包问题有点类似，只是完全背包问题每件物品都能够取无限多个，而多重背包问题**每件物品件数都有上限**。

**问题描述：**

有n种物品和一个容量是m的背包。

第i种物品最多有si件，每件体积是vi，价值是wi。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大价值。

**输入格式：**

第一行两个整数，n，m，用空格隔开，分别表示物品种数和背包容积。

接下来有 n行，每行三个整数 vi,wi,si，用空格隔开，分别表示第i种物品的体积、价值和数量。

**输出格式：**

输出一个整数，表示最大价值。

**数据范围**

0<n,m≤100

0<vi,wi,si≤100

**输入样例**

4 5

1 2 3

2 4 1

3 4 3

4 5 2

**输出样例：**

10

在上次完全背包问题我们讲到，一般来说我们从01背包问题能够想到以下状态转移方程：

f[j]=max(f[j],f[j-k\*v[i]]+k\*w[i])(k∈N,k>=0)

很容易得到：

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

{

 for(int j=m;j>=v[i];j--)

{

for(int k=0;k<=s[i]&&k\*v[i]<=j;k++)

{

f[j]=max(f[j],f[j-k\*v[i]]+k\*w[i]);

 }

 }

}

不过很显然，这段代码时间复杂度较高，如何优化时间复杂度呢?

考虑之前的完全背包问题，我们可以**将多重背包问题转化为01背包问题**。

**一个很简单的想法**

第i件物品有s件，因此我们可以**将其拆分成s[i]件价值为w[i]的物品**，这样的话，我们只需要考虑现在的每件物品需不需要拿，即现在每件物品只有拿与不拿两种选择，就变成了01背包问题。

但是

在数据较小的时候我们可以这么考虑，但当其数据太大时，比如现在有1000种物品，每种物品最多有1000件，这样的话我们转化成的01背包问题物品数将达到10^6太多了。

事实上我们没必要将第i件物品拆分为s[i]件。

**采用二进制拆分**

对于任意十进制数，我们都能用1、2、4、8…来表示。

比如：

8=8

15=1+2+4+8

23=1+2+4+16

38=2+4+32

......

因此，如果按照二进制拆分的话，我们就能将物品拆分为logn件。

**但是这其中有个问题**

比如如果现在有种物品有12件，按照上面的想法，我们可以用1、2、4、8来表示，但是用1、2、4、8来表示的话，又多了13、14、15，这样的话会衍生出13、14、15件该物品，显然这是不对的。

因此我们可以改进一下，最后一位数我们采用件数减去前面所有数的和。

比如：

12我们采用1、2、4、5来表示，其中5是由12-1-2-4得来的。

这样的话我们拆分便没有问题了。

这种二进制优化方式在编程里面是很常见的，需掌握。

具体代码：

//#include <bits/stdc++.h>

#include <cstring>

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

const int MAX = 2010;

int f[MAX];

int v,w,s;

struct Good

{

       int v,w;

};

int main()

{

int n,m;

vector<Good> goods;//用来存拆分的物品

cin>>n>>m;

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

{

       cin>>v>>w>>s;

for(int j=1;j<=s;j\*=2)

{

s-=j;

goods.push\_back({j\*v,j\*w});//将拆分完的物品的新体积与新价值存入goods

}

if(s>0)    goods.push\_back({s\*v,s\*w});//以免拆分完超过原来的件数

}

for(auto good: goods)

{

for(int j=m;j>=good.v;j--)

{

f[j]=max(f[j],f[j-good.v]+good.w);

}

}

/\*     for(vector<Good>::iterator it=goods.begin();it!=goods.end();it++)

{

for(int j=m;j>=(\*it).v;j--){            f[j]=max(f[j],f[j-(\*it).v]+(\*it).w);        }

}

 \*/

 cout<<f[m]<<endl;

 return 0;

}

二进制拆分能够有效地降低时间复杂度，但是当数据继续加大时，二进制拆分也显不足。

**单调队列优化**

多重背包问题的究极版是单调队列优化，将问题化为经典的单调队列问题。

单调队列问题可参考LeetCode 239题。

我看不懂，等我后面学习经典单调队列问题再回头来看。

**相关问题：**

[【每日算法DP】背包九讲(其一)0-1背包问题](https://mp.weixin.qq.com/s?__biz=MzI3OTA0NzU1Mg==&mid=2247483680&idx=1&sn=04dd8db5ad1539c489543844062bec9e&chksm=eb4cfdcfdc3b74d9a5e3623c1dabd04ccb572edce09cc1085babed9639d6f2aa64ea9a9f3afe&token=918147061&lang=zh_CN&scene=21" \l "wechat_redirect)

[【每日算法DP】01背包问题为什么要将背包的容量逆序？](https://mp.weixin.qq.com/s?__biz=MzI3OTA0NzU1Mg==&mid=2247483687&idx=1&sn=d2388bfc0f7dad744856e7f78c5fcdfa&chksm=eb4cfdc8dc3b74de2ccba49f849d13efa55922213e8bdd65b4b406d55e5f58fd4488d3d9bf0f&token=918147061&lang=zh_CN&scene=21" \l "wechat_redirect)

[【每日算法DP】背包九讲（其二）完全背包问题](https://mp.weixin.qq.com/s?__biz=MzI3OTA0NzU1Mg==&mid=2247483695&idx=1&sn=57f7c054a3274d5f737a70fb0ef25c9d&chksm=eb4cfdc0dc3b74d6f308d2af6a88e6d61268ef127a47c301a07414a69811ab74c38923e54ed1&token=918147061&lang=zh_CN&scene=21" \l "wechat_redirect)



算法一推

慢