# 一、01背包理解

先让我把01背包的题。和输入输出用例写出来把

## 题目先写上

### 1、题目详解

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v[i]，价值是 w[i]。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

### 2、输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v[i],w[i]，用空格隔开，分别表示第 ii 件物品的体积和价值。

### 3、输出格式

输出一个整数，表示最大价值。

### 4、数据范围

$$0 < N,V \leq 10000 < N,V \leq 10000 < vi,wi \leq 1000$$

### 5、输入和输出样例

**输入**

| 输入 | 输出 |
| --- | --- |
| 4 | 5 |
| 1 | 2 |
| 2 | 4 |
| 3 | 4 |
| 4 | 5 |

**输出**

$$8$$

## 暴力DFS方法讲解01 背包

我们都知道DFS基础，就是利用递归暴力搜索，把所有的可能性的列出来，那么对于这个题 我们要用 什么样的方法进行爆搜呢。

很显然是，每一个物品都可以放入背包，和不放入背包两种可能。那么递归基（就是递归的退出条件） 我们就是判断当放入物品使背包装满或者溢出了。那么我们就不能深入递归了。

下面我开始上代码，大家可以看代码中的注释来理解代码。

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 1010;
int n,m;
// v[i] 存储 物品的体积，w[i] 存储物品的价值
int v[N],w[N];
int res;
// u表示第u个物品 sum 表示背包内已经存放物品的体积

void dfs(int u,int sumv,int sumw)
{
    // 如果 第u 个物品已经超过或者等于 n个了，注：这里不会超过只会等于，超过只是个人习惯，怕出错。当等于n个的时候，
    // 我们判断 背包中所有物品的体积是不是 能够放入背包，如果可以放入那么刷新 res 结果.w
    if(u >= n)
    {
        if(sumv <= m)
```

```
            {
                res = max(res,sumw);
            }
            return;
        }
        // 如果 物品的体积 大于等于 背包的容量时 我们判断 如果等于就刷新res
        if(sumv >= m)
        {
            if(sumv == m)
            {
                res = max(res,sumw);
            }
            return;
        }
        // 选当前物品
        dfs(u + 1,sumv + v[u],sumw + w[u]);
        // 不选择当前物品
        dfs(u + 1,sumv,sumw);
    }
    int main()
    {
        cin >> n >> m;
        for(int i = 0; i < n; i++) cin >> v[i] >> w[i];
        dfs(0,0,0);
        cout << res << endl;
        return 0;
    }
```

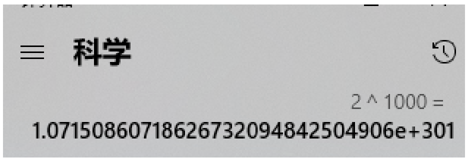
大家看注释应该时可以理解我的意思的。下面我们看在 acwing 上的测试结果吧。


恭喜恭喜 。骗到了六个分，TLE了。那么为什么TLE了呢？TLE就是超时，我们超时了，就有两种可能，死递归和时间复杂度太高了。

那么我们看这个题，通过了六个，那么应该不是死递归。所以就是时间复杂度太高了。

那么我们开始判断，为什么时间复杂度会高呢。

我们可以这样想，每一个物品都有 两种可能，那么有n个物品的时间复杂度就是 n个物品的两种可能相乘。也就是O(2^n) 一千个数据。2的一千次方




这个吧 应该就是1后面三百个0的时间复杂度了，太猛了。王思聪的电脑搬过来都不咋行啊。

## 科普

现在测题的oj 一般一秒运行 1e8次-1e9次的样子。

所以我们复杂度太高了，换种方法

# 二维DP解决01背包问题

既然是二维数组了，那么肯定有q[i][j]

**二维数组的含义：**

这里的$dp[i][j]$; $i$表示从$0$到$i$中所有物品随便取（每种都有取或者不取两种可能）放进容量为$j$得背包里（这里是要确保能放开）。得到的最大价值为$dp$

**记住这个含义，下面的介绍都会围绕这个含义展开哦**

| dp[i][j] | 0 | 1 | 2 | 3 | 4 | 这一行表示的是背包容量的可能 从最小到最大 | | | |
|---|---|---|---|---|---|---|---|---|---|
| 物品0 | | | | | | | | | |
| 物品1 | | | | | | | | | |
| 物品2 | | | | | | | | | |
| 这里是遍历一下所有物品，也就是所有物品都放一下 | | | | | | | | | |

如果当你看到这个表就懂了 那么你就想一下i和j代表的含义哦

### 确定递推公式

我们首先回忆一下。i和j的含义哦

这里的$dp[i][j]; i$表示从0到$i$中所有物品随便取（每种都有取或者不取两种可能）放进容量为$j$得背包里（这里是要确保能放开）。得到的最大价值为$dp$

那么我们有两个方向可以推出来 dp【i】【j】

**不放物品**：那么就是 dp【i】【j】 = dp【i- 1】【 j 】因为我们i表示从0 到i随机选，那么不选i 自然就是从0 到 i - 1 随便选咯

**放物品**： 那么就是能不能放开了。不能放开就是和上面的不放物品一致，放物品 那么就要从上一个物品放入之后到j所剩余的空间能够放开当前物品i才行。又因为dp【i】【j】 表示的是最大的价值，所以我们要将上一个容量距离 **j** 差距为当前物品时的价值加上当前 **i** 物品的价值；那么递推公式就出来了。

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

### 如何初始化。

我们的递推公式可以看出来，我们的dp【i】【j] 不是从**i - 1** 来就是从 上一行的左边来，那么最初的就必然时从最上面一排或者最左边一排出发的，所以我们将dp【0】【i】 和 dp【i】【0】赋值为0即可

### 代码

```
#include<iostream>
using namespace std;

const int N = 1010;
int n, m;
int dp[N][N],v[N],w[N];

int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i++) cin >> v[i] >> w[i];
    // 初始化 我们这里定义的是全局变量已经默认为0 不需要初始化了就  我们便于理解所以初始化
    for(int i = 0; i < n; i++) dp[0][i] = dp[i][0]= 0;

    for(int i = 0; i < n ;i ++)
    {
        for(int j = 0; j <= m; j++)
        {
            if(j < v[i])
            {
                dp[i][j] = dp[i - 1][j];
            }
            else
            {
                dp[i][j] = max(dp[i - 1][j],dp[i - 1][j - v[i]]+w[i]);
            }
        }
     }
     // 这里要特别说明一下，最右下角必然是 最大值，
     cout << dp[n - 1][m] << endl;
    return 0;
}
```


同样的代码，那么就让读者帮我修改错误吧，我真心不知道哪里有问题哎！！

### 理解遍历

| dp[i][j] | 0 | 1 | 2 | 3 | 4 | 这一行表示的是背包容量的可能 从最小到最大 |
|---|---|---|---|---|---|---|
| 物品0 | 0 | 2 | 2 | 2 | 2 | |
| 物品1 | 0 | 0 | 0 | 0 | 0 | |
| 物品2 | 0 | 0 | 0 | 0 | 0 | |
| 这里是遍历一下所有物品，也就是所有物品都放一下 | | | | | | |

看代码我们可以知道，是一个二维遍历，那么我们交换遍历顺序有问题吗?

大家想哈，我们是左上角，我们如果交换一下顺序，那么就是沿着对角线对称一下，还是从左上角出发，所以理论上没有影响的。这个你们实际操作一下猜猜看，

二维遍历就讲到这里了！下面我们开始看以为遍历

## 一维DP解决01背包问题

首先我说一下一维代替二维的思想，其实就是将，上一行的数据代替当前行的数据 并进行递推，

```
#### 确定dp数组的定义
```

dp【i】 表示 背包容量为 i的背包最大的所容纳的最大物品的体积为dp【i】

**确定递推公式**

首先我们如何将上一行的数据 递推到这一行呢? 这个别想了，越想越复杂其实就是直接用dp【i】表示 也别修改， 就是上一行遍历的数据嘛。

那么开始我们判断递推公式了：首先，我们dp【j】是由 dp【j - v[i]】推导来的，也就是

$$dp【j】 = dp【j - v【i】】 + W[i];$$

那么我们汇总一下上面两句话说的就出来了。递推公式了，就是上一行的结果和当前行的结果取一个最大值嘛

所以递推公式为:

$$dp[j] = max(dp[j], dp[j - v[i]] + w[i]);$$

### 初始化

全为0 别想那么多有的没的

### 代码

我写的代码会有注释 也可以帮助大家理解

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 1010;
int n,m;
int dp[N],v[N],w[N];
```

```
int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i++) cin >> v[i] >> w[i];

    for(int i = 0; i < n; i++)
    {
        // 这里为什么要用倒叙呢？看完代码我会再后文中解释
        for(int j = m; j >= v[i]; j--)
        {
            // 这里不用判断的原因是因为放不开就不放了嘛。
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
    }

    printf("%d",dp[m]);

    return 0;
}
```

看完代码 应该都能理解了吧?

### 为什么代码中遍历背包容量的时候为什么要倒叙呢?

首先我们假如 正序

```
i = 0时
dp[0] = 0;  因为放不开
dp[1] = max(dp[1] , dp[1 - 1] + 2);
dp[1 - 1] + 2 = 0 + 2 = 2
dp[2] = max(dp[2],dp[2-1] + 2);
dp[2-1] + 2 = 2 + 2 = 4
我们会发现 2 重复加了
那么我们倒叙怎么来呢?

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

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

因为前面的值其实都没变，所以我们用的时候前面的值其实还是上一行的值，
```

到这里我们的01背包希望大家都理解了。这里是希望 我可没说都理解了熬，因为我理解了好多天呢。才理解到现在的地步大家加油吧

# 二、完全背包问题

## 题目先写上

### 1、题目详解

有 N 种物品和一个容量是 V 的背包，每种物品都有无限件可用。

第 ii 种物品的体积是 vi，价值是 wi。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

### 2、输入格式

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 vi,wi，用空格隔开，分别表示第 ii 种物品的体积和价值。

### 3、输出格式

输出一个整数，表示最大价值。

### 4、数据范围

$$0 < N, V \leq 10000 < vi, wi \leq 1000$$

## 5、输入和输出样例

```
输入
4 5
1 2
2 4
3 4
4 5

输出
10
```

# 介绍解法一 一维的方法

在01背包中我们说到，每个物品只能用一次不能用多次，所以要用倒叙，而正序则是每一个都会出现重复的情况。所以我们这个题也就有了解决方法

下面喊我列举一些用例来模拟熬

```
首先dp数组初始化全为0：给定物品种类有4种，包最大体积为5，dp定义为全局时默认为0所以这里不需要初始化
v[1] = 1, w[1] = 2
v[2] = 2, w[2] = 4
v[3] = 3, w[3] = 4
v[4] = 4, w[4] = 5

i = 1 时： j从v[1]到5
dp[1] = max(dp[1],dp[0]+w[1]) = w[1] = 2 (用了一件物品1)
dp[2] = max(dp[2],dp[1]+w[1]) = w[1] + w[1] = 4（用了两件物品1）
dp[3] = max(dp[3],dp[2]+w[1]) = w[1] + w[1] + w[1] = 6（用了三件物品1）
dp[4] = max(dp[4],dp[3]+w[1]) = w[1] + w[1] + w[1] + w[1] = 8（用了四件物品1）
dp[5] = max(dp[3],dp[2]+w[1]) = w[1] + w[1] + w[1] + w[1] + w[1] = 10（用了五件物品）

i = 2 时：j从v[2]到5
dp[2] = max(dp[2],dp[0]+w[2]) = w[1] + w[1] = w[2] =  4（用了两件物品1或者一件物品2）
dp[3] = max(dp[3],dp[1]+w[2]) = 3 * w[1] = w[1] + w[2] =  6（用了三件物品1，或者一件物品1和一件物品2）
dp[4] = max(dp[4],dp[2]+w[2]) = 4 * w[1] = dp[2] + w[2] =  8（用了四件物品1或者，两件物品1和一件物品2或两件物品2）
dp[5] = max(dp[5],dp[3]+w[2]) = 5 * w[1] = dp[3] + w[2] =  10（用了五件物品1或者，三件物品1和一件物品2或一件物品1和两件物品2）

i = 3时：j从v[3]到5
dp[3] = max(dp[3],dp[0]+w[3]) = dp[3] = 6  保持第二轮的状态
dp[4] = max(dp[4],dp[1]+w[3]) = dp[4] = 8  保持第二轮的状态
dp[5] = max(dp[5],dp[2]+w[3]) = dp[4] = 10 保持第二轮的状态

i = 4时：j从v[4]到5
dp[4] = max(dp[4],dp[0]+w[4]) = dp[4] = 10  保持第三轮的状态
dp[5] = max(dp[5],dp[1]+w[4]) = dp[5] = 10  保持第三轮的状态

上面模拟了完全背包的全部过程，也可以看出，最后一轮的dp[m]即为最终的返回结果。
```

**这里强调一下 一维其实就是二维直接上一行该列的dp结果**

所以我们可以很简单的写出代码来

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 1010;
int n,m;
int dp[N],v[N],w[N];
int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i ++) cin >> v[i] >> w[i];
    for(int i = 0; i < n; i ++)
    {
        for(int j = v[i]; j <= m; j ++)
        {
            dp[j] = max(dp[j],dp[j - v[i]] + w[i]);
        }
    }
    printf("%d",dp[m]);
    return 0;
 }
```

看完01背包 这个就会很容易理解的

## 介绍解法 --- 二维的方法

这里我们先回忆一下，01背包，每一个都放一下试试，，然后直到放满了才更新dp

我们多重背包就可以考虑 利用每一个都可以最多放多少个进行遍历一下（这里会TLE 但是我们通过下面代码理解一下该题，后面我再优化）

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 1010;
int dp[N][N],v[N],w[N];
int n,m;
int main()
{
    cin >> n >> m;
    for(int i = 1; i < n; i++) cin >> v[i] >> w[i];
    for(int i = 1; i < n; i++) // 遍历所有的物品
    {
        for(int j = 0; j <= m; j++) //遍历背包的容量的可能性
         {
            for(int k = 0; k * v[i] <= j; k++)
            {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - k * v[i]] + k * w[i]);
            }
         }
    }
    cout << dp[n - 1][m];
    return 0;
}
```

我们算一下时间 假如 n 和 m 都是1000 你们 k的值也可能是一千了，因为他是根据v[i] 和 j 的可能来推导的 也就是1000 * 1000 * 1000 = 1e9 必超时

下面我们看一下优化后的

```
#include<iostream>
using namespace std;

const int N = 1010;

int n, m;
int dp[N][N], v[N], w[N];

int main(){
    cin >> n >> m;
    for(int i = 1; i <= n; i ++ ){
        int v, w;
        cin >> v >> w;
        for(int j = 0; j <= m; j ++ ){
            // 把上一行的拷贝过来  下次再次使用的时候已经重复使用了上一次的了
            dp[i][j] = dp[i - 1][j];
            if(j >= v)
                dp[i][j] = max(dp[i][j], dp[i][j - v] + w);
        }
    }
    cout << dp[n][m] << endl;
}
```

其实细心的同学会看到，这个二维和一维的用的逻辑是一样的，都是将上一行的结果直接拷贝到这一行来，这样的话实现每一个物品的多次使用

# 三、多重背包问题

# 题目先写上

### 1、题目详解

有 N 种物品和一个容量是 V 的背包。

第 ii 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，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，每行三个整数 vi,wi,si，用空格隔开，分别表示第 ii 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

$$0 < N, V \leq 1000 < vi, wi, si \leq 100$$

## 输入输出样例

```
输入
4 5
1 2 3
2 4 1
3 4 3
4 5 2

输出
10
```

经过题意我们可以发现，其实就是固定了每种物品的固定多少个。我们可以直接想到 把他们直接当成01背包来做，差距也就只是几个重复i嘛，

这样一想就会很简单了 下面我开始上代码

## 暴力拆分+01背包做法

```
#include<iostream>
using namespace std;
const int N = 10100;
int n,m;
int v1,w1,k;
int v[N],w[N],dp[N];
int t;

int main()
{
    cin >> n >> m;
    for(int i = 0; i < n; i++)
    {
        // 暴力展开
        cin >> v1 >> w1 >> k;
        while(k --)
        {
            v[t] = v1;
            w[t ++] = w1;
        }
    }
    // 套用01 背包的思路
    for(int i = 0; i < t; i++) //遍历物品
    {
        for(int j = m; j >= v[i]; j --) // 遍历背包容量
        {
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
     }
    cout << dp[m] << endl;
    return 0;
}
```

## 节省空间模式 超简洁的代码

```
#include<iostream>
#include<algorithm>
using namespace std;
const int N = 1010;
int dp[N];

int n,m;
int v,w,k;
```

```
int main()
{
     cin >> n >> m;
    while(n --)
    {
        cin >> v >> w >> k;
        for(int i = 0; i < k; i++)
        {
            for(int j = m; j >= v; j--)
            {
                dp[j] = max(dp[j],dp[j - v] + w);
            }
        }
    }
    printf("%d",dp[m]);
    return 0;
 }
```

# 下期预告

下一个博客将写一些与背包问题有关的习题，带大家更深入理解和应用背包