# 多重背包问题：

> 有 N 种物品和一个容量为 V 的背包。第 i 种物品最多有 Mi 件可用，每件耗费的空间是 Ci，价值是 Wi。求解将哪些物品装入背包可使这些物品的耗费的空间总和不超过背包容量，且价值总和最大。

## *one* 记忆化搜索+没有任何处理：$O(V\sum M_i)$

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 104;
int n, m;
int w[maxn], v[maxn], M[maxn];
int f[maxn][maxn];
const int inf = -1e9;
int dfs(int now, int have) //正推选择前i个。
{
    //转移成规模比较小的解来得到这一个函数。

    if (now == 0)
        return 0;
    if (f[now][have] > 0)
        return f[now][have];
    int t = inf;
    for (int i = 0; i <= min(M[now], have / w[now]); i++)
        t = max(t, dfs(now - 1, have - i * w[now]) + v[now] * i);
    return f[now][have] = t;
}
int main()
{
    cin >> n >> m;
    memset(f, inf, sizeof(f));
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i] >> M[i];
    cout << dfs(n, m) << '\n';
}
```

---

## *two* 抽象出二次循环+没有优化处理：

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1003;
int v[maxn], w[maxn], M[maxn];
int n, m;
int f[maxn][maxn];
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
```

```
        cin >> w[i] >> v[i] >> M[i];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            for (int k = 0, ttt = min(M[i], j / w[i]); k <= ttt; k++)
                f[i][j] = max(f[i][j], f[i - 1][j - k * w[i]] + v[i] * k);
    cout << f[n][m] << '\n';
}
```

---

## $three$ 将空间优化到一维：+ 时间上没有什么优化处理。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1003;
int v[maxn], w[maxn], M[maxn];
int n, m;
int f[maxn];
int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> w[i] >> v[i] >> M[i];
    for (int i = 1; i <= n; i++)
        for (int j = m; j >= w[i]; j--)
            for (int k = 1, ttt = min(M[i], j / w[i]); k <= ttt; k++)
                f[j] = max(f[j], f[j - k * w[i]] + v[i] * k);
    cout << f[m] << '\n';
}
```

---

### 补充以及回顾tips：

- 由于每件物品只是在第一层的第i层的时候使用，所以不用开数组。
- 如果是一维的方式表示指标函数，那么不改变就表示了选择0个的情况的指标函数。所第三层枚举是从1开始枚举的。

## $four$ 多重背包问题优化：二进制,$O(V \times \sum logn)$

- 基本想法
  - 转化成01背包的问题，将可选第$i$件物品可以选$m_i$件，就将第i件物品放入$m_i$件。但是发现最终
  - 选择物品的数量依然是高达$\sum m_i$个复杂度依然没有优化。如果开的是二维的数组，空间复杂度反而会变得更大。
  - 二进制的思想，任何一个自然数都转化二进制，即可以拆为二的自然数倍的数相加。
    $0--M_i$件的选择可以由一系列2的整数倍来构造。对于整体的子问题的解，与物品的选择次序无关，将有关系的几件物品放在一块。
    这一部分的决策就等效为，对一件物品的数目的选择情况。（关注贡献，体会等效性。）
- 关键问题
  - 应该引入拿哪一些的物品？
    - 如果概述就是一个2的自然数次方倍，只要选择$logM_i$个就能够恰好实现对$0--M_i$数的组合。

- 问题：当物品可选择数量并不是一个整数幂的时候。选择$logM_i$向上取整个，可能组合出不合理解即比$m_i$大。
- 解决方案： 只要2的自然次方数$logm_i$向下取整个，再抽离出$M_i - maxpow2$就可用着一些数表表示所有的范围。

---

## 自己的方案

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 2003;
int f[maxn];
int main()
{
    int n, m;
    cin >> n >> m;
    vector<int> w, v, M;
    w.push_back(0);
    v.push_back(0);
    for (int i = 1; i <= n; i++)
    {
        int a, b, c;
        cin >> a >> b >> c;
        for (int j = 1; true; j *= 2)
        {
            if (c - j > 0)
            {
                w.push_back(a * j);
                v.push_back(b * j);
                c -= j;
            }
            else
            {
                w.push_back(a * c);
                v.push_back(b * c);
                break;
            }
        }
    }
    for (int i = 1, tt = v.size(); i < tt; i++)
        for (int j = m; j >= w[i]; j--)
            f[j] = max(f[j], f[j - w[i]] + v[i]);
    cout << f[m] << '\n';
}
```

## 改进

- 不用开一个存储结构来存储记录每种物品的属性，即得即用就行。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 2010;
int f[maxn];
struct good
```

```
{
    int w;
    int v;
};
int main()
{
    int n, m;
    cin >> n >> m;
    vector<good> Good;
    for (int i = 1; i <= n; i++)
    {
        int w, v, s;
        cin >> w >> v >> s;
        for (int k = 1; k < s; k *= 2) //枚举。
        {
            s -= k;
            Good.push_back({w * k, v * k});
        }
        Good.push_back({w * s, v * s});
    }
    for (auto i : Good)
        for (int j = m; j >= i.w; j--)
            f[j] = max(f[j], f[j - i.w] + i.v);
    cout << f[m] << '\n';
}
```

## $Five$ 用单调队列进一步优化。$O(VM)$

- 思路：
  - 看最基本的状态转移方程：
    $d_{i,j} = max(d_{i-1,j-k*w_i} + k * v_i), 0 <= K <= s_i$
  - 可以发现取max运算中，是对系列s+1个连续的相差w的数进行操作。联想到单调队列求一个长度确定的窗口中所有数的最大值。
  - 但是$d_{i-1,j}$并不是max的对象。考虑向下移动的过程中有规律的给它们加一个数。$s * v$这样保证前一个前一个比后一个的差值其实和理想的max的差值相同
    同时这样就可以将保证了它们的顺序关系，又可以将它们还原。

---

### 初步的方案：（因为用了stl常数比较大）tle

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1010, maxv = 20010;
int f[maxn][maxv];
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        int w, v, s;
        cin >> w >> v >> s;
```

```
        for (int tt = 0; tt < w; tt++)
        {
            deque<int> que;                                    //双端队列实现单调栈。
            for (int j = tt, k = 1; j <= m; j += w, k++) //利用k这一个变量可以做一些什么样的记录?
            {
                if (k > s + 1) //代表着曾经有多少个元素进栈了。如果超过了三个就说明一些元素要弹栈。
                {
                    //还要确认当前栈头元素是否为弹出的元素。
                    //只要弹出元素和它相等即可。
                    if (que.front() == f[i - 1][j - (s + 1) * w] - (k - (s + 1)) * v)
                        que.pop_front();
                }
                while (!que.empty() && que.back() + k * v < f[i - 1][j]) //滑动窗口对应的数组是什么?                                    //添加的过程中可以加一些处理，这样并不会影响它们的次序同时也可以还原真实量的大小
                    que.pop_back();
                que.push_back(f[i - 1][j] - k * v);
                f[i][j] = que.front() + k * v;
            }
        }
    }
    cout << f[n][m] << '\n';
}
```

## 改进角度

- 自己写一个简化的双端队列，减少常数
- 尝试用一维数组优化。

---

## 第一个优化：自己实现一个简单双端队列。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1010, maxv = 20010;
int f[maxn][maxv];
int que[maxv];
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        int w, v, s;
        cin >> w >> v >> s;
        for (int tt = 0; tt < w; tt++)
        {
            int l = 0, r = -1;                                 //双端队列实现单调队列。
            for (int j = tt, k = 1; j <= m; j += w, k++) //利用k这一个变量可以做一些什么样的记录?
```

```
            {
                if (k > s + 1) //代表着曾经有多少个元素进栈了。如果超过了三个就说明一些元素要弹栈。
                {
                    //还要确认当前栈头元素是否为弹出的元素。
                    //只要弹出元素和它相等即可。
                    if (que[l]== f[i - 1][j - (s + 1) * w] - (k - (s + 1)) * v)
                        l++;
                }
                while (l<=r && que[r] + k * v < f[i - 1][j]) //滑动窗口对应的数组是什么？                                //添加的过程中可以加一些处理，这样并不会影响它们的次序同时也可以还原真实量的大小
                    r--;
                que[++r]=(f[i - 1][j] - k * v);
                f[i][j] = que[l] + k * v;
            }
        }
    }
    cout << f[n][m] << '\n';
}
```

## 应用滚动数组优化空间复杂度：

```
#include <bits/stdc++.h>
using namespace std;
const int maxv = 20010;
int f[maxv], que[maxv], pre[maxv]; // que只记录下标即可
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        int w, v, s;
        cin >> w >> v >> s;
        memcpy(pre, f, sizeof(f)); //新的函数
        for (int tt = 0; tt < w; tt++)
        {
            int l = 0, r = -1;                //双端队列实现单调栈。
            for (int j = tt; j <= m; j += w) //利用k这一个变量可以做一些什么样的记录？
            {
                if (r >= l && que[l] < j - s * w) //该处是尽头的下标
                    l++;
                while (l <= r && pre[que[r]] - que[r] / w * v <= pre[j] - j / w * v) //滑动窗口对应的数组是什么？                                        //添加的过程中可以加一些处理，这样并不会影响它们的次序同时也可以还原真实量的大小
                    r--;
                que[++r] = j;
                f[j] = pre[que[l]] + (j - que[l]) / w * v;
            }
        }
    }
    cout << f[m] << '\n';
```

```
}
```

## 生长思考：

- 一些函数memset，memset这一类函数。
- 这里为了简洁代码补统一处理，也是正向的进行。为了防止数据丧失，只开了两个数组。
- 单调队列存储的信息是当前的元素的位置。我们可以通过这一个索引，查询元素的具体大小。就相当于存储了当前元素。