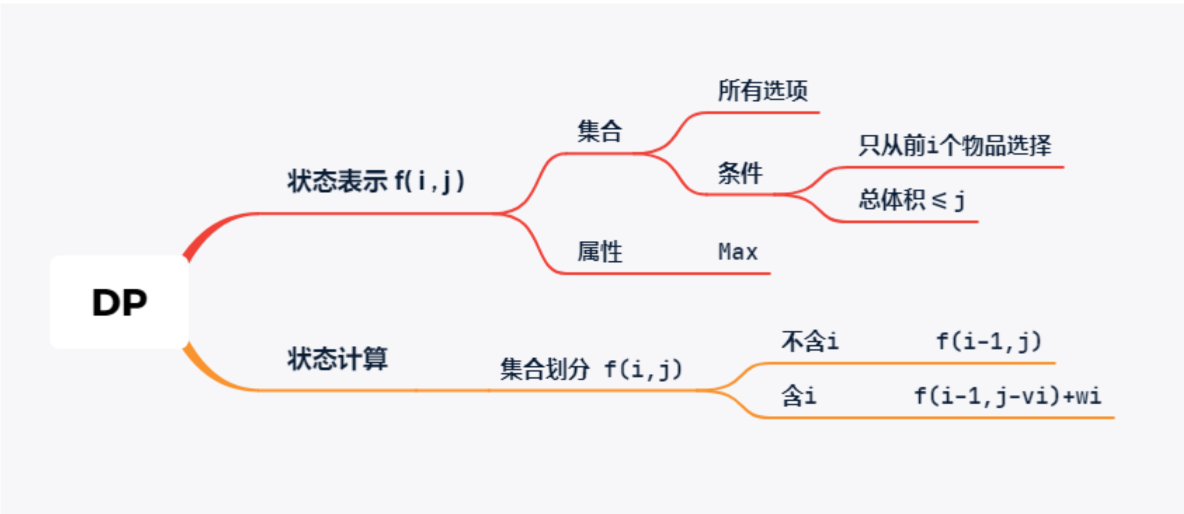
常用模型

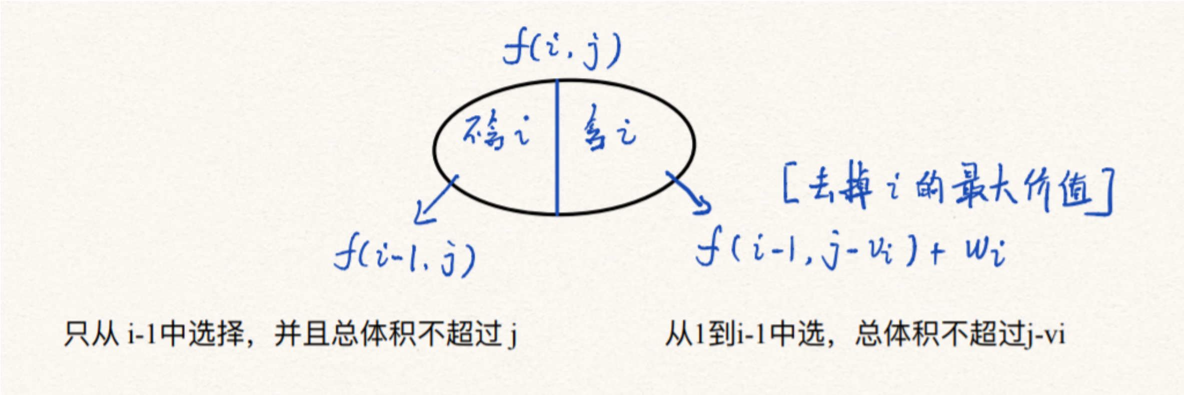
- 01背包
- 完全背包
- 多重背包
- 分组背包
- 二维费用背包
- 有依赖的背包

0-1背包

每件物品最多只用一次

> 详细描述：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 vi，价值是wi
>
> 求解将哪些物品装进背包，可使这些物品的总体积不超过背包容量，且总价值最大





对上图进行一下补充：在计算含i的状态时，我们将其分为变化的部分和不变的部分

```
变化的部分   必选部分
1. ---------- i
2. ---------- i
那么变化的部分的最大值，就是 f[i-1][j-v[i]]，在加上w[i]即为我们所求
```

代码：朴素版本　　$O(N^2)$

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1010;

int n, m; // n 表示物品数量， m表示背包容积
int v[N], w[N]; // vi表示体积，wi表示价值
int f[N][N];    // fij 表示所有方案中，从前i个物品中选并体积不超过j的最大价值

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i ++) cin >> v[i] >> w[i];
    for (int i = 1; i <= n; i ++) { // 从1个不选到n个全要
        for (int j = 1; j <= m; j ++) { // 体积从1-m
            f[i][j] = f[i-1][j]; // 不包含i的情况
            if (j >= v[i]) {
```

```
                f[i][j] = max(f[i][j], f[i-1][j-v[i]] + w[i]);
            }
        }
    }
    cout << f[n][m] << endl; // fnm 表示从前n个中选择，体积为m的最大价值
    return 0;
}
```

重点是状态转移方程：

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

代码优化思路： `i` 只用到了 `f[i]` 和 `f[i-1]` 这两层，`j` 只用到了 `j` 和 `j - vi`

我们一步一步来看（对代码进行恒等变形）：

```
for (int i = 0; i <= n; i ++) { // 从1个不选到n个全要
    for (int j = 1; j <= m; j ++) { // 体积从1-m
        f[i][j] = f[i-1][j];     // 不包含i的情况
        // 1. f[i][j] = f[i-1][j];
        // 计算顺序时先计算右边，然后再更新左边，此时右面的值还在上一层
        if (j >= v[i]) {
            f[j] = max(f[j], f[j-v[i]] + w[i]);
            // 由于我们时从小到大的枚举j，且 j - v[i] < j
            // 所以 f[j-v[i]] 一定是再 f[j] 之前被算出来的，于是等价于
            // f[i][j] = max(f[i][j], f[i][j-v[i]] + w[i]) 不符合
            // 只需要从大到小枚举j就可以解决
        }
    }
}
```

修改过后的代码：

```
for (int i = 0; i <= n; i ++)
    for (int j = m; j >= v[i]; j --)
        f[j] = max(f[j], f[j-v[i]] + w[i]);
```

注：将j循环改成由大到小之后，就可以保证，先计算 `f[j]` 而后计算的 `f[j-v[i]]` ，等价于 `f[i-1][j-v[i]]`

我们只有上一层dp值的一维数组，更新dp值只能原地滚动更改；

注意到，当我们更新索引值较大的dp值时 `dp[j]` ，需要用到上一层dp值 `dp[j - v[i]]` ，也就是说，在更新索引值较大的dp值之前，索引值较小的 `上一层dp值必须还在` ，还没被更新；

所以只能索引从大到小更新，否则会发生覆盖现象。

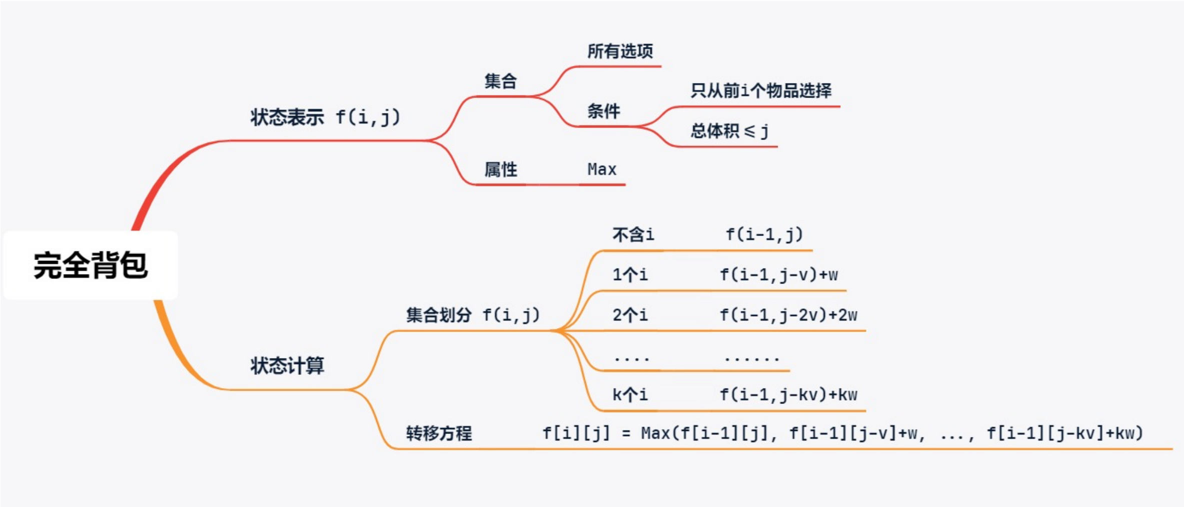
> **如果没看明白，继续看完 完全背包后的总结**

PS：一些其他小问题

> Q：yls为什么背包九讲里说：最终的答案并不一定是 `f[N][V]` ，而是 `f[N][0-V]` 的最大值呢？
>
> A：取决于状态的定义是“ `恰好使用` V的体积，还是 `最多使用` V的体积”。

完全背包

每件物品有无限个



根据这个状态转移方程，我们可以写出一个三层循环的朴素版本

```
for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= m; j ++)
        for (int k = 0; k * v[i] <= j; k ++)
            f[i][j] = max(f[i][j], f[i-1][j-k * v[i]] + k * w[i]);
```

优化的时候我们可以着重优化第三重循环

$$\begin{aligned} f(i,j) &= Max(f[i-1,j] \quad ,f[i-1,j-v]+w \quad ,f[i-1,j-2v]+2w, \quad \dots) \\ f(i,j-v) &= Max(\quad\quad\quad\quad\quad ,f[i-1,j-v] \quad\quad ,f[i-1,j-2v]+w, \quad\quad \dots) \end{aligned}$$

由此可得

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

从而写出$O(N^2)$的优化版本

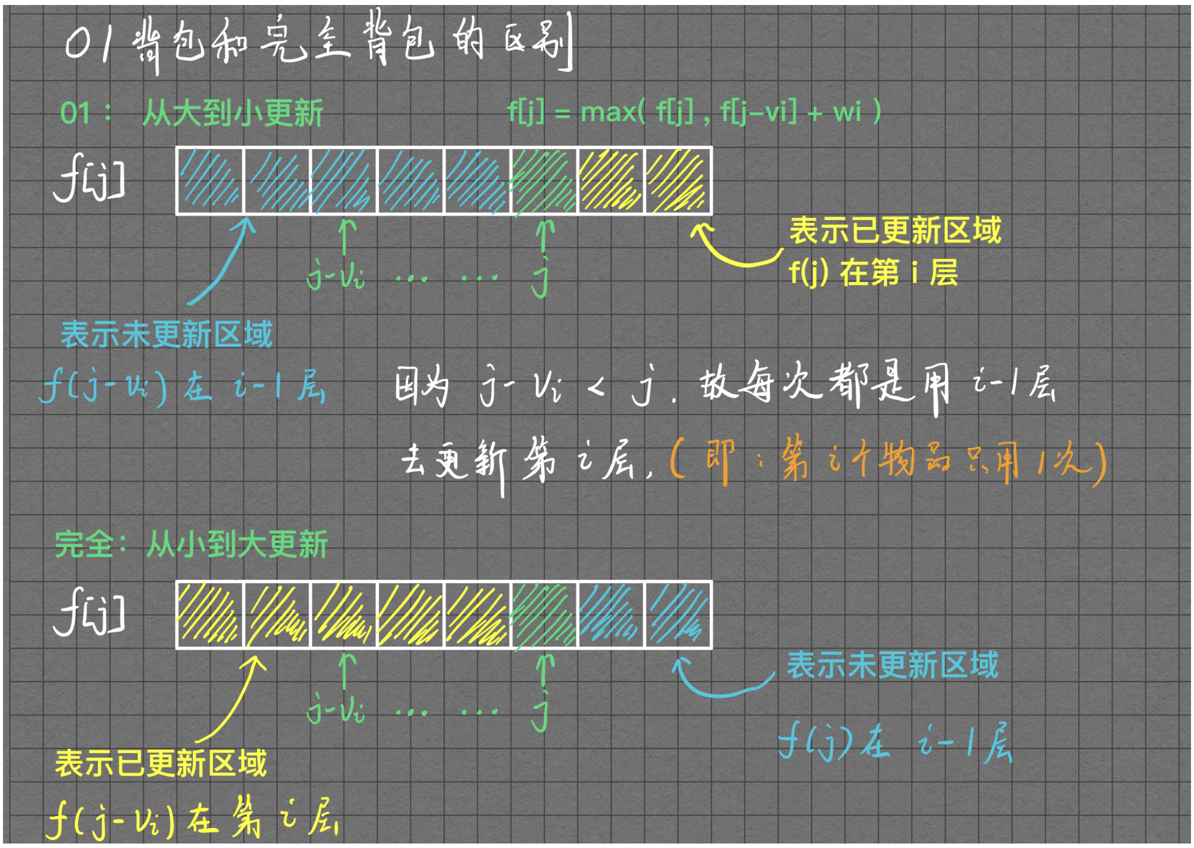
```
for (int i = 1; i <= n; i ++) {
    for (int j = 1; j <= m; j ++) {
        f[i][j] = f[i-1][j];
        if (j >= v[i]) f[i][j] = max(f[i][j], f[i][j-v[i]] + w[i]);
    }
}
```

最终去掉一维数组，得到最终的空间优化版本

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

因为 `j-v[i]` 一定是小于 `j` ，所以 `f[j-v[i]]` 一定是在 `f[j]` 之前被计算出来，所以等价于 `f[i][j-v[i]]`

总结

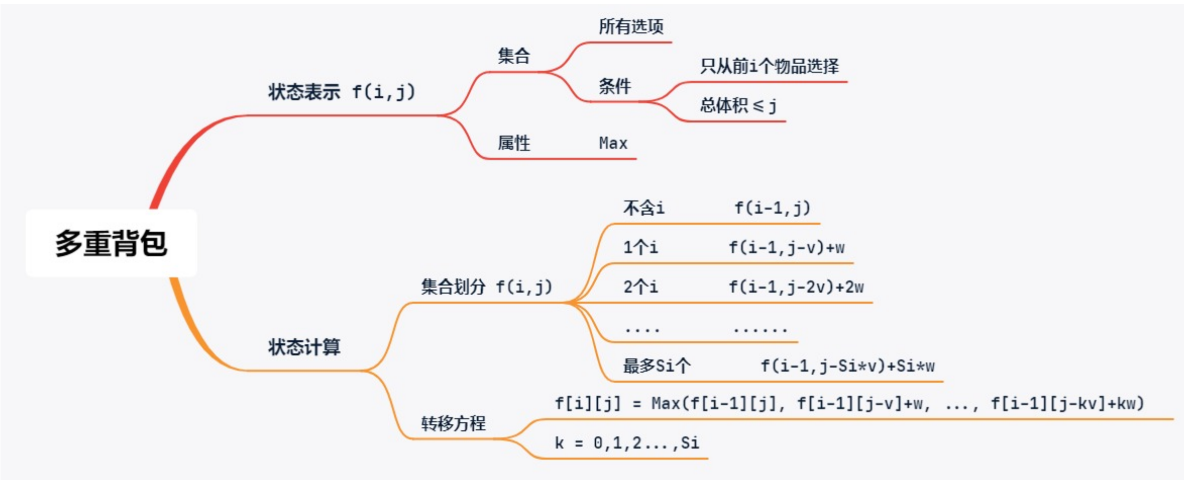


完全背包在进行更新的时候，有可能使用到的 `f[j-v[i]]` 已经是选择了若干次 `i` 物品的状态了，因为转移是发生在整个第 `i` 层上相同层状态的转移，（选择了一个 `i` 的被选择两个 `i` 的状态更新）。

多重背包

每个物品数量不一样，最多有Si个

> 有 N 种物品和一个容量是 V 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。



朴素版本的多重背包问题和完全背包非常相似，只是在物体的数量上做出了限制

代码如下:

```
for (int i = 1; i <= n; i ++) cin >> v[i] >> w[i] >> s[i];
for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= m; j ++)
        for (int k = 0; k <= s[i] && k * v[i] <= j; k ++)
            f[i][j] = max(f[i][j], f[i-1][j-v[i]*k] + w[i]*k);
cout << f[n][m] << endl;
```

但是问题也很明显，时间复杂度是$O(NVS)$当数据量达到1000的时候就会超时

那么，采用和完全背包一样的优化方法可以吗？（不行）

```
f(i,j)   = Max(f[i-1,j], f[i-1,j-v] + w, f[i-1,j-2v]+2w,..., f[i-1,j-sv]+ sw)
f(i,j-v) = Max(         , f[i-1,j-v]    , f[i-1,j-2v]+w,..., f[j-1,j-sv]+(s-1)w,
                 f[i-1,j-(s+1)v]+sw)
```

因为这次项数是有限的，最后会多出来一项 `f[i-1,j-(s+1)v]+sw` 所以优化失败。

这里采用了二进制拆分来进行优化

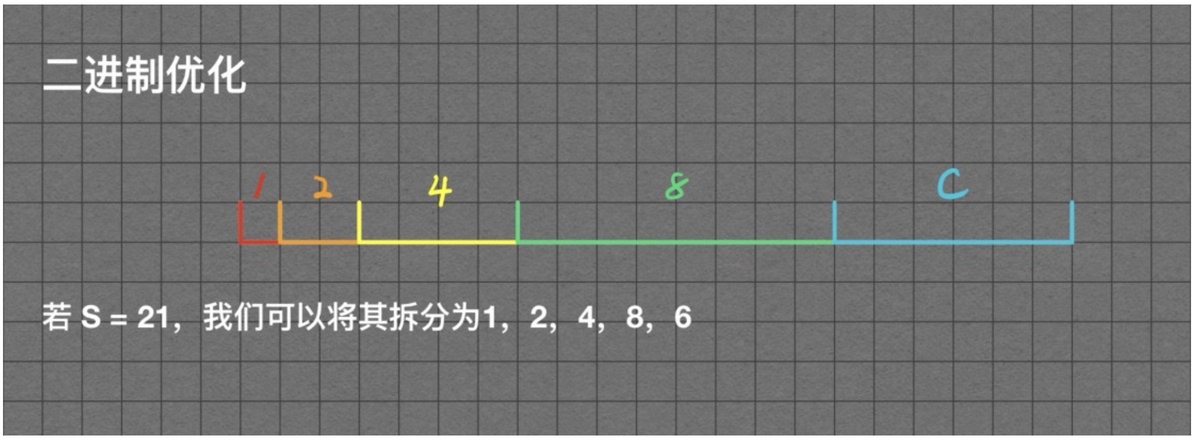
如果一个物体有 `S` 个，正常枚举需要枚举 `S` 次，但是我们可以将其分别打包成为 `log(s)` 组，用这 `log(S)` 组物品可以凑出 `0-S` 中的任何一个数，就成功将其转化为 `01背包` 问题。

打包方法

$$1+2+4+8+...+2^{k-1}+2^k+C=S$$

$$\text{其中，}C<2^{k+1} \quad \sum_{i=0}^{k}2^i\le S$$

例子：



那么从 `0-21` 中的选择就转化为在这5个数中，选或者不选的 `01背包问题` 了。

优化步骤：

- 将每个物品的数量 `Si` 进行二进制拆分为 `logS` 个物体
- 计算分组后的新体积和价值
- 用 `01背包` 进行计算

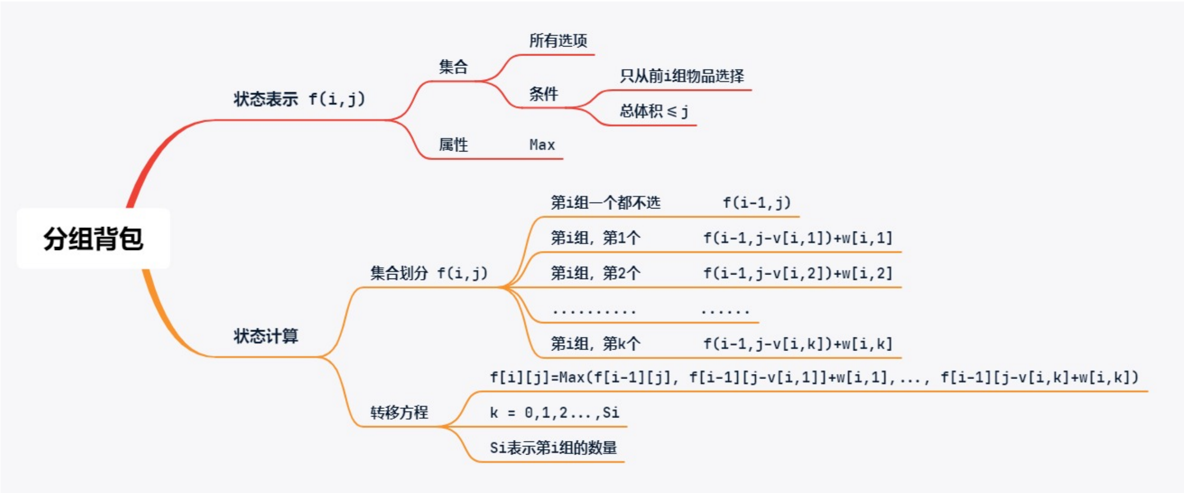
代码

```
#include <bits/stdc++.h>
using namespace std;

const int N = 2020;
int n, m;
int f[N];

struct good {
    int v, w;
};

vector<good> goods;

int main() {
    cin >> n >> m;
    for (int i = 1; i <= n; i ++) {
        int v, w, s;
```

```cpp
        cin >> v >> w >> s;
        for (int k = 1; k <= s; k <<= 1) {
            s -= k;
            goods.push_back({k*v, k*w});
        }
        if (s > 0) goods.push_back({s*v, s*w});
    }

    for (auto good: goods)
        for (int j = m; j >= good.v; j --)
            f[j] = max(f[j], f[j-good.v] + good.w);
    cout << f[m];
    return 0;
}
```

分组背包

物品分为若干种，每一组里面最多只能选择一个物品，是互斥的。



多重背包是枚举第i件物品选几个，分组背包是枚举第i组物品选哪个

就是在每组里面跑 `01背包`

```
for (int i = 1; i <= n; i ++ ){
    cin >> s[i];
    for (int j = 0; j < s[i]; j ++ )
        cin >> v[i][j] >> w[i][j];
}

for (int i = 1; i <= n; i ++ )
    for (int j = m; j >= 0; j -- ) // n组物品跑01背包
        for (int k = 0; k < s[i]; k ++ ) // 枚举组内物品，选一个最大的
            if (j >= v[i][k])
                f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);
printf("%d", f[m]);
```

还有更加清奇的写法（对上面的写法进行**等价变形**）

```
while(n --)  // n组物品
{
    int s;
    scanf("%d", &s);
    for(int i = 1; i <= s; i ++)  scanf("%d%d", &v[i], &w[i]);
    for(int j = m; j >= 0; j --) // 跑01背包
        for(int k = 1; k <= s; k ++) // 枚举 k
            if(j >= v[k])  f[j] = max(f[j], f[j - v[k]] + w[k]);
}
printf("%d", f[m]);
```