背包问题

信息竞赛组

广州大学附属中学

2024 年 7 月

目录

- 有 N 件物品和一个容量为 V 的背包。

- 有 N 件物品和一个容量为 V 的背包。
- 放入第 i 件物品的重量是 C_i（即占用背包的空间容量），得到的价值是 W_i。

- 有 N 件物品和一个容量为 V 的背包。
- 放入第 i 件物品的重量是 C_i（即占用背包的空间容量），得到的价值是 W_i。
- 求解将哪些物品装入背包可使价值总和最大。

- 有 N 件物品和一个容量为 V 的背包。
- 放入第 i 件物品的重量是 C_i（即占用背包的空间容量），得到的价值是 W_i。
- 求解将哪些物品装入背包可使价值总和最大。
- 特点：每种物品仅有一件，可以选择放或不放。

假设有 4 件物品（$C = [2, 3, 5, 5], W = [2, 4, 3, 7]$）和一个背包容量为 10 的背包。

假设有 4 件物品（$C = [2, 3, 5, 5], W = [2, 4, 3, 7]$）和一个背包容量为 10 的背包。

定义 $f(i, j)$ 表示前 i 件物品，放入容量为 j 的背包可以获得的最大价值。

假设有 4 件物品（$C = [2, 3, 5, 5], W = [2, 4, 3, 7]$）和一个背包容量为 10 的背包。

定义 $f(i, j)$ 表示前 i 件物品，放入容量为 j 的背包可以获得的最大价值。

状态转移方程：$f(i, j) = \max(f(i-1, j), f(i-1, j-C_i) + W_i)$。

假设有 4 件物品（$C = [2, 3, 5, 5], W = [2, 4, 3, 7]$）和一个背包容量为 10 的背包。

定义 $f(i, j)$ 表示前 i 件物品，放入容量为 j 的背包可以获得的最大价值。

状态转移方程：$f(i, j) = \max(f(i-1, j), f(i-1, j-C_i) + W_i)$。

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
2	2	1											
3	4	2											
5	3	3											
5	7	4											

假设有 4 件物品（$C = [2, 3, 5, 5], W = [2, 4, 3, 7]$）和一个背包容量为 10 的背包。

定义 $f(i, j)$ 表示前 i 件物品，放入容量为 j 的背包可以获得的最大价值。

状态转移方程：$f(i, j) = \max(f(i-1, j), f(i-1, j-C_i) + W_i)$。

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

```cpp
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        if (j < C[i]) {
            f[i][j] = f[i - 1][j]; // 只能不选
        } else {
            f[i][j] = max(f[i - 1][j], f[i - 1][j - C[i]] + W[i]); // 不选和选
        }
    }
}
```

时间复杂度为 $O(NV)$。

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

第 i 行的值与哪些值有关?

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

第 i 行的值与哪些值有关?

根据状态转移方程，发现第 i 行的值只与第 $i-1$ 行的值有关，即转移过程中只用到两行的值。

- **滚动数组**是一种能够在动态规划中降低空间复杂度的方法。

- **滚动数组**是一种能够在动态规划中降低空间复杂度的方法。
- 通过观察状态转移方程来判断需要使用哪些数据，可以抛弃哪些数据，一旦找到关系，就可以用新的数据不断覆盖旧的数据来减少空间的使用。

在 01 背包中，状态转移过程中只需要用到两行的值，所以定义一个只有 2 行的二维数组 $f[2][V]$ 即可。

在 01 背包中，状态转移过程中只需要用到两行的值，所以定义一个只有 2 行的二维数组 $f[2][V]$ 即可。

C	W	$f[t]$	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	$f[0]$	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	$f[1]$	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	$f[0]$	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	$f[1]$	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	$f[0]$	4	0	0	2	4	4	7	7	9	11	11	**13**

在 01 背包中，状态转移过程中只需要用到两行的值，所以定义一个只有 2 行的二维数组 $f[2][V]$ 即可。

C	W	$f[t]$	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	$f[0]$	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	$f[1]$	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	$f[0]$	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	$f[1]$	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	$f[0]$	4	0	0	2	4	4	7	7	9	11	11	**13**

```
1  int t = 0;
2  for (int i = 1; i <= N; i++) {
3      for (int j = 0; j <= V; j++) {
4          if (j < C[i]) f[t ^ 1][j] = f[t][j]; // 只能不选
5          else f[t ^ 1][j] = max(f[t][j], f[t][j - C[i]] + W[i]); // 不选和选
6      }
7      t ^= 1; // t在01之间切换
8  }
```

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。
- 观察下面的表格，空间能否优化成一维的？

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。
- 观察下面的表格，空间能否优化成一维的？

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。
- 观察下面的表格，空间能否优化成一维的？

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。
- 观察下面的表格，空间能否优化成一维的？

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

- 发现如果倒着枚举背包容量，当枚举到 j 时，$f[j]$ 和 $f[j - C_i]$ 都是上一行的数据。

- 定义 $f[j]$ 表示背包容量为 j 时可以获得的最大价值。
- 观察下面的表格，空间能否优化成一维的？

C	W	i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	2	1	0	0	**2**	2	2	2	2	2	2	2	2
3	4	2	0	0	2	4	4	**6**	6	6	6	6	6
5	3	3	0	0	2	4	4	**6**	6	6	7	7	9
5	7	4	0	0	2	4	4	7	7	9	11	11	**13**

- 发现如果倒着枚举背包容量，当枚举到 j 时，$f[j]$ 和 $f[j - C_i]$ 都是上一行的数据。

```
1  for (int i = 1; i <= N; i++) {
2      for (int j = V; j >= C[i]; j--) { // 必须倒着枚举背包容量
3          f[j] = max(f[j], f[j - C[i]] + W[i]); // 不选和选
4      }
5  }
```

目录

- 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。

- 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。
- 放入第 i 种物品的重量是 C_i，价值是 W_i。

- 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。
- 放入第 i 种物品的重量是 C_i，价值是 W_i。
- 求解：将哪些物品装入背包，可使这些物品占用的空间总和不超过背包容量，且价值总和最大。

- 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。
- 放入第 i 种物品的重量是 C_i，价值是 W_i。
- 求解：将哪些物品装入背包，可使这些物品占用的空间总和不超过背包容量，且价值总和最大。
- 特点：每种物品有无限件。

- 完全背包模型与 01 背包类似，与 01 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。

- 完全背包模型与 01 背包类似，与 01 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。
- 我们可以借鉴 01 背包的思路，进行状态定义：设 $f[i][j]$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

- 完全背包模型与 01 背包类似，与 01 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。
- 我们可以借鉴 01 背包的思路，进行状态定义：设 $f[i][j]$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。
- 可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 $O(n^3)$ 的。

- 完全背包模型与 01 背包类似，与 01 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。
- 我们可以借鉴 01 背包的思路，进行状态定义：设 $f[i][j]$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。
- 可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 $O(n^3)$ 的。
- 状态转移方程：

$$f(i,j)=\max_{k=0}^{+\infty}\{f(i-1,j-k\times C_i)+W_i\}(j-k\times C_i\geq 0)$$

- 考虑做一个简单的优化。可以发现，对于 $f(i,j)$，只要通过 $f(i,j-C_i)$ 转移就可以了。

- 考虑做一个简单的优化。可以发现，对于 $f(i,j)$，只要通过 $f(i,j-C_i)$ 转移就可以了。
- 状态转移方程：$f(i,j)=\max(f(i-1,j),f(i,j-C_i)+W_i)$

- 考虑做一个简单的优化。可以发现，对于 $f(i,j)$，只要通过 $f(i,j-C_i)$ 转移就可以了。
- 状态转移方程：$f(i,j)=\max(f(i-1,j),f(i,j-C_i)+W_i)$
- 理由是当我们这样转移时，$f(i,j-C_i)$ 已经由 $f(i,j-2\times C_i)$ 更新过，那么 $f(i,j-w_i)$ 就是充分考虑了第 i 件物品所选次数后得到的最优结果。

```
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        if (j < C[i]) {
            f[i][j] = f[i - 1][j]; // 只能不选
        } else {
            f[i][j] = max(f[i][j], f[i][j - C[i]] + W[i]); // 不选和再选一次
        }
    }
}
```

```
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        if (j < C[i]) {
            f[i][j] = f[i - 1][j]; // 只能不选
        } else {
            f[i][j] = max(f[i][j], f[i][j - C[i]] + W[i]); // 不选和再选一次
        }
    }
}
```

是否可以优化空间?

```
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        if (j < C[i]) {
            f[i][j] = f[i - 1][j]; // 只能不选
        } else {
            f[i][j] = max(f[i][j], f[i][j - C[i]] + W[i]); // 不选和再选一次
        }
    }
}
```

是否可以优化空间?

空间优化: f[j] 表示背包容量为 j 时可以获得的最大价值。

```
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        if (j < C[i]) {
            f[i][j] = f[i - 1][j]; // 只能不选
        } else {
            f[i][j] = max(f[i][j], f[i][j - C[i]] + W[i]); // 不选和再选一次
        }
    }
}
```

是否可以优化空间?

空间优化: f[j] 表示背包容量为 j 时可以获得的最大价值。

```
for (int i = 1; i <= N; i++) {
    for (int j = C[i]; j <= V; j++) { // 必须正着枚举背包容量
        f[j] = max(f[j], f[j - C[i]] + W[i]); // 不选和再选一次
    }
}
```

目录

- 有 N 种物品和一个容量为 V 的背包。

- 有 N 种物品和一个容量为 V 的背包。
- 第 i 种物品最多有 K_i 件可用，每件物品的重量是 C_i，价值是 W_i。

- 有 N 种物品和一个容量为 V 的背包。
- 第 i 种物品最多有 K_i 件可用，每件物品的重量是 C_i，价值是 W_i。
- 求解将哪些物品装入背包可使这些物品的占用的空间总和不超过背包容量，且价值总和最大。

- 有 N 种物品和一个容量为 V 的背包。
- 第 i 种物品最多有 K_i 件可用，每件物品的重量是 C_i，价值是 W_i。
- 求解将哪些物品装入背包可使这些物品的占用的空间总和不超过背包容量，且价值总和最大。
- 特点：每种物品数量有限。

- 一个很朴素的想法就是：把「每种物品选 K_i 次」等价转换为「有 K_i 个相同的物品，每个物品选一次」。

- 一个很朴素的想法就是：把「每种物品选 K_i 次」等价转换为「有 K_i 个相同的物品，每个物品选一次」。
- 这样就转换成了一个 01 背包模型。

- 一个很朴素的想法就是：把「每种物品选 K_i 次」等价转换为「有 K_i 个相同的物品，每个物品选一次」。
- 这样就转换成了一个 01 背包模型。

```
for (int i = 1; i <= N; i++) {
    for (int j = V; j >= C[i]; j--) {
        for (int k = 0; k <= K[i]; k++) { // 选k个
            if (j - k * C[i] >= 0) {
                f[j] = max(f[j], f[j - k * C[i]] + k * W[i]);
            }
        }
    }
}
```

- 一个很朴素的想法就是：把「每种物品选 K_i 次」等价转换为「有 K_i 个相同的物品，每个物品选一次」。
- 这样就转换成了一个 01 背包模型。

```
for (int i = 1; i <= N; i++) {
    for (int j = V; j >= C[i]; j--) {
        for (int k = 0; k <= K[i]; k++) { // 选k个
            if (j - k * C[i] >= 0) {
                f[j] = max(f[j], f[j - k * C[i]] + k * W[i]);
            }
        }
    }
}
```

- 时间复杂度为 $O(V \sum_{i=1}^{n} K_i)$。

- 如果 $K_i \times C_i \geq V$，那么第 i 个物品相当于有无限个，可以直接使用完全背包的转移方式。

- 如果 $K_i \times C_i \geq V$，那么第 i 个物品相当于有无限个，可以直接使用完全背包的转移方式。

```
for (int i = 1; i <= N; i++) {
    if (K[i] * C[i] >= V) { // 转完全背包
        for (int j = C[i]; j <= V; j++) {
            f[j] = max(f[j], f[j - C[i]] + W[i]);
        }
    }
}
```

- 如果 $K_i \times C_i \geq V$，那么第 i 个物品相当于有无限个，可以直接使用完全背包的转移方式。

```
for (int i = 1; i <= N; i++) {
    if (K[i] * C[i] >= V) { // 转完全背包
        for (int j = C[i]; j <= V; j++) {
            f[j] = max(f[j], f[j - C[i]] + W[i]);
        }
    }
}
```

- 如果满足 $K_i \times C_i \geq V$ 的物品个数比较多，时间复杂度就接近完全背包的时间复杂度。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。
- 显然，复杂度中的 $O(NV)$ 部分无法再优化，我们只能从 $O(\sum K_i)$ 处入手。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。
- 显然，复杂度中的 $O(NV)$ 部分无法再优化，我们只能从 $O(\sum K_i)$ 处入手。
- 我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。
- 显然，复杂度中的 $O(NV)$ 部分无法再优化，我们只能从 $O(\sum K_i)$ 处入手。
- 我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。
- 在朴素的做法中，$\forall j \leq K_i, A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。
- 显然，复杂度中的 $O(NV)$ 部分无法再优化，我们只能从 $O(\sum K_i)$ 处入手。
- 我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。
- 在朴素的做法中，$\forall j \leq K_i, A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。
- 举例来说，我们考虑了「同时选 $A_{i,1}, A_{i,2}$」与「同时选 $A_{i,2}, A_{i,3}$」这两个完全等效的情况。

- 考虑优化。我们仍考虑把多重背包转化成 01 背包模型来求解。
- 显然，复杂度中的 $O(NV)$ 部分无法再优化，我们只能从 $O(\sum K_i)$ 处入手。
- 我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。
- 在朴素的做法中，$\forall j \leq K_i, A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。
- 举例来说，我们考虑了「同时选 $A_{i,1}, A_{i,2}$」与「同时选 $A_{i,2}, A_{i,3}$」这两个完全等效的情况。
- 这样的重复性工作我们进行了许多次。那么**优化拆分方式**就成为了解决问题的突破口。

我们可以通过「二进制分组」的方式使拆分方式更加优美。

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$

我们可以通过「二进制分组」的方式使拆分方式更加优美。
具体来说：

1. 令 $A_{i,j}\,(j \in [0, \lfloor \log_2(K_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。
2. 特殊地，若 K_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $K_i - 2^{\lfloor \log_2(K_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$
- $31 = 1 + 2 + 4 + 8 + 16$

- 显然，通过上述拆分方式，可以表示任意 $\leq K_i$ 个物品的等效选择方式。

- 显然，通过上述拆分方式，可以表示任意 $\leq K_i$ 个物品的等效选择方式。
- 将每种物品按照上述方式拆分后，使用 01 背包的方法解决即可。

- 显然，通过上述拆分方式，可以表示任意 $\leq K_i$ 个物品的等效选择方式。
- 将每种物品按照上述方式拆分后，使用 01 背包的方法解决即可。
- 时间复杂度 $O(V \sum_{i=1}^{n} \log_2 K_i)$。

二进制分组优化转 01 背包：

二进制分组优化转 01 背包：

```
int tot = 0;
int c[M], w[M]; // 拆分后的物品，假设M足够大
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= K[i]; j *= 2) { // 二进制拆分的部分
        c[++tot] = j * C[i];
        w[tot] = j * W[i];
        K[i] -= j;
    }
    if (K[i]) { // 剩余的部分
        c[++tot] = j * C[i];
        w[tot] = j * W[i];
    }
}
```