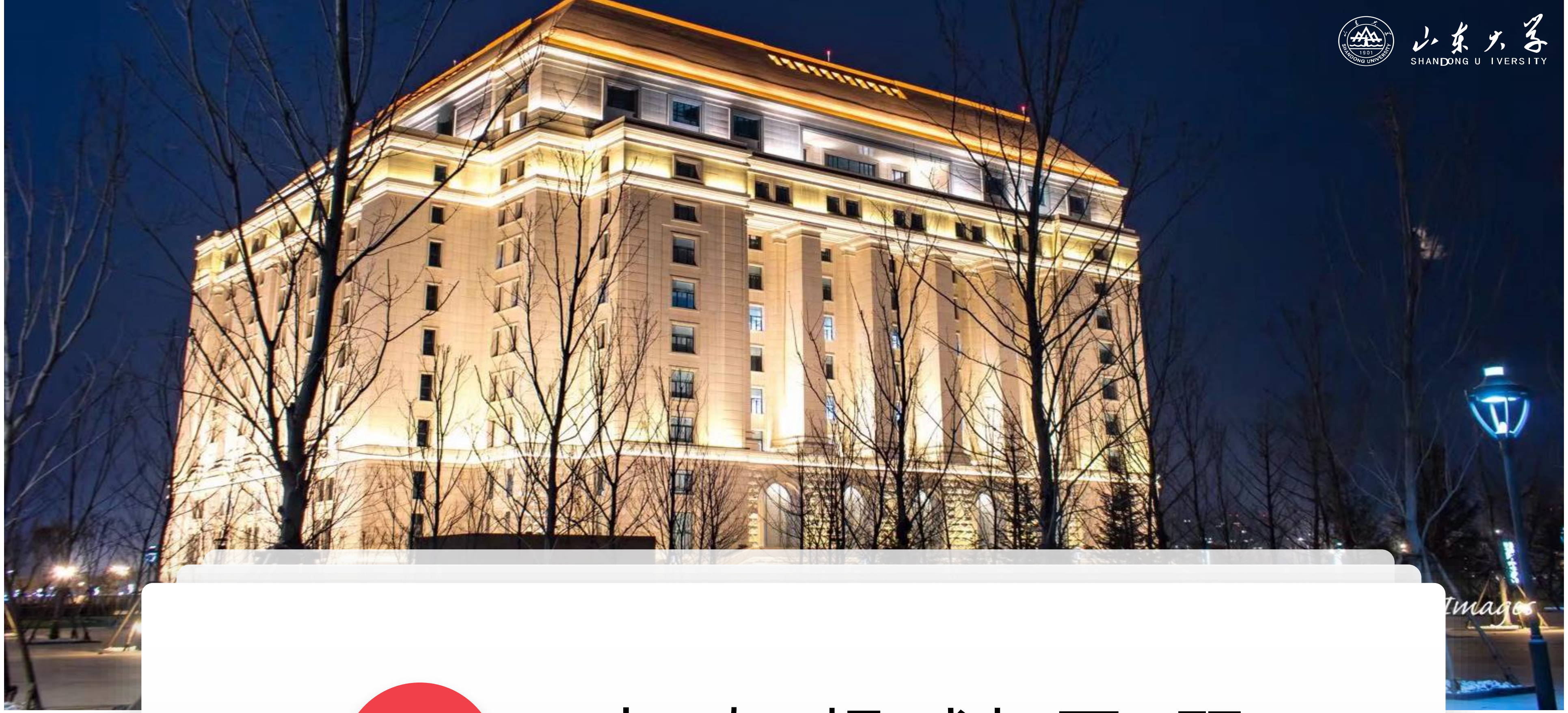
程序设计思维与实践

Thinking and Practice in Programming

动态规划（二） | 内容负责：师浩晏



0 动态规划回顾

Dynamic Programming review

动态规划回顾

- 动态规划解题步骤：

  1. 状态定义
  2. 状态转移方程
  3. 状态初始化
  4. 输出答案

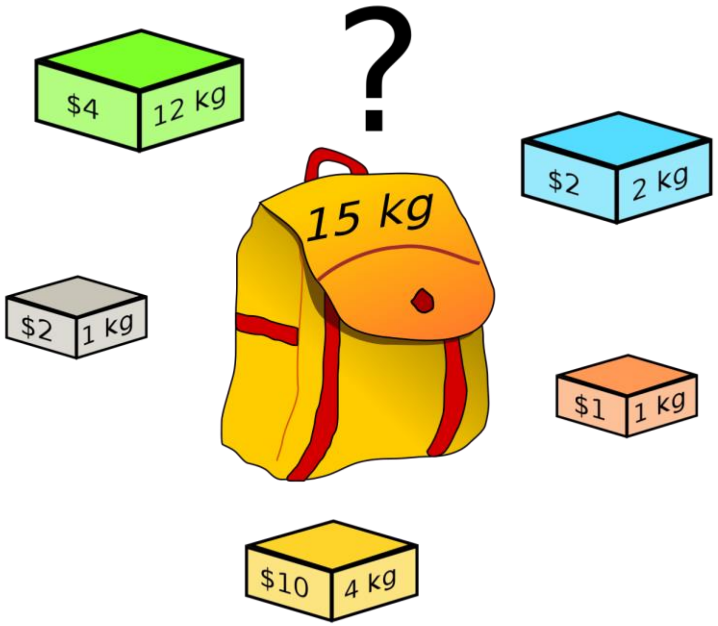
- 走迷宫问题：
  - 定义f[i][j]代表从(1,1)走到(i,j)的方案数
  - f[i][j] = f[i-1][j] + f[i][j-1]
  - f[1][1] = 1

动态规划回顾

- 动态规划常见模型
  - 线性型
  - 坐标型
  - **背包型**
  - 区间型
  - 状态压缩型
  - 树型
  - 矩阵型

动态规划回顾

- 背包问题：一类关于背包的问题
  - 0–1背包
  - 完全背包
  - 多重背包
  - 分组背包
  - ……



- 可以用动态规划完美解决！

1

0-1背包问题

0-1 knapsack problem

0-1背包

- 01 背包-问题描述：
  - 有 N 件物品和一个容量为 V 的背包。 第 i 件物品体积是 W_i，价值是V_i。求解将哪些物品装入背包可使这些物品的体积总和不超过背包容量，且总价值最大 。
  - 特点：每种物品仅有一件，可以选择放或不放 (对应 1 或 0)。
  - 样例

3件物品，背包容量为10

编号	体积w	价值v
1	5	200
2	4	100
3	7	300

- Ans = 300 （前两个或只选第三个）

0-1背包

- 怎样选择？贪心?

1. 价值大的?
   - 反例

输入样例1：
4 10
4 3 5 7
15 7 20 25
输出样例1：
35

2. 单位价值大的?
   - 反例

输入样例2：
4 20
2 9 10 15
2 9 10 16
输出样例2：
19

0-1背包

- 01背包：
  - 用子问题来定义状态
  - 设计状态：$f_{i,j}$ 表示仅考虑前 i 件物品，放入一个容量为 j 的背包可以获得的最大价值。
  - 状态转移方程

$$f_{i,j} = \max(f_{i-1,j},\ f_{i-1,j-w_i} + v_i)$$

0-1背包

- 01背包：
  - 用子问题来定义状态

考虑前 i 个物品，背包大小为 j

第 i 个物品不选 → 考虑前 i-1 个物品，背包大小为j

第 i 个物品选 → 考虑前 i-1 个物品，背包大小为 j-w[i]

价值即为 f[i-1][j]

价值即为 f[i-1][j-w[i]] + v[i]

$$f_{i,j} = \max(f_{i-1,j}, \ f_{i-1,j-w_i} + v_i)$$

0-1背包

$$f_{i,j} = \max(f_{i-1,j},\ f_{i-1,j-w_i} + v_i)$$

- 这个方程非常重要！！
  基本上所有与背包相关的问题的方程都是由它衍生出来的。

- 详细解释：
  - “前 i 件物品放入容量为 j 的背包中”这个子问题中，我们现在考虑第 i 件物品的策略（放、不放）
  - 如果选择不放，那么问题转化为“前 i-1件物品放入容量为 j 的背包中”，也就是 f[i-1][j] 所表示的子问题的状态；
  - 如果选择放，那么问题转化成“前 i-1 件物品放入容量为 j-w[i] 的背包中”，此时能获得的最大价值就是 f[i-1][j-w[i]] + v[i]
  - 两种策略，决策为取价值更大的策略

0-1背包

- 答案是什么?
  - f[N][V]

0-1背包

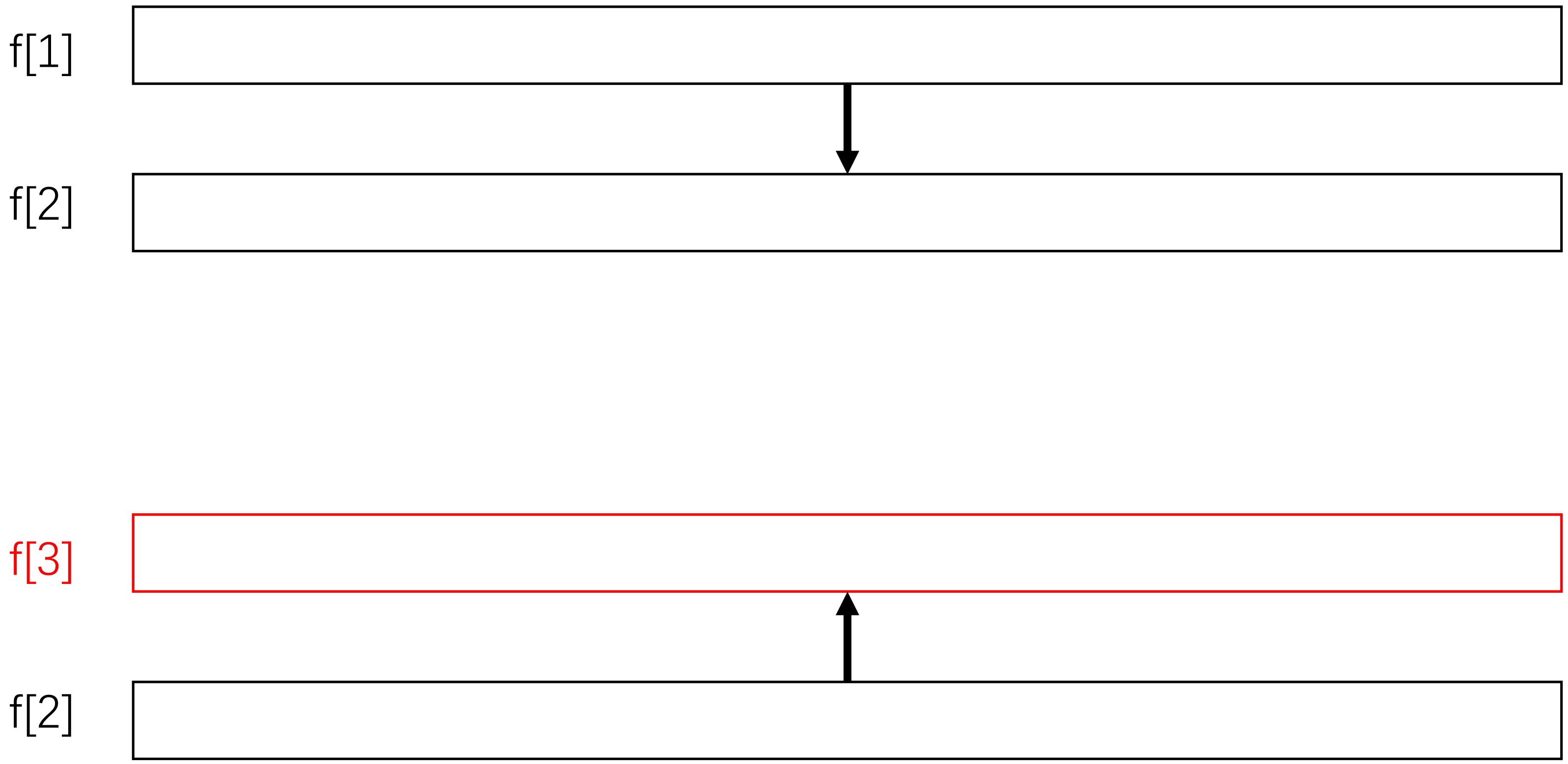
- 复杂度
  - 时间复杂度 —— O(N×V)
  - 空间复杂度 —— O(N×V)
  - 其中时间复杂度基本已经不能再优化了
  - 但是可以优化空间复杂度，将其降到O(V)
- 滚动数组
  - 首先回到刚刚的状态 f[i][j]，其表示 “前 i 件物品，背包容量为 j，背包内物品体积至多为 j” 的最大价值
  - 此时答案是 f[N][V]
  - 观察一下

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= V; j++) {
        f[i][j] = f[i - 1][j];
        if (j >= w[i]) {
            f[i][j] = max(f[i - 1][j], f[i - 1][j - w[i]] + c[i]);
        }
    }
}
```

  - 计算f[i]这一行的时候只用到了f[i-1]这一行的信息，没有用到更靠前的行

0-1背包

- 只开两行数组，交替计算即可



0-1背包

- 只开两行数组，交替计算即可
- 奇数行用第 1 行，偶数行用第 0 行
- 答案是 f[N & 1][V]

```
long long f[2][5010];
```

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= V; j++) {
        f[i & 1][j] = f[(i - 1) & 1][j];
        if (j >= w[i]) {
            f[i & 1][j] = max(f[(i - 1) & 1][j], f[(i - 1) & 1][j - w[i]] + c[i]);
        }
    }
}
```

0-1背包

- 滚动数组
  - 对于这个题，还可以优化成只需要一个一维数组
  - 第 2 层循环改为逆序，不影响答案

```
for (int i = 1; i <= N; ++i) {
    for (int j = 0; j <= V; ++j) {
        f[i][j] = f[i - 1][j];
        if (j - w[i] >= 0)
            f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);
    }
}
```

```
for (int i = 1; i <= N; ++i) {
    for (int j = V; j >= 0; --j) {
        f[i][j] = f[i - 1][j];
        if (j - w[i] >= 0)
            f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);
    }
}
```

  - 可以原地更新
  - 枚举到一个位置时，只用到了上一行它左边位置的值

0-1背包

编号	体积w	价值v
1	4	15
2	3	7
3	5	20
4	7	25

- 滚动数组
  - 更直观地表示一下计算过程：比如一共由 4 件物品，背包容量为 10
  - f 数组中的数据为

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	15	15	15	15	15	15	15
2	0	0	0	7	15	15	15	22	22	22	22
3	0	0	0	7	15	20	20	22	27	35	35
4	0	0	0	7	15	20	20	25	27	35	35

容量

物品

$$f_{i,j} = \max(f_{i-1,j}, \ f_{i-1,j-w_i} + v_i)$$

0-1背包

编号	体积w	价值v
1	4	15
2	3	7
3	5	20
4	7	25

- 滚动数组
  - 更直观地表示一下计算过程：比如一共由 4 件物品，背包容量为 10
  - f 数组中的数据为

j	0	1	2	3	4	5	6	7	8	9	10
…											
i=1	0	0	0	15	15	15	15	15	15	15	
i=2	0	0	7	15	20	20	25	27	35	35	
…											

j	0	1	2	3	4	5	6	7	8	9	10
…											
i=1	0	0	7	15	20	20	25	27	35	35	
i=2											
…											

0-1背包

- 滚动数组
  - 如果只用一个一维数组 f[V]，能不能保证第 i 次循环结束后 f[j] 中表示的就是我们定义的状态 f[i][j]?
  - f[i][j] 是由 f[i-1][j] 和 f[i-1][j-w[i]] 两个子问题递推过来
  - 只需要保证枚举顺序从 j = V → 0 即可
  - 滚动数组的关键点是：逆序!

    逆序保证每个位置求解时，所用到的数据都是未被覆盖的
  - 代码：

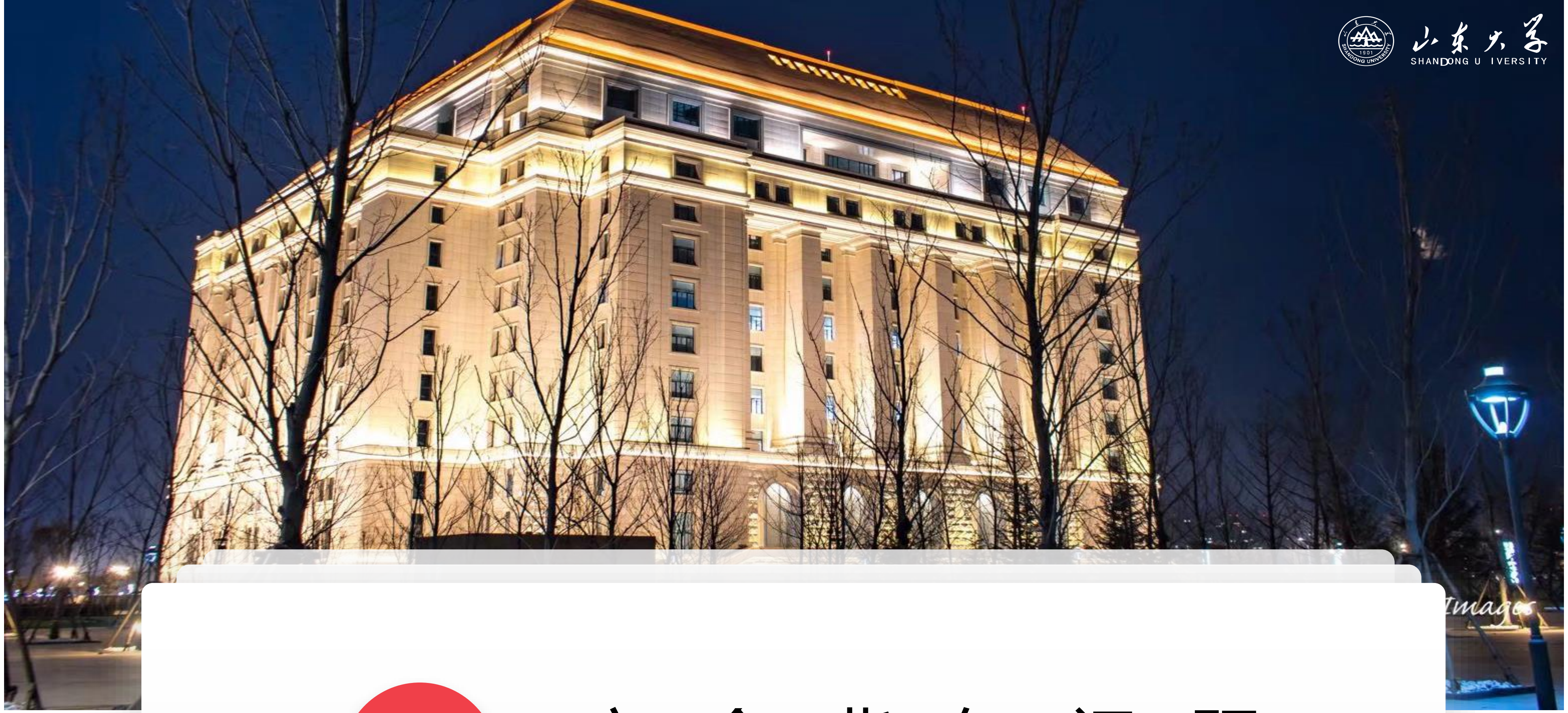
```
for (int i = 1; i <= N; i++) {
    for (int j = V; j >= 1; j--) {
        if(j - w[i] >= 0) f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
ans = f[V];
```

0-1背包

- 滚动数组
  - 这种对空间的优化通常被称作“滚动数组”
  - 在递推法中，如果计算顺序很特殊，而且计算新状态所用到的原状态不多，可以尝试用滚动数组减少内存开销
  - 一定的弊端：
    - 比如打印方案变得困难
    - 在 dp 过程结束后，只有最后一个阶段的状态值，而没有前面的值

0-1背包

- 小结
  - 0-1 背包问题是最基本的背包问题，它包含了背包问题中设计状态、状态转移方程的最基本思想
  - 并且，别的类型的背包问题往往也可以转换成 0-1 背包问题求解
  - 故一定要仔细体会上面基本思路的得出方法、状态转移方程的意义，以及滚动数组的思想



2 完全背包问题

Complete knapsack problem

完全背包

- 完全背包：
  - 有 N 种物品和一个容量为 V 的背包。 第 i 件物品体积是 W_i，价值是V_i。求解将哪些物品装入背包可使这些物品的体积总和不超过背包容量，且总价值最大 。
  - 特点：每种物品有无数件，可以选择 0 或多件
  - 样例

    3件物品，背包容量为10

编号	体积w	价值v
1	5	200
2	4	100
3	7	300

- Ans = 400 （第一个物品选两个）

完全背包

- 完全背包
  - 无限个物品?
  - 显然不是
  - 因为有容量 V 的限制，每种物品最多 V÷w[i] 个

完全背包

- 完全背包
- 背包容积 V = 10

编号	体积w	价值v	数量
1	5	200	2
2	4	100	2
3	7	300	1

完全背包

→

编号	体积w	价值v
1	5	200
2	5	200
3	4	100
4	4	100
5	7	300

0/1背包

完全背包

- 完全背包
  - 无限个物品?
  - 显然不是
  - 因为有容量 V 的限制，每种物品最多 V÷w[i] 个
  - 因此对于每种物品而言，与它相关的策略已经并非取、不取两种策略
  - 而是取 0 件、取 1 件、……、取 V÷w[i]件

完全背包

- 完全背包
  - 设计状态：依旧按照 0-1 背包时的思路，f[i][j] 表示前 i 种物品，放入一个容量为 j 的背包的最大价值
  - 状态转移方程：

$$f_{i,j} = \max\{f_{i-1,j},\ f_{i-1,j-k\times w_i} + k \times v_i \mid k = 0, \cdots, \lfloor \frac{V}{w_i} \rfloor\}$$

  - 时间复杂度?
    - O(N×V)?
    - 虽然也是有 N×V 个状态需要求解，但是求解单个状态 f[i][j] 的时间复杂度已不再是常数，而是变成了 O(V/w[i])
    - 总的复杂度是 —— $O(V \times \sum_{i=1}^{n} \frac{V}{w_i})$

完全背包

- 前面
  - 将 0-1 背包的基本思路进行改进，得到这样一个清晰的方法
  - 说明 0-1 背包问题的方程的确很重要，可以推其他类型的背包问题

  - 但是，当前时间复杂度并不理想，在 w[i] 较小的时候运算量会很大，可以对其进行优化。空间复杂度是 O(N×V)，同样可以进行改进

完全背包

- 一个非常容易想到的优化：
  - 假如现在有两件物品 i,j，满足：$W_i <= W_j$ 且 $V_i >= V_j$
  - “物 美 价 廉”：那就可以将物品 j 去掉

- 如果数据是随机生成的，那么这样优化会很好地减少运算量
- 但是这种优化并不能改善最坏情况下的时间复杂度 O，即面对极端数据的时候并不能更有效地降低计算量

- —— 还需要其他的技巧来进行优化

完全背包

- 转变思路

$$f_{i,j} = max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

考虑前 i 个物品，背包大小为 j

第 i 个物品不选

第 i 个物品选1个

考虑前 i-1 个物品，背包大小为j

依然考虑前 i 个物品，背包大小为 j-w[i]

价值即为 f[i-1][j]

价值即为 f[i][j-w[i]] + v[i]

完全背包

- 转变思路

$$f_{i,j} = max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= V; j++) {
        f[i][j] = f[i - 1][j];
        if (j >= w[i]) {
            f[i][j] = max(f[i - 1][j], f[i][j - w[i]] + c[i]);
        }
    }
}
```

- 思考：怎样滚动数组优化

- 和 0-1背包 写法相同，只需要把第二层倒序循环改成正序循环即可

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= V; j++) {
        if(j - w[i] >= 0) f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
ans = f[V];
```

完全背包

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= V; j++) {
        if(j - w[i] >= 0) f[j] = max(f[j], f[j - w[i]] + v[i]);
    }
}
ans = f[V];
```

- 为什么可以这样写？
  - 0-1背包时，因为要保证第 i 次循环中的状态 f[i][j] 是从状态 f[i-1][j-w[i]] 递推而来，为了保证“物品只选一次”而逆序。即“选入第 i 件物品”的策略时，依据的是一个没有选入第 i 件物品的子结果 f[i-1][j-w[i]]
  - 现在完全背包考虑“加选一个物品”时，正需要一个可能已选入第 i 种物品的子结果 f[i][j-w[i]]，所以必需采用正序循环。
- 时间复杂度优化为 O(N×V)
- 空间复杂度优化为 O(V)

3

多 重 背 包 问 题

Multiple knapsack problem

多重背包

- 多重背包：
  - 有 N 件物品和一个容量为 V 的背包。 第 i 件物品体积是 Wi，价值是Vi，有 Ci 件可用，求解将哪些物品装入背包可使这些物品的体积总和不超过背包容量，且总价值最大
  - 特点：每种物品有 有限件
  - 样例

    3件物品，背包容量为10

编号	体积w	价值v	数量
1	2	200	1
2	4	160	2
3	7	300	1

- Ans = 520 （第一个物品选 1 个，第二个物品选 2 个）

多重背包

- 多重背包
  - 思路和完全背包类似
  - 区别在于完全背包的物品有 $V \div W_i$ 个，此处有 C_i 个。
  - 设计状态：f[i][j] 表示前 i 种物品恰放入一个容量为 V 的背包的最大权值
  - 方程几乎完全相同：

$$f_{i,j} = \max\{f_{i-1,j},\ f_{i-1,j-k\times w_i} + k \times v_i \mid k = 0, \cdots, \boxed{C_i}\}$$

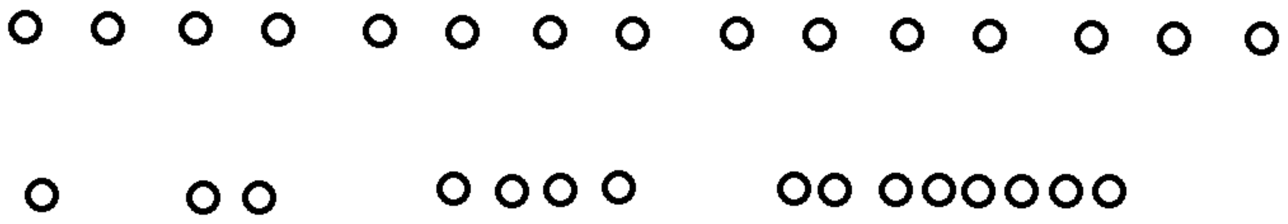
  - 复杂度是 $O(V \times \sum_{i=1}^{n} C_i)$

多重背包

- 多重背包
  - 能不能优化?
  - 可以看到，在极端情况下，多重背包的复杂度是与未优化的完全背包一样的
  - 但又有个数的限制，所以不能采用完全背包的优化方式将其优化到 $O(N \times V)$
  - 解决方法是 —— 二进制拆分
- 二进制拆分
  - 采用进制的思想将 C_i 进行二进制拆分，然后转换成 0-1背包问题

多重背包

- 二进制拆分



- 原本有某物品有15个，相当于有15个物品
- 把 15个 分组为 1个，2个，4个，8个，这样变成了4个物品
- 拿 3 个 → 拿 1 和 2
- 拿 12 个 → 拿 4 和 8
- 0 到 15 任意一种决策，都可以拆分成若干个物品的组合
- 分组后和原来等价

多重背包

- 二进制拆分
  - 更特殊的情况，13 怎么拆?
  - 拆成1，2，4，6，其中 6 = 13 – (1+2+4)
  - 小于 6：用1，2，4凑
  - 大于 6：先凑一个6，再用1，2，4凑剩下的部分

多重背包

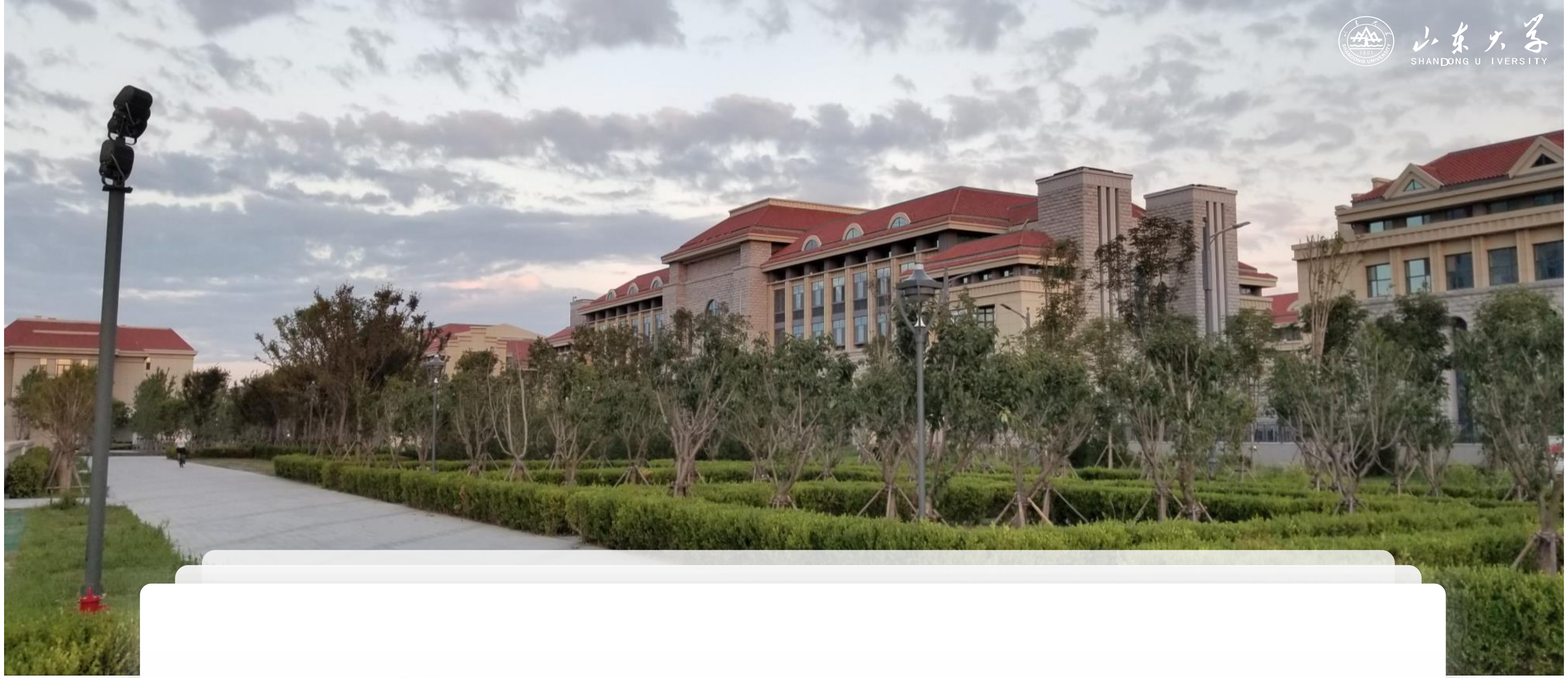
- 二进制拆分
  - 代码

```
while (N--) {
    int weight, value, count;
    cin >> weight >> value >> count;
    for (int i = 1; i <= count; i *= 2) {
        addItem(weight * i, value * i);
        count -= i;
    }
    if (count > 0) {
        addItem(weight * count, value * count);
    }
}
```

  - 处理后直接对产生的新物品，使用0-1背包进行求解即可

多重背包

- 二进制拆分优化，时间复杂度
  - 这样第 i 种物品我们就分成了 $O(\log C_i)$ 种物品
  - 原问题的时间复杂度降为 $O(V\times\sum\log C_i)$ 的 01 背包问题
  - 已经有了很大的改进



4 分组背包问题

Grouped knapsack problem

分组背包

- 分组背包:
  - 有 N 件物品和一个容量为 V 的背包，第 i 种物品的体积是 w_i ，价值 v_i, 所有的物品划分成若干组，每个组里面的物品最多选一件。求解将哪些物品装入背包可使这些物品的体积总和不超过背包容量，且价值总和最大。
  - 特点是：每种物品有 1 件，每组只能选 1 件。
  - 样例

组号	体积	价值
1	5	200
	3	100
2	7	300

- Ans = 400 （只选第 2、3 个）

分组背包

- 这个题和0-1背包有什么不同？
- 原来是一个物品选还是不选，现在是一个组选哪种物品
- 设计状态
  - f[i][j] 表示，只考虑前 i 组，背包容量为 j 时最大价值
- 状态转移方程：

$$f_{i,j} = max(f_{i-1,j}, f_{i-1,j-w_k} + v_k | k\text{为第}i\text{组的物品})$$

分组背包

- 代码：

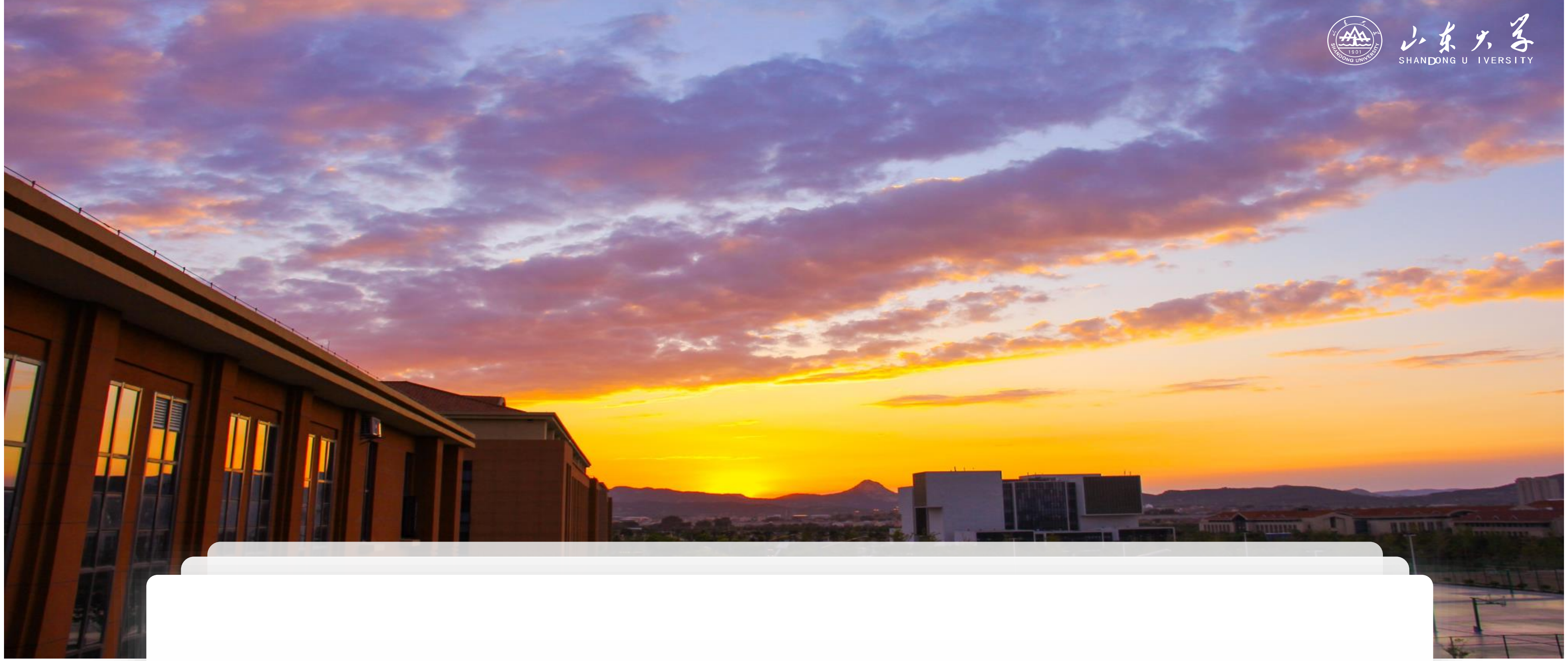
```
for (int i = 1; i <= numGroup; i++) {
    for (int j = 1; j <= V; j++) {
        f[i][j] = f[i - 1][j];
        for (int k : group[i]) {
            if (j - w[k] >= 0) f[i][j] = max(f[i][j], f[i - 1][j - w[k]] + v[k]);
        }
    }
}
```

分组背包

- 时间复杂度为 O(N×V)。
- 注意，虽然是三重循环，但是 k 和 i 的总枚举量也是N
- 同样也可以采用“滚动数组”的方式，优化空间复杂度

滚动数组：

```
for (int i = 1; i <= numGroup; i++) {
    for (int j = V; j >= 1; j--) {
        for (int k : group[i]) {
            if (j - w[k] >= 0) f[j] = max(f[j], f[j - w[k]] + v[k]);
        }
    }
}
```



5 超 大 背 包 问 题

Super knapsack problem

超大背包

- 超大背包：
  - 有 N 件物品和一个容量为 V 的背包。第 i 种物品的体积是 W_i，价值 V_i。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。
  - 约束：$N <= 40, W_i <= 10^{15}, V_i <= 10^{15}$
  - 特点是：0-1 背包问题变种，体积巨大。与 DP 无关！
  - 样例

3 个物品，容量为10

编号	体积w	价值v
1	5	200
2	4	100
3	7	300

Ans = 300 （前两个或只选第三个）

超大背包

- 超大背包：
  - 要注意此时的 N 很小，但 V 很大
  - 如果依然考虑 0-1 背包的思路的话，无论是时间复杂度 $O(N\times V)$，还是空间复杂度 $O(V)$，都是无法承受的

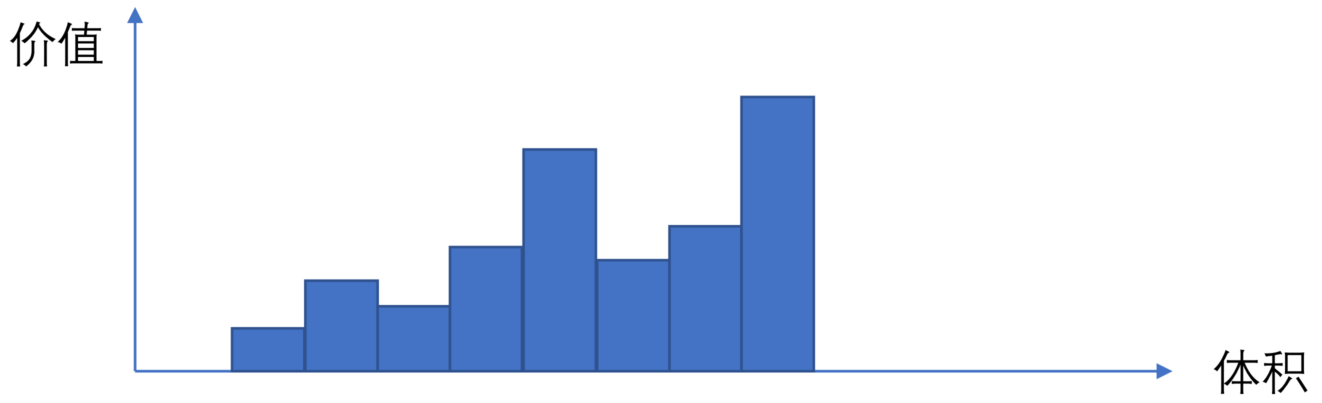
  - 回归到最朴素的想法：
    - 枚举 N 的所有子集，在所有子集中选取体积合法，价值最大的子集
    - 时间复杂度 $O(2^N)$
    - 显然还需要进一步优化 ($N <= 40$)

超大背包

- 一种全新的思路：
  - 首先将物品分成两组，每组 N/2 个物品
  - 对这两组物品分别进行子集枚举，得到一系列的<w1,v1>二元组，代表该集合的总体积，该集合的总价值
  - 考虑单个组：
    - 将所有的二元组排序，按照怎样的顺序排？
    - 如果 a[i].w1 > a[j].w1，且 a[i].v1 < a[j].v1，那么 a[i] 就可以舍去，因为 a[j] 更 “物 美 价 廉”
    - 所以剩下的所有二元组，一定满足 a[i].w1 < a[j].w1 且 a[i].v1 < a[j].v1。这样就可以进行排序了

超大背包

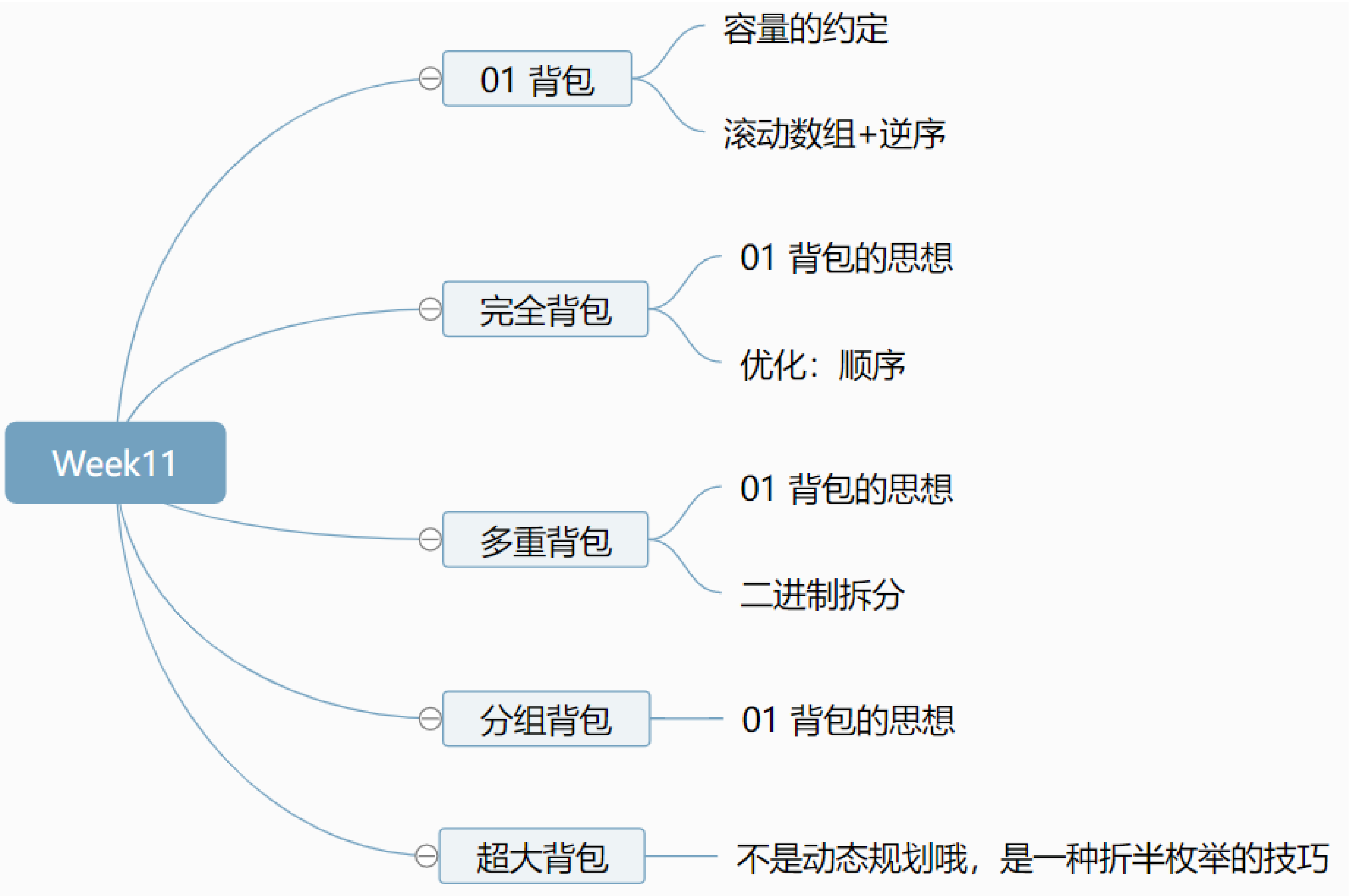
- 一种全新的思路：
  - 首先将物品分成两组，每组 N/2 个物品
  - 对这两组物品分别进行子集枚举，得到一系列的<w1,v1>二元组，代表该集合的总体积，该集合的总价值
  - 考虑单个组：



超大背包

- 一种全新的思路：
  - 对于排好序的两个组，假设分别为组 1 和组 2
  - 对于组1的一个二元组，价值vi，体积wi，如果选取，则存在V-wi的剩余空间，我们可以从组2中选取不超过V-wi空间且价值尽可能大的二元组补充
  - 对组1每一个二元组都找到对应组2的二元组，得到价值总和，求最大值
- 计算时间复杂度：
  - 分组+枚举的总量为 $2 \times 2^{(N/2)} \times (N/2) = N \times 2^{(N/2)}$
  - 后续枚举+二分的总量为 $2^{(N/2)} \times \log(2^{(N/2)}) = (N/2) \times 2^{(N/2)}$
  - 因此时间复杂度为 $O(N \times 2^{(N/2)})$

总结


Week11
01 背包
容量的约定
滚动数组+逆序
完全背包
01 背包的思想
优化：顺序
多重背包
01 背包的思想
二进制拆分
分组背包
01 背包的思想
超大背包
不是动态规划哦，是一种折半枚举的技巧

感谢收听

Thank You For Your Listening

参考 2019 年程序设计思维与实践PPT、《背包九讲》