背包dp

TEACHING COURSEWARE POWPOINT

授课时间：2025.08.12

目录

01

概念

TEACHING
COURSEWARE

TEACH

概念

背包问题是一类经典的组合优化问题，属于动态规划（DP）的核心研究对象。其严格定义可以概括为：

给定一组物品，每个物品有两个核心属性：

重量（weight）：通常用 w[i] 表示第 i 个物品的重量（或消耗的资源量）；

价值（value）：通常用 v[i] 表示第 i 个物品的价值（或带来的收益）。

同时给定一个背包容量，用 C 表示（即背包最多能承载的总重量）。

02

01背包

TEACHING
COURSEWARE

TEACH

01背包

说明

一个旅行者有一个最多能装M公斤的背包，现在有n件物品，它们的重量分别是W_1，W_2，...,W_n.它们的价值分别为C_1,C_2,...,C_n，求旅行者能获得最大总价值。

输入格式

第一行:两个整数，M(背包容量，$M \leq 200$)和N(物品数量，$N \leq 30$);第2..$N+1$行:每行二个整数W，C，表示每个物品的重量和价值。

输出格式

仅一行，一个数，表示最大总价值。

样例

输入数据 1

Copy

```
10 4
2 1
3 3
4 5
7 9
```

输出数据 1

Copy

```
12
```

01背包

状态定义

设dp[i][j]表示“考虑前 i 种物品，且背包容量为 j 时，能获得的最大价值”。

其中 i 的范围是 0 到 n（0 表示不考虑任何物品），j 的范围是 0 到 m（0 表示背包容量为 0）。

状态转移逻辑

对于第 i 种物品，有两种决策：不选择第 i 种物品：此时背包内的物品仍为前 i-1 种，最大价值与dp[i-1][j]相等，即dp[i][j] = dp[i-1][j]。

选择第 i 种物品：需满足背包容量 j ≥ 物品重量 w [i]（能装下该物品）。此时的最大价值为 “放入该物品前的最大价值” 加上 “该物品的价值”，即dp[i-1][j - w[i]] + c[i]。这里用i-1是因为第 i 种物品只能选一次，放入后就不能再选，需基于前 i-1 种物品的状态计算。

最终取两种决策的最大值：dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i]] + c[i])（当 j ≥ w [i] 时）。

01背包

```cpp
#include<bits/stdc++.h>
using namespace std;
int w[35], c[35];
int dp[35][205];
int main() {
    int m, n;  // m:背包容量，n:物品数量
    cin >> m >> n;
    // 读入物品的重量和价值（1~n下标）
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> c[i];
    }
    // 遍历前i个物品
    for (int i = 1; i <= n; i++) {
        // 遍历背包容量（0~m）
        for (int j = 0; j <= m; j++) {
            // 不选第i个物品
            dp[i][j] = dp[i-1][j];
            // 选第i个物品（如果容量足够）
            if (j >= w[i]) {
                dp[i][j] = max(dp[i][j], dp[i-1][j - w[i]] + c[i]);
            }
        }
    }
```

01背包


```
    // 输出最大价值
    cout << dp[n][m] << endl;
    return 0;
}
```

01背包

优化：
在一维优化中，我们定义：
dp[j] 表示 “当前考虑过的物品中，容量为 j 的背包能装下的最大价值”。

需要注意：随着外层循环（遍历物品）的推进，dp[j] 的含义会动态更新 —— 每处理完一个物品，dp[j] 就表示 “考虑前 i 个物品时，容量 j 的最大价值”。

01背包

回顾二维数组的状态转移：
`dp[i][j] = max(dp[i-1][j], dp[i-1][j - w[i]] + c[i])`
观察发现：

计算 `dp[i][j]` 只需要 上一行（`i-1` 层）的数据，不需要更早的行（如 `i-2`、`i-3` 等）。

二维数组中，第 `i` 行的所有数据只依赖第 `i-1` 行，因此可以用一个一维数组 “覆盖” 上一行的数据，无需保留完整的二维表。

01背包

状态定义
用dp[j]表示 “背包容量为 j 时能获得的最大价值”。
一维数组需通过反向遍历确保每个物品仅被选择一次。
状态转移方程
对于第 i 种物品，内层循环从容量 m 反向遍历至物品重量 w [i]，此时：
dp[j] = max(dp[j], dp[j - w[i]] + c[i])
反向遍历的原因：若正向遍历，dp[j - w[i]]会先被更新（包含第 i 种物品），导致同一物品被多次选择（违背 01 背包约束）；反向遍历则保证dp[j - w[i]]仍为上一轮（i-1 种物品）的状态，确保物品只选一次。

01背包


```cpp
#include<bits/stdc++.h>
using namespace std;
// 0-1背包问题：每个物品只能选择放或不放，求在背包容量限制下的最大价值
// w[i]：第i个物品的重量
// c[i]：第i个物品的价值
int w[35], c[35];
// dp[j]：表示背包容量为j时，能获得的最大价值
// 一维数组优化的核心：复用数组空间，通过反向遍历避免重复使用同一物品
int dp[205];
int main() {
    int m, n;
    // m：背包的最大容量
    // n：物品的数量
    cin >> m >> n;
    // 输入每个物品的重量和价值（从1开始编号，方便后续处理）
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> c[i];
    }
```

01背包


```
    // 遍历每个物品（第i个物品）
    for (int i = 1; i <= n; i++) {
        // 从背包最大容量m开始，反向遍历到当前物品的重量w[i]
        // 反向遍历的目的：确保每个物品只被使用一次（避免重复选择）
        for (int j = m; j >= w[i]; j--) {
            // 状态转移方程：
            // 对于容量j，有两种选择：
            // 1. 不放第i个物品：价值保持为dp[j]
            // 2. 放第i个物品：价值为 dp[j - w[i]]（放物品前的最大价值） + c[i]（当前物品价值）
            // 取两种选择中的最大值作为新的dp[j]
            dp[j] = max(dp[j], dp[j - w[i]] + c[i]);
        }
    }
    // 输出背包容量为m时的最大价值
    cout << dp[m] << endl;
    return 0;
}
```

03

完全背包

TEACHING
COURSEWARE

TEACH

完全背包

说明

设有n种物品，每种物品有一个重量及一个价值。但每种物品的数量是无限的，同时有一个背包，最大载重量为M，今从n种物品中选取若干件(同一种物品可以多次选取)，使其重量的和小于等于M，而价值的和为最大。

输入格式

第一行:两个整数，M(背包容量，$M \leq 200$)和N(物品数量，$N \leq 30$);第2..$N+1$行:每行二个整数W_i,C_i，表示每个物品的重量和价值。

输出格式

仅一行，一个数，表示最大总价值。

样例

输入数据 1

Copy

```
10 4
2 1
3 3
4 5
7 9
```

输出数据 1

Copy

```
max=12
```

完全背包

状态定义

设dp[i][j]表示 "考虑前 i 种物品，背包容量为 j 时能获得的最大价值"。

状态转移

对于第 i 种物品，有两种选择：

不选第 i 种物品：价值与只考虑前 i-1 种物品相同，即dp[i][j] = dp[i-1][j]。

选第 i 种物品：若背包容量 j ≥ 物品重量 w [i]，则可选择该物品（且可重复选），价值为 dp[i][j-w[i]] + c[i]（选择后仍可继续选第 i 种物品）。

取两种选择的最大值：dp[i][j] = max(不选的价值, 选的价值)。

初始状态

当 i=0（无物品）或 j=0（容量为 0）时，最大价值均为 0，即dp[0][j] = 0和dp[i][0] = 0。

关键区别：

完全背包允许重复选

而 0-1 背包禁止重复选

完全背包


```cpp
#include <iostream>
#include <algorithm>
using namespace std;
// 全局数组定义在main函数外部
int w[31];  // 存储物品重量，下标1~n对应n个物品
int c[31];  // 存储物品价值，下标1~n对应n个物品
int dp[31][201];  // dp[i][j]表示前i种物品在容量j时的最大价值
int main() {
    int m, n;  // m:背包最大容量，n:物品总数
    cin >> m >> n;
    // 输入n个物品的重量和价值
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> c[i];
    }
    // 外层循环：依次考虑第1到第n种物品
```

完全背包


```
    // 外层循环：依次考虑第1到第n种物品
    for (int i = 1; i <= n; i++) {
        // 内层循环：依次计算容量1到m的最大价值
        for (int j = 1; j <= m; j++) {
            // 情况1：不选择第i种物品，价值等于前i-1种物品的结果
            dp[i][j] = dp[i-1][j];
            // 情况2：选择第i种物品（可多次选），需满足容量足够
            if (j >= w[i]) {
                // 与0-1背包的区别：这里用dp[i][j-w[i]]
                // 表示选择第i种物品后，仍可继续选择该物品
                dp[i][j] = max(dp[i][j], dp[i][j - w[i]] + c[i]);
            }
        }
    }
    // 输出最大价值，格式带"max="前缀
    cout << "max=" << dp[n][m] << endl;
    return 0;
}
```

完全背包

状态定义
用 dp [j] 表示 "背包容量为 j 时能获得的最大价值"。
一维数组需通过正向遍历允许物品被多次选择。
状态转移方程
对于第 i 种物品，内层循环从物品重量 w [i] 正向遍历至容量 m，此时：
dp [j] = max (dp [j], dp [j - w [i]] + c [i])
正向遍历的原因：若正向遍历，当计算 dp [j] 时，dp [j - w [i]] 已经是本轮（考虑第 i 种物品）更新过的状态，这意味着可以包含之前选择的第 i 种物品，从而实现了同一物品的多次选择；正向遍历保证了物品可以被重复选取

完全背包

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int w[31];  // 存储物品重量
int c[31];  // 存储物品价值
int dp[201];  // 优化为一维数组，dp[j]表示容量j时的最大价值
int main() {
    int m, n;  // m:背包最大容量，n:物品总数
    cin >> m >> n;
    // 输入n个物品的重量和价值
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> c[i];
    }
    // 外层循环：依次考虑每个物品
    for (int i = 1; i <= n; i++) {
        // 内层循环：正序遍历容量（完全背包允许物品重复选择）
        for (int j = w[i]; j <= m; j++) {
            // 比较不选和选当前物品的最大价值
            dp[j] = max(dp[j], dp[j - w[i]] + c[i]);
        }
    }
    // 输出最大价值
    cout << "max=" << dp[m] << endl;
    return 0;
}
```

04

总结

TEACHING
COURSEWARE

TEACH