背包 DP

前置知识：动态规划部分简介。

引入

在具体讲何为「背包 dp」前，先来看如下的例题：

> **「USACO07 DEC」Charm Bracelet**
>
> 题意概要：有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」。

0-1 背包

解释

例题中已知条件有第 i 个物品的重量 w_i，价值 v_i，以及背包的总容量 W。

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i}+v_i$。

由此可以得出状态转移方程：

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

> 这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j=\max\left(f_j,f_{j-w_i}+v_i\right)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

实现

还有一点需要注意的是，很容易写出这样的 **错误核心代码**：

C++

```cpp
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
// 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
// f[i][l + w[i]]); 简化而来
```

Python

```python
for i in range(1, n + 1):
    for l in range(0, W - w[i] + 1):
        f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
# 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
# f[i][l + w[i]]) 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$，在 $j \geqslant w_i$ 时，$f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新。

因此实际核心代码为

C++

```cpp
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

Python

```python
for i in range(1, n + 1):
    for l in range(W, w[i] - 1, -1):
        f[l] = max(f[l], f[l - w[i]] + v[i])
```

例题代码

```cpp
#include <iostream>
using namespace std;
constexpr int MAXN = 13010;
int n, W, w[MAXN], v[MAXN], f[MAXN];

int main() {
  cin >> n >> W;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];  // 读入数据
  for (int i = 1; i <= n; i++)
    for (int l = W; l >= w[i]; l--)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
// 状态方程
  cout << f[W];
  return 0;
}
```

完全背包

解释

完全背包模型与 0-1 背包类似，与 0-1 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 0-1 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。

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

过程

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。

状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{+\infty}(f_{i-1,j-k\times w_i} + v_i \times k)$$

考虑做一个简单的优化。可以发现，对于 $f_{i,j}$，只要通过 $f_{i,j-w_i}$ 转移就可以了。因此状态转移方程为：

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

> 理由是当我们这样转移时，$f_{i,j-w_i}$ 已经由 $f_{i,j-2\times w_i}$ 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。换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度。

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度。如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）。

「Luogu P1616」疯狂的采药

题意概要：有 n 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

例题代码

```cpp
#include <iostream>
using namespace std;
constexpr int MAXN = 1e4 + 5;
constexpr int MAXW = 1e7 + 5;
int n, W, w[MAXN], v[MAXN];
long long f[MAXW];

int main() {
  cin >> W >> n;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
  for (int i = 1; i <= n; i++)
    for (int l = w[i]; l <= W; l++)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
// 核心状态方程
  cout << f[W];
  return 0;
}
```

多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有 k_i 个，而非一个。

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 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$$f_{i,j}=\max_{k=0}^{k_i}(f_{i-1,j-k\times w_i}+v_i\times k)$$

时间复杂度 $O(W\sum_{i=1}^{n}k_i)$。

核心代码

```
for (int i = 1; i <= n; i++) {
  for (int weight = W; weight >= w[i]; weight--) {
    // 多遍历一层物品数量
    for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {
      dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *
v[i]);
    }
  }
}
```

二进制分组优化

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$ 处入手。为了表述方便，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。

过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令 $A_{i,j}\,(j \in [0, \lfloor \log_2(k_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。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。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度 $O(W \sum_{i=1}^{n} \log_2 k_i)$

实现

二进制分组代码

C++

```cpp
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

Python

```python
index = 0
for i in range(1, m + 1):
    c = 1
    p, h, k = map(int, input().split())
    while k > c:
        k -= c
        index += 1
        list[index].w = c * p
        list[index].v = c * h
        c *= 2
    index += 1
    list[index].w = p * k
    list[index].v = h * k
```

单调队列优化

见 单调队列/单调栈优化。

习题：「Luogu P1776」宝物筛选_NOI 导刊 2010 提高（02）

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取 k 次。

> 这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

```
for（循环物品种类）{
  if（是 0 - 1 背包）
    套用 0 - 1 背包代码;
  else if（是完全背包）
    套用完全背包代码;
  else if（是多重背包）
    套用多重背包代码;
}
```

例题

「Luogu P1833」樱花

有 n 种樱花树和长度为 T 的时间，有的樱花树只能看一遍，有的樱花树最多看 A_i 遍，有的樱花树可以看无数遍。每棵樱花树都有一个美学值 C_i，求在 T 的时间内看哪些樱花树能使美学值最高。

核心代码

```
for (int i = 1; i <= n; i++) {
  if (cnt[i] == 0) {  // 如果数量没有限制使用完全背包的核心代码
    for (int weight = w[i]; weight <= W; weight++) {
      dp[weight] = max(dp[weight], dp[weight - w[i]] + v[i]);
    }
  } else {  // 物品有限使用多重背包的核心代码，它也可以处理0-1背包问题
    for (int weight = W; weight >= w[i]; weight--) {
      for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {
        dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *
v[i]);
      }
    }
  }
}
```

习题：HDU 5410 CRB and His Birthday

二维费用背包

「Luogu P1855」榨取 kkksc03

有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，产生 c_i 元的开支。

现在有 T 分钟时间，W 元钱来处理这些任务，求最多能完成多少任务。

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

这时候就要注意，再开一维存放物品编号就不合适了，因为容易 MLE。

实现

C++

```cpp
for (int k = 1; k <= n; k++)
  for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
    for (int j = t; j >= ti; j--)  // 对时间进行一层枚举
      dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

Python

```python
for k in range(1, n + 1):
    for i in range(m, mi - 1, -1):  # 对经费进行一层枚举
        for j in range(t, ti - 1, -1):  # 对时间进行一层枚举
            dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1)
```

分组背包

「Luogu P1757」通天之分组背包

有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i，体积为 v_i。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个。

实现

C++

```cpp
for (int k = 1; k <= ts; k++)           // 循环每一组
  for (int i = m; i >= 0; i--)          // 循环背包容量
    for (int j = 1; j <= cnt[k]; j++)   // 循环该组的每一个物品
      if (i >= w[t[k][j]])              // 背包容量充足
        dp[i] = max(dp[i],
                    dp[i - w[t[k][j]]] + c[t[k][j]]);  // 像0-1背包一样状态转移
```

Python

```python
for k in range(1, ts + 1):  # 循环每一组
    for i in range(m, -1, -1):  # 循环背包容量
        for j in range(1, cnt[k] + 1):  # 循环该组的每一个物品
            if i >= w[t[k][j]]:  # 背包容量充足
                dp[i] = max(
                    dp[i], dp[i - w[t[k][j]]] + c[t[k][j]]
                )  # 像0-1背包一样状态转移
```

这里要注意：**一定不能搞错循环顺序**，这样才能保证正确性。

有依赖的背包

「Luogu P1064」金明的预算方案

金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i，重要度为 p_i。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。

目标是让所有购买的物品的 $v_i \times p_i$ 之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

泛化物品的背包

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为 V 的背包问题中，当分配给它的费用为 v_i 时，能得到的价值就是 $h\left(v_i\right)$。这时，将固定的价值换成函数的引用即可。

杂项

小优化

> 根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品 i, j 且 i 的价值大于 j 的价值并且 i 的费用小于 j 的费用时，只需保留 i。

背包问题变种

输出方案

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用 $g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。输出时的伪代码：

```
int v = V;  // 记录当前的存储空间

// 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for（从最后一件循环至第一件）{
  if（g[i][v]）{
    选了第 i 项物品;
    v -= 第 i 项物品的重量;
  } else {
    未选第 i 项物品;
  }
}
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum(dp_i, dp_{i-c_i})$$

初始条件：$dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。

求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数。

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值。

$g_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数。

转移方程：

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}\neq f_{i-1,j-v}+w$ 说明我们此时不选择把物品放入背包更优，方案数由 $g_{i-1,j}$ 转移过来，

如果 $f_{i,j}\neq f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明我们此时选择把物品放入背包更优，方案数由 g_{i-1} 转移过来，

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明放入或不放入都能取得最优解，方案数由 $g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来。

初始条件：

```
memset(f, 0x3f3f, sizeof(f));  // 避免没有装满而进行了转移
f[0] = 0;
g[0] = 1;  // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是 f_m。

最后我们通过找到最优解的价值，把 g_j 数组里取到最优解的所有方案数相加即可。

实现

```
for (int i = 0; i < N; i++) {
  for (int j = V; j >= v[i]; j--) {
    int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
    int c = 0;
    if (tmp == dp[j]) c += cnt[j];                        // 如果从
dp[j]转移
    if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]];  // 如果从
dp[j-v[i]]转移
    dp[j] = tmp;
    cnt[j] = c;
  }
}
int max = 0;  // 寻找最优解
for (int i = 0; i <= V; i++) {
  max = std::max(max, dp[i]);
}
int res = 0;
for (int i = 0; i <= V; i++) {
  if (dp[i] == max) {
    res += cnt[i];  // 求和最优解方案数
  }
}
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法。 具体来讲：$dp_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $dp_{i,j}$ 扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是 $dp_{i,j}=\max(dp_{i-1,j},dp_{i-1,j-v_i}+w_i)$，现在我们则是要合并 $dp_{i-1,j}$，$dp_{i-1,j-v_i}+w_i$ 这两个大小为 k 的递减序列，并保留合并后前 k 大的价值记在 $dp_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的。

例题 HDU 2639 Bone Collector II

求 0-1 背包的严格第 k 优解。$n\le 100, v\le 1000, k\le 30$

实现

```
memset(dp, 0, sizeof(dp));
int i, j, p, x, y, z;
scanf("%d%d%d", &n, &m, &K);
for (i = 0; i < n; i++) scanf("%d", &w[i]);
for (i = 0; i < n; i++) scanf("%d", &c[i]);
for (i = 0; i < n; i++) {
  for (j = m; j >= c[i]; j--) {
    for (p = 1; p <= K; p++) {
      a[p] = dp[j - c[i]][p] + w[i];
      b[p] = dp[j][p];
    }
    a[p] = b[p] = -1;
    x = y = z = 1;
    while (z <= K && (a[x] != -1 || b[y] != -1)) {
      if (a[x] > b[y])
        dp[j][z] = a[x++];
      else
        dp[j][z] = b[y++];
      if (dp[j][z] != dp[j][z - 1]) z++;
    }
  }
}
printf("%d\n", dp[m][K]);
```

参考资料与注释

- 背包问题九讲 - 崔添翼。

本页面最近更新：2025/8/9 14:37:03，更新历史

发现错误？想一起完善？ 在 GitHub 上编辑此页！

本页面贡献者：Ir1d, sshwy, StudyingFather, Marcythm, partychicken, H-J-Granger, NachtgeistW, countercurrent-time, Enter-tainer, weiranfu, greyqz, iamtwz, Konano, ksyx, ouuan, paigeman, Tiphereth-A, wolfdan666, AngelKitty, CCXXXI, cjsoft, diauweb, Early0v, ezoixx130, GekkaSaori, GoodCoder666, Henry-ZHR, HeRaNO, Link-cute, LovelyBuggies, LuoshuiTianyi, Makkiy, mgt, minghu6, odeinjul, oldoldtea, P-Y-Y, PotassiumWings, SamZhangQingChuan, shenshuaijie, Suyun514, weiyong1024, Xeonacid, xyf007, Alisahhh, Alphnia, CBW2007, dhbloo, fps5283, GavinZhengOI, Gesrua, hsfzLZH1, hydingsy, kenlig, kxccc, lychees, Menci, Peanut-Tang, PlanariaIce, sbofgayschool, shawlleyw, Siyuan, SukkaW, tLLWtG, WAAutoMaton, x4Cx58x54, xk2013, zhb2000, zhufengning

© 本页面的全部内容在 **CC BY-SA 4.0 和 SATA** 协议之条款下提供，附加条款亦可能应用