0-1 背包问题与完全背包问题选讲

0 - 1 背包问题：LeetCode 416、474、494；
完全背包问题：LeetCode 322、518

美西时间 6:30 pm（2020.05.22）
美东时间 9:30 pm（2020.05.22）
北京时间 9:30 am（2020.05.23）

0-1 背包问题

提示：背包问题在绝大多数情况下只需要学习到「0-1 背包」和「完全背包」。

- 0-1 背包问题是「动态规划」的典型问题、入门问题
- 熟练掌握「0 - 1 背包」问题：对于理解「动态规划」，通过「表格法」，从最小 规模问题开始，逐步得到最终规模问题，并且记录中间过程的思想，有很大帮助
- 「0 - 1 背包」问题是掌握复 杂的背包问题的基础

书本下载
链接：https://pan.baidu.com/s/1XZZGBK4P_zZ4rJvrIRG5VA
密码：sjop

参考资料：

1、《背包九讲》；
2、在 AcWing 网上上练习背包问题：https://www.acwing.com/problem/；
3、《算法图解》**0-1 背包**问题讲得很生动（P134）；
4、《挑战程序设计竞赛》**完全背包**问题推导是正确的，排版错误（P57）。

背包问题九讲 2.0 alpha1

崔添翼 (Tianyi Cui, a.k.a. dd_engi)

September 15, 2011

本文题为《背包问题九讲》，从属于《动态规划的思考艺术》系列。

这系列文章的第一版于2007年下半年使用EmacsMuse制作，以HTML格式发布到网上，转载众多，有一定影响力。

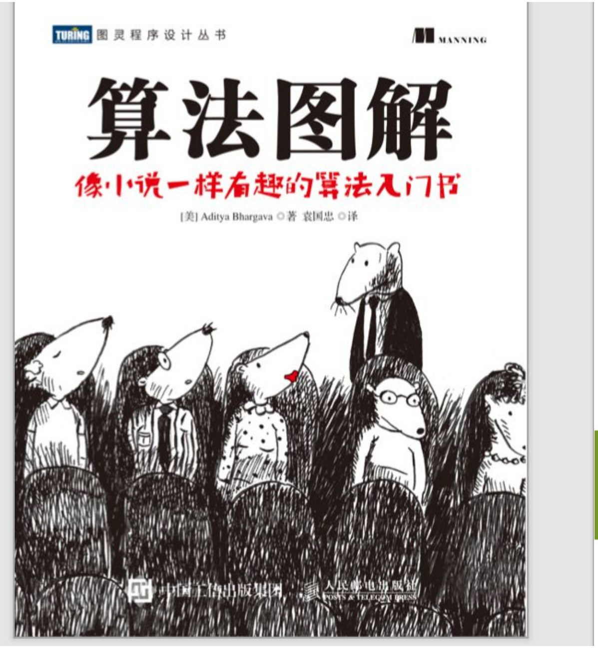
2011年9月，本系列文章由原作者用LaTeX重新制作并全面修订，您现在看到的是2.0 alpha1版本，修订历史及最新版本请访问 https://github.com/tianyicui/pack 查阅。

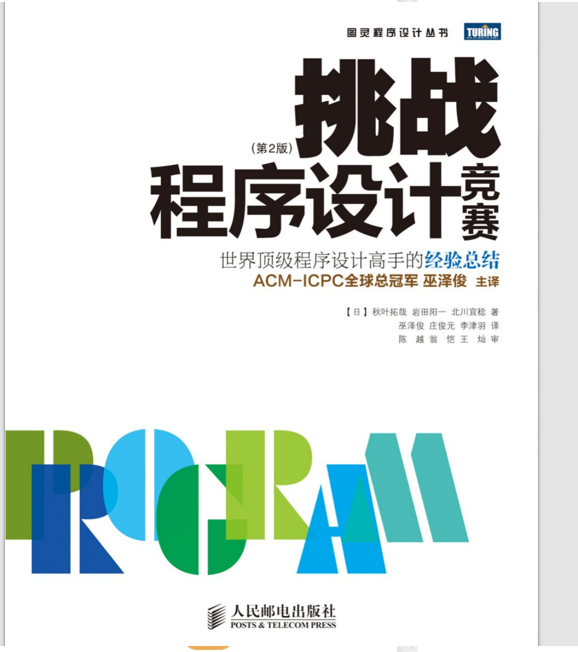
AcWing 首页 题库 题解 分享 问答 活动 应用 吐槽 登录/注册

AcWing题库

搜索题号、标题、题目来源、算法、题目描述

#	标题
1	A + B
2	01背包问题
3	完全背包问题
4	多重背包问题 I
5	多重背包问题 II
6	多重背包问题 III
7	混合背包问题
8	二维费用的背包问题
9	分组背包问题
10	有依赖的背包问题
11	背包问题求方案数
12	背包问题求具体方案





0-1 背包问题

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 w_i，价值是 v_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

- 贪心算法无效
- 回溯搜索算法复杂度太高

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])。 **选与不选**

0-1 背包问题

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])。

代码版本 1：

```
    // dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；
    // 因为包含价值为 0 的计算，所以 + 1
    int[][] dp = new int[N][V + 1];

    // 先写第 1 行
    for (int j = 1; j <= V; j++) {
        // 第 1 个物品的体积要小于等于背包容量
        if (weight[0] <= j) {
            dp[0][j] = value[0];
        }
    }

    for (int i = 1; i < N; i++) {
        for (int j = 0; j <= V; j++) {
            dp[i][j] = dp[i - 1][j];
            if (weight[i] <= j) {
                dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - weight[i]] + value[i]);
            }
        }
    }

    // 输出
    System.out.println(dp[N - 1][V]);
}
```

0-1 背包问题

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])。

代码版本 1’:

```
// 多开一行，避免对第 1 行单独赋值
// 后面要注意下标，有下标 weight[i] 和 value[i] 的地方都要减 1
int[][] dp = new int[N + 1][V + 1];

for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        dp[i][j] = dp[i - 1][j];
        if (weight[i - 1] <= j) {
            dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
        }
    }
}
// 输出
System.out.println(dp[N][V]);
```

0-1 背包问题

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])。

优化空间：由于只关心表格最后一行最后一格的 值。

1、滚动数组：下标为 i 的行只参考了下标为 i - 1 的行；
2、倒序填表：下标为 i 的行只参考了下标为 i - 1 的行，并且参考的区域在左上方。
保证每一个值都能参考到正确的值。

价值（数值）		0	1	2	3	4	5	6	7	8	9	10	11
下标 0 的体积	1												
下标 1 的体积	5												
下标 2 的体积	11												
下标 3 的体积	5												

0-1 背包问题

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = dp[i - 1][j], dp[i - w[i]] + v[i]

代码版本 2：

```java
// 优化空间的写法：
int[] dp = new int[V + 1];
for (int i = 1; i <= N; i++) {
    for (int j = V; j >= weight[i - 1]; j--) {
        dp[j] = Math.max(dp[j], dp[j - weight[i - 1]] + value[i - 1]);
    }
}

// 输出
System.out.println(dp[V]);
```

注意：倒序填表会丢失信息，如果需要反推一个合理的 选择路径，这种方法失效。

完全背包问题

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，每种物品都有无限件可用。

第 i 件物品的体积是 w_i，价值是 v_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

状态定义：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态转移方程：dp[i][j] = max(dp[i - 1][j - k · w[i]] + k · v[i])，这里 k >= 0。

优化的状态转移方程： dp[i][j] = max(dp[i - 1][j], dp[i][j - w[i]] + v[i])

0-1 背包问题与完全背包问题比较

「0-1背包」问题：

状态转移方程：dp[i][j] = max(dp[i − 1][j], dp[i − 1][j − w[i]] + v[i])。

「完全背包」问题优化的状态转移方程：

dp[i][j] = max(dp[i − 1][j], dp[i][j − w[i]] + v[i])

区间只在红色标出来的地方：「0 - 1」背包参考上一行，「完全背包」参考当前行。

完全背包问题

代码版本 1：不符合面试要求

```
// dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；
// 因为包含价值为 0 的计算，所以 + 1
int[][] dp = new int[N][V + 1];
// 先写第 1 行
for (int k = 0; k * weight[0] <= V; k++) {
    dp[0][k * weight[0]] = k * value[0];
}

// 最朴素的做法
for (int i = 1; i < N; i++) {
    for (int j = 0; j <= V; j++) {
        // 多一个 for 循环，枚举下标为 i 的物品可以选的个数
        for (int k = 0; k * weight[i] <= j; k++) {
            dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - k * weight[i]] + k * value[i]);
        }
    }
}
// 输出
System.out.println(dp[N - 1][V]);
```

完全背包问题状态转移推导：dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；

状态定义：dp[i][j] = max(dp[i - 1][j - k · w[i]] + k · v[i])，这里 k >= 0。 ①

单独把 k == 0 拿出来，作为一个 max 的比较项。

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - k · w[i]] + k · v[i])，这里 k >= 1。 ②

max(dp[i - 1][j - k · w[i]] + k · v[i])，k >= 1。 把 v[i] 单独拿出来。

= v[i] + max(dp[i - 1][j - k · w[i]] + (k - 1) · v[i])，k >= 1。③

将 ① 中左边的 j 用 j - w[k] 代入。

dp[i][j - w[i]] = max(dp[i - 1][j - w[i] - k · w[i]] + k · v[i])，这里 k >= 0。

= max(dp[i - 1][j - (k + 1)· w[i]] + k · v[i])，这里 k >= 0。

= max(dp[i - 1][j - k· w[i]] + (k - 1) · v[i])，这里 k >= 1。④

结合 ②、③ 和 ④，推出 dp[i][j] = max(dp[i - 1][j], dp[i][j - w[i]]) + v[i]。

完全背包问题

代码版本 2：符合面试要求

```java
// dp[i][j] 表示考虑物品区间 [0, i] 里，不超过背包容量，能够获得的最大价值；
// 因为包含价值为 0 的计算，所以 + 1
int[][] dp = new int[N + 1][V + 1];
// 优化
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= V; j++) {
        // 至少是上一行抄下来
        dp[i][j] = dp[i - 1][j];
        if (weight[i - 1] <= j){
            dp[i][j] = Math.max(dp[i][j], dp[i][j - weight[i - 1]] + value[i - 1]);
        }
    }
}
// 输出
System.out.println(dp[N][V]);
```

完全背包问题

代码版本 3：优化空间

```
int[] dp = new int[V + 1];
// 先写第 1 行

// 优化空间
for (int i = 1; i <= N; i++) {
    // 细节，j 从 weight[i - 1] 开始遍历
    for (int j = weight[i - 1]; j <= V; j++) {
        dp[j] = Math.max(dp[j], dp[j - weight[i - 1]] + value[i - 1]);
    }
}
// 输出
System.out.println(dp[V]);
```

问题选讲

0 - 1 背包问题：

LeetCode 416：非常典型的 0-1 背包问题；
LeetCode 474：约束有 2 个；
LeetCode 494：发现等价关系；

完全背包问题：特点：一个元素可以使用多个，且不 计算顺序。**LeetCode 377 不能套完全背包。**

LeetCode 322：三种方法：
1、直接推状态转移方程；
2、BFS；
3、套完全背包模式（内外 层顺序正好相反）

LeetCode 518：建议直接推公式，做等价 转换。

补充例题(普通公司面试可以不掌握):完全背包 问题，并且倒序得到路径。

1449. 数位成本和为目标值的最大数字

难度 困难 8

给你一个整数数组 `cost` 和一个整数 `target` 。请你返回满足如下规则可以得到的 **最大** 整数：

- 给当前结果添加一个数位（`i + 1`）的成本为 `cost[i]` （`cost` 数组下标从 0 开始）。
- 总成本必须恰好等于 `target` 。
- 添加的数位中没有数字 0 。

由于答案可能会很大，请你以字符串形式返回。

如果按照上述要求无法得到任何整数，请你返回 "0" 。

```java
public String largestNumber(int[] cost, int target) {
    // 套完全背包模型
    int[] dp = new int[target + 1];
    Arrays.fill(dp, val: -1);

    dp[0] = 0;
    for (int i = 1; i <= target; i++) {
        for (int c : cost) {
            if (i >= c && dp[i - c] != -1) {
                dp[i] = Math.max(dp[i], dp[i - c] + 1);
            }
        }
    }

    // 判断是否有结果
    if (dp[target] == -1) {
        return "0";
    }

    // 在有结果的前提下，倒序反推路径
    int t = target;
    StringBuilder res = new StringBuilder();
    while (t > 0) {
        int pre = -1;
        int choose = -1;

        for (int i = 9; i > 0; i--) {
            if (t >= cost[i - 1] && dp[t - cost[i - 1]] > pre) {
                pre = dp[t - cost[i - 1]];
                choose = i;
            }

        }
        res.append(choose);
        t -= cost[choose - 1];
    }
    return res.toString();
}
```