背包DP基础 2024SZUACM算法周训 (241130)

回放: https://meeting.tencent.com/crm/2ObWzaGaca

题单: https://vjudge.net/contest/676407

---

0. 前言

0.1 前置知识

递推、递归.

0.2 背包问题

背包问题的难度不在于 DP, 而在于通过建模, 将一个实际问题转化为背包问题. 具体地, 发现实际问题中的物品、背包、重量 (或 "体积")、价值等对应关系.

下面讲解, 如何用**动态规划** (Dynamic Programming, DP) 求解最基础的 3 类背包问题: **0-1 背包**、**完全背包**、**多重背包**, 以及常用的背包问题的优化.

---

1. 0-1 背包

1.1 例题

1.1.1 [USACO07DEC] Charm Bracelet S

链接: https://www.luogu.com.cn/problem/P2871

题意 ($1\ \mathrm{s}$, $128\ \mathrm{MB}$)

有 $n\ \ (1\leq n\leq 3402)$ 个物品和一个容量为 $m\ \ (1\leq m\leq 12880)$ 的背包, 其中 $i\ \ (1\leq i\leq n)$ 号物品的重量为 $w_i\ \ (1\leq w_i\leq 400)$, 价值为 $d_i\ \ (1\leq d_i\leq 100)$. 每个物品只能放入背包一次.

求将哪些物品装入背包, 可使得物品重量之和不超过背包容量, 且价值之和最大. 输出最大总价值.

思路 I

每个物品是否装入背包是 $0/1$ 的状态, 故该模型称为 **0-1 背包**.

枚举每个物品是否装入背包, 计算每个方案的总价值, 总时间复杂度 $O(2^n\cdot n)$, 会 TLE .

$dp[i][j]$ 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值.

初始条件 $dp[0][0]=0$, 最终答案 $ans=\max\limits_{0\leq j\leq m}dp[n][j]$ 或 $ans=dp[n][m]$.

按是否选 i 号物品分类:

(1) 若不选 i 号物品, 则 $dp[i][j]=dp[i-1][j]$.

(2) 若选 i 号物品, 则 $dp[i][j]=dp[i-1][j-w_i]+d_i$, 条件为 $j\geq w_i$.

状态转移方程 $dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-w_i]+d_i\ \ (j\geq w_i)\}$.

最大总价值 $n\cdot\max d_i=3.4\mathrm{e}5$.

时间复杂度 $O(n\cdot m)$, 最大为 $3402\times 12880\approx 4.4\mathrm{e}7$.

空间复杂度 $O(n\cdot m)$, 最大为 $\dfrac{4\times 3402\times 12880}{1024\times 1024}\ \mathrm{MB}\approx 167\ \mathrm{MB}>128\ \mathrm{MB}$, 会 MLE.

代码 I (MLE)

```cpp
void solve() {
    int n, m;
    cin >> n >> m;

    vector<int> w(n + 5), d(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> d[i];
    }

    // dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
    vector<vector<int>> dp(n + 5, vector<int>(m + 5));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            dp[i][j] = dp[i - 1][j];

            if (j >= w[i]) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + d[i]);
            }
        }
    }

    cout << *max_element(dp[n].begin(), dp[n].end()) << '\n';
    // cout << dp[n][m] << '\n';
}

int main() {
    solve();
}
```

思路 II

注意到状态转移方程 $dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-w_i]+d_i\ \ (j\geq w_i)\}$ 中, $dp[i][]$ 只与 $dp[i-1][]$ 有关, 故无需记录 $dp[][]$ 数组的第一维, 滚动更新即可, 即**滚动数组优化**.

若直接去掉第一维, 状态转移的过程变为:

```
// dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
vector<int> dp(m + 5);
for (int i = 1; i <= n; i++) {
    for (int j = w[i]; j <= m; j++) {
        dp[j] = max(dp[j], dp[j - w[i]] + d[i]);
    }
}
```

分析上述代码的转移知: $j \geq w_i$ 时, $dp[i][j]$ 会被 $dp[i][j-w_i]$ 更新, 即 i 号物品可以多次放入背包, 这与 0-1 背包的条件不符. (事实上, 这正是完全背包的更新方式.)

出现上述问题的原因: 去掉第一维后, $dp[j-w_i]$ 表示的是 $dp[i][j-w_i]$ 而非 $dp[i-1][j-w_i]$.

更正方式: 调整内层循环的顺序, 使得 $dp[i][j]$ 在 $dp[i][j-w_i]$ 前更新, 即:

```
// dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
vector<int> dp(m + 5);
for (int i = 1; i <= n; i++) {
    for (int j = m; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + d[i]);
    }
}
```

初始条件 $dp[0]=0$, 最终答案 $ans=dp[m]$.

空间复杂度 $O(n+m)$, 最大为 $\dfrac{4\times(2\times3402+12880)}{1024\times1024}\ \mathrm{MB}\approx0.075\ \mathrm{MB}$.

代码 II

```
void solve() {
    int n, m;
    cin >> n >> m;

    vector<int> w(n + 5), d(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> d[i];
    }

    // dp[i][j] 表示只考虑前 i 个物品、总重量为 j 时的最大总价值
    vector<int> dp(m + 5);
    for (int i = 1; i <= n; i++) {
        for (int j = m; j >= w[i]; j--) {
            dp[j] = max(dp[j], dp[j - w[i]] + d[i]);
        }
    }

    cout << dp[m] << '\n';
}

int main() {
    solve();
```

```
}
```

思考: **代码 I** 中状态转移的过程, 能否改为:

```
    // dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
    vector<vector<int>> dp(n + 5, vector<int>(m + 5));
    for (int i = 1; i <= n; i++) {
        for (int j = m; j >= 1; j--) {  // 修改为倒序枚举
            dp[i][j] = dp[i - 1][j];

            if (j >= w[i]) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + d[i]);
            }
        }
    }
```

能.

思路 III

实践中, 还有一种用 `std::vector` 实现的滚动数组的写法.

代码 III

```
void solve() {
    int n, m;
    cin >> n >> m;

    vector<int> w(n + 5), d(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> d[i];
    }

    // dp[i][j] 表示只考虑前 i 个物品、总重量为 j 时的最大总价值
    vector<int> dp(m + 5);
    for (int i = 1; i <= n; i++) {
        // pd = dp[i][], dp = dp[i - 1][]
        auto pd = dp;

        for (int j = w[i]; j <= m; j++) {
            pd[j] = max(pd[j], dp[j - w[i]] + d[i]);
        }

        dp = pd;
    }

    cout << dp[m] << '\n';
}

int main() {
    solve();
}
```

推广

若问题改为求解总重量恰为 m 时的最大值, 则初始化 $dp[i]=\begin{cases}0, & i=0\\ -\infty, & 1\leq i\leq n\end{cases}$.

1.2 习题

[1] [蓝桥杯 2021 省 AB] 砝码称重 (https://www.luogu.com.cn/problem/P8742)

2. 完全背包

2.1 例题

2.1.1 疯狂的采药

链接: https://www.luogu.com.cn/problem/P1616

题意 ($1\ \mathrm{s}$, $128\ \mathrm{MB}$)

有 $m\ \ (1\leq m\leq 1\mathrm{e}4)$ 个物品和容量为 $t\ \ (1\leq t\leq 1\mathrm{e}7, 1\leq m\cdot t\leq 1\mathrm{e}7)$ 的背包, 其中 $i\ \ (1\leq i\leq m)$ 号物品的体积为 $a_i\ \ (1\leq a_i\leq 1\mathrm{e}4)$, 价值为 $b_i\ \ (1\leq b_i\leq 1\mathrm{e}4)$. 每个物品可多次放入背包.

求将哪些物品装入背包, 可使得物品重量之和不超过背包容量, 且价值之和最大. 输出最大总价值.

思路 I

$dp[i][j]$ 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值.

初始条件 $dp[0][0]=0$, 最终答案 $ans=dp[n][m]$.

延续 0-1 背包的思路, 易得状态转移方程:

$$dp[i][j]=\max\begin{cases}dp[i-1][j],\\ dp[i-1][j-a_i]+b_i, & j\geq a_i\\ dp[i-1][j-2\cdot a_i]+2\cdot b_i, & j\geq 2\cdot a_i\\ \cdots\end{cases}\cdots ①.$$

状态转移时, 只需将 0-1 背包的二重循环改为三重循环:

```
    // dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
    vector<vector<int>> dp(m + 5, vector<int>(t + 5));
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= t; j++) {
            for (int k = 0; k * a[i] <= j; k++) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j - k * a[i]] + k * b[i]);
            }
        }
    }
```

或根据讨论 0-1 背包中, 去掉 $dp[][]$ 数组的第一维后为何要倒序枚举体积的讨论, 直接写出完全背包正确的状态转移方式:

```
// dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
vector<int> dp(t + 5);
for (int i = 1; i <= m; i++) {
    for (int j = w[i]; j <= t; j++) {
        dp[j] = max(dp[j], dp[j - a[i]] + b[i]);
    }
}
```

但上述解释不够严谨. 下面给出严格的解释.

① 式中, 令 $j \mapsto (j - a_i)$ 得:

$$dp[i][j-a_i] = \begin{cases} dp[i-1][j-a_i], j \geq a_i \\ dp[i-1][j-2 \cdot a_i] + b_i, j \geq 2 \cdot a_i \cdots ② . \\ \cdots \end{cases}$$

注意到 ① 式与 ② 式相差一个 b_i , 则状态转移方程:

$dp[i][j] = \max\{dp[i-1][j], dp[i][j-a_i] + b_i \ \ (j \geq a_i)\}$.

最大总价值 $t \cdot \max b_i \leq 1e7 \times 1e4 = 1e11$, 需开 long long.

时间复杂度 $O(m \cdot t)$, 最大为 $m \cdot t \leq 1e7$.

空间复杂度 $O(m \cdot t)$, 最大为 $\dfrac{8 \times 1e7}{1024 \times 1024} \approx 76 \text{ MB}$.

代码 I (MLE)

```
void solve() {
    int t, m;
    cin >> t >> m;
    vector<int> a(m + 5), b(m + 5);
    for (int i = 1; i <= m; i++) {
        cin >> a[i] >> b[i];
    }

    // ll** dp = new ll*[m + 5];
    // for (int i = 0; i < m + 5; i++) {
    //     dp[i] = new ll[t + 5];

    //     for (int j = 0; j < t + 5; j++) {
    //         dp[i][j] = 0;
    //     }
    // }

    // dp[i][j] 表示只考虑前 i 个物品, 总体积不超过 t 时的最大总价值
    vector<vector<ll>> dp(m + 5, vector<ll>(t + 5));

    for (int i = 1; i <= m; i++) {
```

```
        for (int j = 1; j <= t; j++) {
            dp[i][j] = dp[i - 1][j];

            if (j >= a[i]) {
                dp[i][j] = max(dp[i][j], dp[i][j - a[i]] + b[i]);
            }
        }
    }

    cout << dp[m][t] << '\n';
}

int main() {
    solve();
}
```

虽然理论空间大小远小于空间限制, 但实际提交后 MLE, 将 `std::vector` 换为动态数组后仍 MLE.

事实上, 更准确的空间复杂度为 $O((m+1)\cdot(t+1))=O(2\cdot m\cdot t)$, 因为不滚动时还有一个 $i=0$ 数组. 故实际空间约 $2\times 76\ \mathrm{MB}=152\ \mathrm{MB}>128\ \mathrm{MB}$.

思路 II

同 0-1 背包, 可去掉 $dp[][]$ 数组的第一维.

空间复杂度 $O(m+t)$, 最大为 $\dfrac{4\times 2\times 1\mathrm{e}4+8\times 1\mathrm{e}7}{1024\times 1024}\ \mathrm{MB}\approx 76\ \mathrm{MB}$.

代码 II

```
void solve() {
    int t, m;
    cin >> t >> m;
    vector<int> a(m + 5), b(m + 5);
    for (int i = 1; i <= m; i++) {
        cin >> a[i] >> b[i];
    }

    // dp[i][j] 表示只考虑前 i 个物品, 总体积不超过 t 时的最大总价值
    vector<ll> dp(t + 5);

    for (int i = 1; i <= m; i++) {
        for (int j = a[i]; j <= t; j++) {
            dp[j] = max(dp[j], dp[j - a[i]] + b[i]);
        }
    }

    cout << dp[t] << '\n';
}

int main() {
    solve();
}
```

比较 0-1 背包与多重背包

背包	状态转移方程	压维后内层循环
0-1 背包	$dp[i][j]=\max\{dp[i-1][j],dp[i-1][j-v_i]+w_i\ \ (j\geq v_i)\}$	倒序枚举体积
多重背包	$dp[i][j]=\max\{dp[i-1][j],dp[i][j-v_i]+w_i\ \ (j\geq v_i)\}$	顺序枚举体积

2.2 习题

[1] [蓝桥杯 2017 省 AB] 包子凑数 (https://www.luogu.com.cn/problem/P8646)

---

3. 多重背包

3.1 例题

3.1.1 宝物筛选

链接: https://www.luogu.com.cn/problem/P1776

题意 ($1\ \mathrm{s}$, $125\ \mathrm{MB}$)

有 $n\ \ (1\leq n\leq 100)$ 个物品和一个容量为 $W\ \ (0\leq W\leq 4\mathrm{e}4)$ 的背包, 其中 $i\ \ (1\leq i\leq n)$ 号物品有 $m_i\ \ (n\leq\sum m_i\leq 1\mathrm{e}5)$ 件, 每件的重量为 w_i , 价值为 v_i .

求将哪些物品装入背包, 可使得物品重量之和不超过背包容量, 且价值之和最大. 输出最大总价值.

答案保证在 `int` 范围内.

思路 I

将 m_i 件 $i\ \ (1\leq i\leq n)$ 号物品视为 m_i 个不同的物品, 转化为 0-1 背包.

时间复杂度 $O\left(n\cdot\sum m_i\cdot W\right)$, 最大为 $100\times 1\mathrm{e}5\times 4\mathrm{e}4=4\mathrm{e}11$, 会 TLE.

类似于快速幂的优化思路, 可将 m_i 件 i 号物品按二进制分组, 即**二进制优化**. 具体地, 考察 m_i 的二进制表示, 将每组大小取为二进制表示中为 1 的位对应的权重.

时间复杂度 $O\left(n\cdot W\cdot\left\lceil\log_2\sum m_i\right\rceil\right)=O\left(n\cdot W\cdot\log\sum m_i\right)$, 最大为 $100\times 4\mathrm{e}4\times\log_2 1\mathrm{e}5\approx 6.6\mathrm{e}7$.

用滚动数组, 空间复杂度 $O(W)$.

代码 I

```
void solve() {
    int n, W;
    cin >> n >> W;

    // 二进制分组后的物品的价值、质量
    vector<int> v, w;
```

```
    while (n--) {
        int tmp_v, tmp_w, tmp_m;
        cin >> tmp_v >> tmp_w >> tmp_m;

        // 完整的 2 的幂次个物品作为一组
        for (int k = 1; k <= tmp_m; k *= 2) {
            tmp_m -= k;

            v.push_back(k * tmp_v), w.push_back(k * tmp_w);
        }

        // 不完整的 2 的幂次个物品作为一组
        if (tmp_m > 0) {
            v.push_back(tmp_m * tmp_v), w.push_back(tmp_m * tmp_w);
        }
    }

    // dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
    vector<int> dp(W + 5);

    n = v.size();
    for (int i = 0; i < n; i++) {
        for (int j = W; j >= w[i]; j--) {
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }

    cout << dp[W] << '\n';
}

int main() {
    solve();
}
```

思路 II

类似于完全背包的推导过程:

$$dp[i][j] = \max \begin{cases} dp[i-1][j] \\ dp[i-1][j-w_i] + v_i, \ \ j \geq w_i \\ \cdots \\ dp[i-1][j-m_i \cdot w_i] + m_i \cdot v_i, \ \ j \geq m_i \cdot v_i \end{cases} \qquad \cdots ①,$$

$$dp[i][j-v_i] = \max \begin{cases} dp[i-1][j-w_i] \\ dp[i-1][j-2 \cdot w_i] + v_i, \ \ j \geq 2 \cdot w_i \\ \cdots \\ dp[i-1][j-(m_i+1) \cdot w_i] + m_i \cdot v_i, \ \ j \geq (m_i+1) \cdot v_i \end{cases} \qquad \cdots ②.$$

去掉 ① 的第一项和 ② 的最后一项后, $dp[i][j]$ 的每一项比 $dp[i][j-v_i]$ 的每一项多一个 v_i .

将 j 按模 w_i 的余数分组, 令 $r = j \bmod w_i$, 则每个 j 可表示为 $k \cdot w_i + r$, 其中合法的 k s.t. $k \cdot w_i + r \leq W$.

每个 $dp[i][j]$ 等于其在 j 所在分组的位置, 往前数 m_i 个 $dp[i][j']$ 的最大值加若干个 v_i .

对 i 号物品的转移, 用单调队列维护长度为 m_i 的滑动窗口中的最大值.

时间复杂度 $O(n \cdot W)$.

代码 II

```cpp
void solve() {
    int n, W;
    cin >> n >> W;

    vector<int> v(n + 5), w(n + 5), m(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i] >> m[i];
    }

    // dp[i][j] 表示只考虑前 i 个物品、总重量不超过 j 时的最大总价值
    vector<int> dp(W + 5);

    for (int i = 1; i <= n; i++) {
        auto pd = dp;

        for (int r = 0; r < w[i]; r++) {  // 枚举 (j % w[i])
            // 单调队列维护当前合法的 k
            deque<int> que;

            for (int k = 0; k * w[i] + r <= W; k++) {
                // 弹出超出范围的元素
                while (que.size() && k - que.front() > m[i]) {
                    que.pop_front();
                }

                // 维护单调性
                while (
                    que.size() &&
                    dp[que.back() * w[i] + r] - que.back() * v[i] < dp[k * w[i] + r] - k * v[i]
                ) {
                    que.pop_back();
                }

                que.push_back(k);

                pd[k * w[i] + r] = dp[que.front() * w[i] + r] - que.front() * v[i] + k * v[i];
            }
        }

        dp = pd;
    }

    cout << dp[W] << '\n';
}

int main() {
    solve();
}
```

3.2 习题

[1] 科技庄园 (https://www.luogu.com.cn/problem/solution/P2760)
