线性DP基础 2024SZUACM算法周训 (241207)

回放: https://meeting.tencent.com/crm/NbZPwA88b9

题单: https://vjudge.net/contest/678251

---

0. 动态规划

动态规划 (Dynamic Programming, DP) 是一种将原问题分解为简单的子问题来求解的思想.

DP 的两个关键问题: 状态设计、状态转移.

DP 的状态设计关系到转移的复杂程度. DP 的状态设计除了经典的套路外, 也很依赖于经验.

DP 的两种转移形式:

(1) 递推: 常见.

(2) 递归: 也称**记忆化搜索**.

---

1. 线性 DP

线性 DP 不是某一种 DP , 而是一类 DP 的统称.

线性 DP 指转移的形式是线性的, 不是指复杂度是线性的.

1.1 一维

1.1.1 最大子段和

链接: https://www.luogu.com.cn/problem/P1115

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

给定一个长度为 $n\ \ (1\leq n\leq 2\mathrm{e}5)$ 的整数序列 $a=[a_1,\cdots,a_n]\ \ (-1\mathrm{e}4\leq a_i\leq 1\mathrm{e}4)$. 求 $a[]$ 中一段连续非空的子段, s.t. 该段中元素之和最大, 输出该最大值.

注:

(1) 子段、子区间、子串: 连续.

(2) 子列: 未必连续.

思路 I

dp_i 表示只考虑前 i 个元素时, a_i 所在的子段的最大和.

初始条件 $dp_0=0$, 最终答案 $ans=\max\limits_{1\leq i\leq n} dp_i$.

按 a_i 是否接到前面一段分类:

(1) 若不接, 则 $dp_i=a_i$.

(2) 若接, 则 $dp_i=dp_{i-1}+a_i$.

状态转移方程 $dp_i=\max\{a_i,dp_{i-1}+a_i\}$.

时间复杂度 $O(n)$, 空间复杂度 $O(n)$.

代码 I

```cpp
const int INF = 2e9 + 5;

void solve() {
    int n;
    cin >> n;
    vector<int> a(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    // dp[i] 表示只考虑前 i 个元素时, a[i] 所在的子段的最大和
    vector<int> dp(n + 5);
    dp[0] = 0;  // 初始条件

    for (int i = 1; i <= n; i++) {
        dp[i] = max(dp[i - 1] + a[i], a[i]);
    }

    int ans = -INF;
    for (int i = 1; i <= n; i++) {
        ans = max(ans, dp[i]);
    }

    cout << ans << '\n';
}

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

思考:

(1) 最终答案为何不是 dp_n . 不能, 但序列非负时能.

(2) ans 能否初始化为 0 . 不能.

(3) INF 能否为 $0x3f3f3f3f$. 能.

思路 II

注意到状态转移方程 $dp_i = \max\{a_i, dp_{i-1} + a_i\}$ 中, dp_i 只与 dp_{i-1} 有关, 故可滚动.

时间复杂度 $O(n)$, 空间复杂度 $O(1)$.

代码 II

```
const int INF = 2e9 + 5;

void solve() {
    int n;
    cin >> n;
    vector<int> a(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    int ans = -INF;

    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum = max(sum + a[i], a[i]);

        ans = max(ans, sum);
    }

    cout << ans << '\n';
}

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

1.2 二维

1.2.1 [蓝桥杯 2020 省 AB1] 走方格

链接: https://www.luogu.com.cn/problem/P8707

题意 (1 s , 128 MB)

有一个 $n\ \ (1 \leq n \leq 30)$ 行 $m\ \ (1 \leq m \leq 30)$ 列的网格. 某人从左上角 $(1, 1)$ 出发, 前往右下角 (n, m) . 若要求每一步只能向下走或向右走, 且不能走入行数和列数都为偶数的格, 求他行进的方案数.

思路

$dp_{i,j}$ 表示从点 $(1,1)$ 走到点 (i,j) 的方案数.

初始条件 $dp_{1,1}=1$, 最终答案 $ans=dp_{n,m}$.

按每一步从上来或从左来分类, 状态转移方程 $dp_{i,j}=dp_{i-1,j}+dp_{i,j-1}\ \ (i,j\text{ 不同时为偶数})$.

时间复杂度 $O(n\cdot m)$, 空间复杂度 $O(n\cdot m)$.

代码

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

    // dp[i][j] 表示走到点 (i, j) 的方案数
    vector<vector<int>> dp(n + 5, vector<int>(m + 5));
    dp[1][1] = 1;  // 初始条件

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (i == 1 && j == 1) continue;

            if ((i & 1) || (j & 1)) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
    }
    cout << dp[n][m] << '\n';
}

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

1.3 三维

1.3.1 [蓝桥杯 2022 省 B] 李白打酒加强版

链接: https://www.luogu.com.cn/problem/P8786

题意 (1 s , 128 MB)

某人带着 2 单位的酒出发, 一路上遇到店 $n\ \ (1\leq n\leq 100)$ 次, 遇到花 $m\ \ (1\leq m\leq 100)$ 次, 最后一次遇到的是花, 且正好将酒喝光. 已知他遇到店时, 拥有的酒量会翻一倍; 遇到花时, 拥有的酒量会减少 1 单位. 求他路上遇到店和花的顺序的方案数, 答案对 $(1\text{e}9+7)$ 取模.

规定: 没有酒时遇到店是合法的, 但遇到花是不合法的.

思路

$dp_{i,j,k}$ 表示走 i 步、经过 j 个店、当前酒量为 k 的方案数.

初始条件 $dp_{1,0,2}=1$, 即走 1 步、经过 0 个店, 即第一步遇到花时, 酒量为 1 .

最终答案 $ans=dp_{n+m,n,1}$, 即最后一步遇到花, 喝之前酒量剩 1 .

枚举步数 i 、经过的店数 j , 按上一步遇到的是花还是店分类:

(1) 遇到花, 则 $dp_{i,j,k}+=dp_{i-1,j,k+1}$.

(2) 遇到店, 则 $dp_{i,j,k}+=dp_{i-1,j-1,k/2}\ \ (j\geq 1, k\text{ 是偶数})$.

时间复杂度 $O((n+m)\cdot n\cdot m)=O(n^2\cdot m^2)$, 空间复杂度 $O(n^2\cdot m^2)$.

代码

```cpp
const int MAXN = 105;
const int MOD = 1e9 + 7;

// dp[i][j][k] 表示走 i 步、遇到 j 个店、当前酒量为 k 的方案数
int dp[MAXN * 2][MAXN][MAXN];  // 注意第一维开两倍

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

    dp[1][0][2] = 1;  // 初始条件

    for (int i = 2; i <= n + m; i++) {
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= m; k++) {
                // 遇到花
                dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j][k + 1]) % MOD;

                // 遇到店
                if (j && (k % 2 == 0)) {
                    dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - 1][k / 2]) % MOD;
                }
            }
        }
    }

    cout << dp[n + m][n][1] << '\n';
}

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

1.4 习题

[1] Particles (https://codeforces.com/contest/1844/problem/C)

[2] [NOIP2004 普及组] 花生采摘 (https://www.luogu.com.cn/problem/P1086)

[3] [蓝桥杯 2022 省 B] 积木画 (https://www.luogu.com.cn/problem/P8784)

[4] Mio visits ACGN Exhibition (https://codeforces.com/gym/103366/problem/A)

---

2. LIS & LCS

2.1 最长上升子列 (LIS)

2.1.1 最长上升子序列

链接: https://www.luogu.com.cn/problem/B3637

题意 ($1\ \mathrm{s}$, $512\ \mathrm{MB}$)

给定一个长度为 $n\ \ (1\leq n\leq 5000)$ 的整数序列 $a=[a_1,\cdots,a_n]\ \ (1\leq a_i\leq 1\mathrm{e}6)$. 从 $a[]$ 中取出一个子列, s.t. 其中的元素严格递增, 输出这样的子列的最大长度.

思路 I

dp_i 表示以 a_i 结尾的 LIS 的长度.

初始条件 $dp_i=1\ \ (1\leq i\leq n)$, 最终答案 $ans=\max\limits_{1\leq i\leq n} dp_i$.

按以 a_i 结尾的 LIS 的倒数第二个元素分类, 状态转移方程 $dp_i=\max\limits_{\substack{1\leq j<i\\ a_j<a_i}}\{dp_j+1\}$.

状态数 $O(n)$, 转移 $O(n)$, 时间复杂度 $O(n^2)$.

空间复杂度 $O(n)$.

代码 I

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

    // dp[i] 表示以 a[i] 结尾的 LIS 的长度
    vector<int> dp(n + 5);
    for (int i = 1; i <= n; i++) {
        dp[i] = 1;  // 初始条件

```

```
        for (int j = 1; j < i; j++) {
            if (a[j] < a[i]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

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

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

思路 II

考察序列 $a = [3, 1, 2, 1, 8, 5, 6]$.

注意到若元素 a_i 可接到元素 3 之后形成上升子列, 则也能接到元素 1 之后形成上升子列.

用数组 $lst[]$ 记录当前 LIS 的结尾数, 其中 lis_i 表示长度为 i 的 LIS 的结尾数, 初始时 $lst_0 = -INF < \min\limits_{1 \leq i \leq n} a_i$.

为 s.t. 上升子列尽可能长, 求以 a_i 结尾的 LIS 时, 应尽量将 a_i 接到之前的 $< a_i$ 的最大数之后.

注意到 LIS 的长度的增大, 其结尾数的最小值单调增, 故求 $< a_i$ 的最大数可二分.

状态数 $O(n)$, 转移 $O(\log n)$, 时间复杂度 $O(n \log n)$.

空间复杂度 $O(n)$.

代码 II

```
const int INF = 0x3f3f3f3f;

void solve() {
    int n;
    cin >> n;
    vector<int> a(n + 5);
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }

    // dp[i] 表示以 a[i] 结尾的 LIS 的长度
    vector<int> dp(n + 5);

    // lst[i] 表示长度为 i 的 LIS 的结尾数的最小值
    vector<int> lst(n + 5);
    lst[0] = -INF;  // < min(a[])

    int max_length = 0;
    for (int i = 1; i <= n; i++) {
        // 当前 lst[] 中 < a[i] 的最大值
        int pos = lower_bound(
            lst.begin(), lst.begin() + max_length + 1,
            a[i]
```

```
        ) - lst.begin() - 1;

        // 将 a[i] 接到该数后
        dp[i] = pos + 1;
        lst[pos + 1] = a[i];

        // 更新 lst[] 的长度
        max_length = max(max_length, dp[i]);
    }

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

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

若只需要整个序列的 LIS, 则无需 $dp[]$ 数组, 最后输出 max_length 即可.

2.2 最长公共子列 (LCS)

2.2.1 最长公共子序列

链接: https://leetcode.cn/problems/longest-common-subsequence/description/

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

给定两个字符串 $a, b\ \ (|a|, |b| \leq 1000)$. 分别从序列 $a[]$ 和 $b[]$ 中取出一个子列, s.t. 它们相等, 输出这样的子列的最大长度.

思路

设 $a[]$ 、$b[]$ 的长度分别为 n, m , 且下标都从 1 开始.

$dp_{i,j}$ 表示前缀 $a[1 \cdots i]$ 与前缀 $b[1 \cdots j]$ 的 LCS 的长度.

初始条件 $dp_{0,0} = 0$, 最终答案 $ans = dp_{n,m}$.

按 a_i 和 b_j 分别是否在 LCS 中分类:

(1) a_i 不在, b_j 不在.

此时的 LCS 即前缀 $a[1 \cdots (i-1)]$ 与前缀 $b[1 \cdots (j-1)]$ 的 LCS ,

其长度为 $dp_{i-1,j-1}$.

(2) a_i 在, b_j 在, 则要求 $a_i = b_j$.

此时的 LCS 即前缀 $a[1 \cdots (i-1)]$ 与前缀 $b[1 \cdots (j-1)]$ 的 LCS 再接上 a_i 或 b_j ,

其长度为 $dp_{i-1,j-1} + 1$.

(3) a_i 不在, b_j 在.

注意到该情况的 LCS 的长度并非 $dp_{i-1,j}$, 因为 $dp_{i-1,j}$ 代表的 LCS 未必包含 b_j .

但注意到求最大值时不可漏但可重, 而 a_i 不在, b_j 在的情况可被 $dp_{i-1,j}$ 覆盖,

故取该情况的 LCS 的长度为 $dp_{i-1,j}$.

(4) a_i 在, b_j 不在.

类似于 (3), 取该情况的 LCS 的长度为 $dp_{i,j-1}$.

状态转移方程 $dp_{i,j}=\max\begin{cases}dp_{i-1,j-1}\\dp_{i-1,j}\\dp_{i,j-1}\\dp_{i-1,j-1}+1\ \ (a_i=b_j)\end{cases}$.

状态数 $O(n\cdot m)$, 转移 $O(1)$, 时间复杂度 $O(n\cdot m)$.

空间复杂度 $O(n\cdot m)$.

代码

本地测试:

```
void solve() {
    string a, b;
    cin >> a >> b;

    int n = a.length(), m = b.length();
    a = " " + a, b = " " + b;  // 下标从 1 开始

    // dp[i][j] 表示 a[1 ... i] 与 b[1 ... j] 的 LCS 的长度
    vector<vector<int>> dp(n + 5, vector<int>(m + 5));  // 初始条件

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            // (a[i] 在, b[j] 不在) 或 (a[i] 不在, b[j] 在)
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

            if (a[i] == b[j]) {
                // a[i] 在, b[j] 在
                dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1);
            }
        }
    }
    cout << dp[n][m] << '\n';
}

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

Lc 提交:

```
class Solution {
public:
    int longestCommonSubsequence(string a, string b) {
```

```
        int n = a.length(), m = b.length();
        a = " " + a, b = " " + b;  // 下标从 1 开始

        // dp[i][j] 表示 a[1 ... i] 与 b[1 ... j] 的 LCS 的长度
        vector<vector<int>> dp(n + 5, vector<int>(m + 5));  // 初始条件

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                // (a[i] 在, b[j] 不在) 或 (a[i] 不在, b[j] 在)
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

                if (a[i] == b[j]) {
                    // a[i] 在, b[j] 在
                    dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1);
                }
            }
        }
        return dp[n][m];
    }
};
```

2.3 习题

[1] [NOIP2004 提高组] 合唱队形 (https://www.luogu.com.cn/problem/P1091)

[2] [NOIP1999 提高组] 导弹拦截 (https://www.luogu.com.cn/problem/P1020)