DP 算法

特征

一个问题能否用 DP 解决，取决于它是否具备以下特征：

- 重叠子问题：在求解过程中，相同的子问题会被多次计算。DP 通过将结果存储在数组中，使每个子问题只计算一次 。
- 最优子结构：大问题的最优解包含着小问题的最优解。
- 无后效性：一旦某个阶段的状态确定，它之后的过程不会影响在此之前的状态。

解题的四大步骤

在 C 组竞赛中，解决普通一维 DP 问题通常遵循以下逻辑：

- 确定 dp 数组及下标的含义：明确 `dp[i]` 代表什么。
  - 例子（爬楼梯）：`dp[i]` 表示爬到第 i 层楼梯共有多少种走法。
- 确定状态转移方程：这是 DP 的灵魂，即如何由已知的 `dp[j]` （$j < i$）推导出 `dp[i]`。
  - 例子：要到第 i 层，可以从第 $i-1$ 层跨 1 步，也可以从第 $i-2$ 层跨 2 步。所以 $dp[i] = dp[i-1] + dp[i-2]$。
- 初始化边界条件：确定最简单情况下的解，防止数组越界或计算逻辑中断。
  - 例子：$dp[1] = 1$（1层只有1种走法），$dp[2] = 2$（2层有 1+1 或 2 两种走法）。
- 确定遍历顺序：通常是从小到大遍历，确保计算 `dp[i]` 时，它所依赖的 `dp[i-1]` 等子问题已经计算完毕。

DP 算法标准代码模板

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    int n;  // 输入的目标数字
    if (!(cin >> n)) return 0;  // 读取输入，失败则直接退出

    // 创建动态规划数组，大小为n+1（包含0到n），初始化为0
    // 使用long long类型防止大数溢出
    vector<long long> dp(n + 1, 0);

    dp[0] = 0;  // 基础情况：第0个值
    if (n >= 1) dp[1] = 1;  // 基础情况：第1个值

    // 动态规划循环：从第2个元素开始计算，直到第n个
    for (int i = 2; i <= n; i++) {
        // 这里需要根据具体动态规划问题填写状态转移方程
        // 例如斐波那契数列：dp[i] = dp[i-1] + dp[i-2]
        // 具体状态转移方程取决于要解决的DP问题
    }

    // 输出第n个计算结果的动态规划值
    cout << dp[n] << endl;

    return 0;  // 程序正常结束
```

```
}
```

爬楼梯

问题描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

DP 解题思路

状态定义：`dp[i]` 表示到达第 i 阶台阶的方案总数。

转移方程：到达第 i 阶只有两种可能：

1. 从第 $i-1$ 阶跨 1 步上来的。
2. 从第 $i-2$ 阶跨 2 步上来的。

- 因此：$dp[i] = dp[i-1] + dp[i-2]$。

边界条件：

- `dp[1] = 1`（1阶只有1种走法）
- `dp[2] = 2`（2阶有 1+1 或 2 两种走法）

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n;  // 输入变量，表示台阶数或问题规模
    if (!(cin >> n)) return 0;  // 读取输入，失败则直接退出

    // 处理特殊情况：当n小于等于2时，直接输出n
    // 在爬楼梯问题中，1个台阶有1种方法，2个台阶有2种方法
    if (n <= 2) {
        cout << n << endl;  // 输出结果
        return 0;           // 提前结束程序
    }

    long long dp[101];  // 创建动态规划数组，大小固定为101（假设n不超过100）
                        // 使用long long类型防止大数溢出

    dp[1] = 1;  // 基础情况：当只有1个台阶时，只有1种爬法（爬1步）
    dp[2] = 2;  // 基础情况：当有2个台阶时，有2种爬法（一次爬2步，或分两次各爬1步）

    // 动态规划循环：从第3个台阶开始计算，直到第n个台阶
    // 状态转移方程：dp[i] = dp[i-1] + dp[i-2]
    // 表示到达第i个台阶的方法数等于到达第i-1个台阶的方法数加上到达第i-2个台阶的方法数
    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];  // 状态转移
    }

    // 输出到达第n个台阶的方法数
    cout << dp[n] << endl;

    return 0;  // 程序正常结束
}
```

最大子数组和

问题描述

在一个给定的整数数组（可能包含正数、负数和零）中，寻找一个具有最大和的连续子数组（子数组最少包含一个元素），并返回其最大和。

- 样例输入：[-2, 1, -3, 4, -1, 2, 1, -5, 4]
- 样例输出：6 （连续子数组 [4, -1, 2, 1] 的和最大）

DP 解题思路

解决此问题的核心在于：对于当前的数字，是加入前面的子数组划算，还是自己独立开一个子数组划算？

1. 第一步：定义状态 (dp 数组含义)

- 定义：$dp[i]$ 表示以第 i 个元素为结尾的连续子数组的最大和。
- 注意：必须以第 i 个元素结尾，这样才能保证子数组的“连续性”，从而让 $dp[i]$ 和 $dp[i-1]$ 产生联系。

2. 第二步：状态转移方程

对于数组中的第 i 个数字 $a[i]$，它的最大和取决于前一个状态 $dp[i-1]$：

- 加入前面的队伍：如果 $dp[i-1] > 0$，那么 $a[i]$ 加上它肯定会变大，此时 $dp[i] = dp[i-1] + a[i]$。
- 另起炉灶：如果 $dp[i-1] \leq 0$，那么加上它只会让 $a[i]$ 变小或不变，此时 $dp[i] = a[i]$。
- 通用公式：$dp[i] = \max(a[i], dp[i-1] + a[i])$

3. 第三步：初始化与边界

- 起始点：$dp[0] = a[0]$，因为第一个元素前面没有数字。
- 初始最大值：设置一个全局变量 max_sum，初始值设为 $dp[0]$。

4. 第四步：遍历顺序与结果

- 顺序：从左到右遍历一次数组即可（时间复杂度 $O(n)$）。
- 最终解：在遍历过程中，不断用 max_sum = max(max_sum, dp[i]) 更新全局最大值。

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    int n;  // 输入数组的长度
    if (!(cin >> n)) return 0;  // 读取数组长度，输入失败则退出程序

    vector<long long> a(n);  // 存储原始数组元素，使用long long防止大数溢出
    vector<long long> dp(n);  // 动态规划数组，dp[i]表示以a[i]结尾的最大子数组和

    // 读取n个整数到数组a中
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    dp[0] = a[0];  // 初始化：以第一个元素结尾的最大子数组和就是它本身
    long long max_sum = dp[0];  // 初始化全局最大和为dp[0]

    // 动态规划循环：计算每个位置结束的最大子数组和
    for (int i = 1; i < n; i++) {
        // 状态转移方程：以a[i]结尾的最大子数组和有两种选择
```

```
        // 1. 只包含当前元素a[i]
        // 2. 将当前元素加到以a[i-1]结尾的最大子数组和上
        dp[i] = max(a[i], dp[i-1] + a[i]);

        // 更新全局最大和
        if (dp[i] > max_sum) {
            max_sum = dp[i];  // 如果当前dp[i]比全局最大和大，则更新
        }
    }

    cout << max_sum << endl;  // 输出整个数组的最大子数组和

    return 0;  // 程序正常结束
}
```

打家劫舍

问题描述

假设你是一名职业小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统：如

果两间相邻的房屋在同一晚上被小偷闯入，系统就会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不惊动警报装置的情况下，一夜之内能够偷窃到的最高金额。

样例输入：[2, 7, 9, 3, 1]

样例输出：12（偷取 1 号、3 号和 5 号房屋：2 + 9 + 1 = 12）

DP 解题思路

1. 第一步：定义状态 dp[i]

- 含义：dp[i] 表示前 i 间房屋能偷窃到的最高总金额。

2. 第二步：推导转移方程

当你面对第 i 间房屋时，你有两个选择：

- 偷这一间：如果你偷第 i 间，那么你就绝对不能偷第 $i-1$ 间。所以你的总收益是：前 $i-2$ 间房的最大收益 + 第 i 间房的金额。
  - 即：dp[i-2] + nums[i]
- 不偷这一间：如果你放弃第 i 间，那么你的总收益就是：前 $i-1$ 间房的最大收益。
  - 即：dp[i-1]
- 通用公式：
  - $dp[i] = \max(dp[i-1], dp[i-2] + nums[i])$

3. 第三步：初始化边界条件

- dp[0]：只有一间房，最大金额就是这间房的钱 nums[0]。
- dp[1]：有两间房，因为不能同时偷，最大金额是 max(nums[0], nums[1])。

4. 第四步：遍历顺序

- 从左到右，从下标 2 开始遍历到 $n-1$ 。

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;
```

```cpp
int main()
{
    int n;  // 输入数组的长度
    if (!(cin >> n)) return 0;  // 读取数组长度，输入失败则退出程序

    // 处理特殊情况：如果数组长度为0，最大和为0
    if (n == 0) {
        cout << 0 << endl;  // 输出0
        return 0;           // 提前结束程序
    }

    vector<long long> nums(n);  // 存储原始数组元素，使用long long防止大数溢出
    // 读取n个整数到数组nums中
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    // 处理特殊情况：如果数组只有1个元素，最大和就是该元素本身
    if (n == 1) {
        cout << nums[0] << endl;  // 输出第一个元素
        return 0;                 // 提前结束程序
    }

    vector<long long> dp(n);  // 动态规划数组，dp[i]表示考虑前i+1个元素时的最大和

    dp[0] = nums[0];  // 基础情况：只有第一个元素时，最大和就是它本身
    dp[1] = max(nums[0], nums[1]);  // 基础情况：两个元素时，取较大值（不能同时取相邻元素）

    // 动态规划循环：从第3个元素开始计算（索引2）
    for (int i = 2; i < n; i++) {
        // 状态转移方程：
        // 1. 不选择当前元素，那么最大和等于前i-1个元素的最大和（dp[i-1]）
        // 2. 选择当前元素，那么不能选择前一个元素，最大和等于前i-2个元素的最大和加上当前元素（dp[i-2] + nums[i]）
        dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);
    }

    // 输出考虑所有n个元素时的最大和
    cout << dp[n - 1] << endl;

    return 0;  // 程序正常结束
}
```

最小路径和

问题描述

给定一个包含非负整数的 $m \times n$ 网格 `grid` ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

- 限制条件：每次只能向下或者向右移动一步。
- 示例：输入：

```
1 3 1
1 5 1
4 2 1
```

- 输出：7（路径：1→3→1→1→1）

DP 解题思路

1. 第一步：定义状态 `dp[i][j]`

- 含义：`dp[i][j]` 表示从坐标 $(0,0)$ 走到坐标 (i,j) 的最小路径和。

2. 第二步：推导转移方程

由于你只能从“上方”或“左方”走到当前格子 (i,j)，那么：

- 从上方走下来的总和：`dp[i-1][j] + grid[i][j]`
- 从左方走过来的总和：`dp[i][j-1] + grid[i][j]`

转移方程：

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$$

3. 第三步：初始化边界

- 起始点：`dp[0][0] = grid[0][0]`。
- 最左列：只能从上方下来，`dp[i][0] = dp[i-1][0] + grid[i][0]`。
- 最上行：只能从左方过来，`dp[0][j] = dp[0][j-1] + grid[0][j]`。

最长递增子序列

问题描述

给定一个长度为 n 的序列，从中找出最长的子序列，使得子序列中的元素是严格单调递增的。

- 注意：子序列不要求在原序列中连续，但相对顺序不能改变。
- 样例输入：[10, 9, 2, 5, 3, 7, 101, 18]
- 样例输出：4 （解释：子序列为 [2, 3, 7, 18] 或 [2, 5, 7, 101] 等）

DP 解题思路

1. 第一步：定义状态 dp[i]

- 含义：dp[i] 表示以原序列第 i 个数字为结尾的最长递增子序列的长度。

2. 第二步：推导转移方程

为了算出 dp[i]，我们需要观察它前面的所有数字（下标从 0 到 $i-1$）：

- 遍历前面的每一个数字 j。
- 如果当前数字 $a[i] > a[j]$，说明 $a[i]$ 可以接到以 $a[j]$ 结尾的序列后面。
- 此时，新的长度就是 dp[j] + 1。
- 通用公式：

$$dp[i] = \max(1, dp[j] + 1) \text{ 其中 } 0 \le j < i \text{ 且 } a[i] > a[j]$$

3. 第三步：初始化边界

- 每个单独的数字本身都是一个长度为 1 的递增子序列。
- 初始化 dp 数组所有元素为 1。

4. 第四步：计算结果

- 最终答案不是 dp[n-1]，而是整个 dp 数组中的最大值。

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n;  // 输入数组的长度
    if (!(cin >> n)) return 0;  // 读取数组长度，输入失败则退出程序

    vector<int> a(n);  // 存储原始数组元素
    // 读取n个整数到数组a中
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // 动态规划数组，dp[i]表示以a[i]结尾的最长递增子序列的长度
    // 初始化为1，因为每个元素本身至少构成一个长度为1的递增子序列
    vector<int> dp(n, 1);

    int ans = 1;  // 记录全局最长递增子序列的长度，初始化为1

    // 动态规划循环：计算以每个位置i结尾的最长递增子序列长度
    for (int i = 1; i < n; i++) {
        // 内层循环：检查a[i]之前的所有元素a[j]
        for (int j = 0; j < i; j++) {
            // 如果a[i]大于a[j]，说明a[i]可以接在a[j]后面形成更长的递增子序列
            if (a[i] > a[j]) {
                // 更新dp[i]：取当前dp[i]和dp[j]+1中的较大值
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        // 更新全局最长递增子序列的长度
        ans = max(ans, dp[i]);
    }

    // 输出整个数组的最长递增子序列的长度
    cout << ans << endl;

    return 0;  // 程序正常结束
}
```

0/1 背包问题

问题描述

给你一个载重量为 W 的背包，以及 n 个物品。每个物品都有两个属性：重量 w 和 价值 v。

- 核心规则：每个物品只有一件，你只能选择放（1）或者不放（0）。
- 目标：在不超过背包总重量的前提下，使背包内物品的总价值最大。

DP 解题思路

1. 第一步：定义状态 `dp[i][j]`

- 含义：前 i 个物品，在背包容量为 j 的情况下，能拿到的最大价值。

2. 第二步：推导转移方程

面对第 i 个物品，你有两个决策：

- 不放第 i 个物品：最大价值继承自前 i-1 个物品在容量 j 下的结果。

    - 公式：`dp[i-1][j]`
- 放第 i 个物品（前提是 j >= w[i]）：腾出第 i 个物品的重量，再加上它的价值。
    - 公式：`dp[i-1][j - w[i]] + v[i]`

通用公式：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$$

3. 第三步：初始化边界

- `dp[0][j]`（不选任何物品）和 `dp[i][0]`（背包容量为 0）的值均为 0。

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n, W;  // n：物品数量，W：背包容量
    if (!(cin >> n >> W)) return 0;  // 读取输入，失败则退出程序

    // 定义重量数组和价值数组，大小为n+1（方便从1开始索引）
    vector<int> w(n + 1), v(n + 1);
    // 读取每个物品的重量和价值
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> v[i];
    }

    // 动态规划数组，dp[j]表示容量为j的背包能获得的最大价值
    vector<long long> dp(W + 1, 0);

    // 0-1背包问题动态规划核心部分
    for (int i = 1; i <= n; i++) {           // 遍历每个物品
        for (int j = W; j >= w[i]; j--) {    // 从大到小遍历背包容量
            // 状态转移方程：
            // 不选当前物品：价值保持dp[j]不变
            // 选择当前物品：价值为dp[j-w[i]] + v[i]
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }

    // 输出背包容量为W时能获得的最大价值
    cout << dp[W] << endl;

    return 0;  // 程序正常结束
}
```

数字三角形

问题描述

给定一个共有 n 行的数字三角形，从第一行（顶点）出发，每次只能移动到下一行相邻的两个数字（即正下方或右下方），求一条从顶部到底部的路径，使得路径上数字之和最大。

示例输入（$n = 5$）：

```
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

输出：`30`（路径：7→3→8→7→5）

DP 解题思路

1. 第一步：定义状态 `dp[i][j]`

- 含义：`dp[i][j]` 表示从顶点 $(1,1)$ 出发，到达第 i 行第 j 列位置时的最大路径和。

2. 第二步：推导转移方程

由于每个点只能从“正上方”或“左上方”走下来：

- 从正上方下来的位置：`dp[i-1][j]`
- 从左上方下来的位置：`dp[i-1][j-1]`
- 通用公式：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-1]) + a[i][j]$$

3. 第三步：初始化边界

- 起点：`dp[1][1] = a[1][1]`。
- 侧边处理：
  - 第一列（最左边）只能由上一行第一列走来。
  - 最后一列（最右边）只能由上一行最后一列走来。
- 技巧：在数组周围多开一圈并初始化为负无穷或 0（如果全为正数），可以避免判断边界。

4. 第四步：计算结果

- 最终答案是最后一行所有 `dp[n][j]` 中的最大值。

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

// 定义两个二维数组，a存储原始三角形数据，dp用于动态规划
long long a[1010][1010];  // 存储数字三角形的原始数据，最大规模1010×1010
long long dp[1010][1010];  // 动态规划数组，dp[i][j]表示到达第i行第j列的最大路径和

int main() {
    int n;  // 数字三角形的行数
    if (!(cin >> n)) return 0;  // 读取行数，输入失败则退出程序

    // 读取数字三角形的数据
    // 第i行有i个数字
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
            cin >> a[i][j];  // 读取第i行第j列的数字
        }
    }

    // 初始化：从三角形顶部开始，dp[1][1]就是第一行第一个数字
    dp[1][1] = a[1][1];

    // 动态规划：从第二行开始计算到达每个位置的最大路径和
    for (int i = 2; i <= n; i++) {
```

```
        for (int j = 1; j <= i; j++) {
            // 处理三种情况：
            if (j == 1) {  // 当前行第一个位置，只能从上一行的第一个位置下来
                dp[i][j] = dp[i-1][j] + a[i][j];
            } else if (j == i) {  // 当前行最后一个位置，只能从上一行的最后一个位置下来
                dp[i][j] = dp[i-1][j-1] + a[i][j];
            } else {  // 中间位置，可以从上一行的左边或右边下来，取较大值
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-1]) + a[i][j];
            }
        }
    }

    // 在最后一行的所有位置中，找出最大的路径和
    long long ans = -1e18;  // 初始化为一个非常小的数，确保任何有效值都比它大
    for (int j = 1; j <= n; j++) {
        ans = max(ans, dp[n][j]);  // 更新最大值
    }

    // 输出从顶部到底部的最大路径和
    cout << ans << endl;

    return 0;  // 程序正常结束
}
```