「白茄子」调试错误总结报告

HDU 2026 杭电春季联赛 3 · 1008

2026-04-09

本文档记录了在使用「DFA 状态自动机 + 子序列计数 DP」解法实现「白茄子」一题过程中所犯的全部错误，涵盖正确性错误（WA）与性能错误（TLE），并逐一给出错因分析与修复方案。

Contents

1 错误一：`next_state` 中修改了循环变量 **2**

2 错误二：`next_state` 中截断条件写反 **2**

3 错误三：BFS 建自动机时 `nxt` 表不完整 **2**

4 错误四：子序列 DP 缺少取模 **3**

5 错误五：答案计算对 f 值求和而非取最小值 **3**

6 错误六：答案计算缺少 $+\infty$ 守卫 **4**

7 错误七（TLE）：循环内 `vector` 拷贝构造导致 10^7 次堆分配 **4**

8 错误总览 **5**

1 错误一：next_state 中修改了循环变量

错误代码

```
[[nodiscard]] State next_state(ll op) const {
    array<array<ll, 2>, 2> nd{INF, INF, INF, INF};
    for (int num1 = 0; num1 < 2; ++num1) {
        for (int rev = 0; rev < 2; ++rev) {
            ll to_num1 = num1, to_rev = rev;
            if (op == 0) to_rev ^= num1;
            if (op == 1) num1 ^= 1;        // 错误：修改了循环变量 num1
            nd[to_num1][to_rev] = min(nd[to_num1][to_rev], d[num1][rev]);
            // ...
        }
    }
    return State{nd};
}
```

错误分析

if (op == 1) num1 ^= 1; 本意是计算”追加字符 1 后 1 的个数奇偶性翻转”，结果应写入 to_num1。但代码直接修改了外层循环变量 num1，导致：

- 后续对 d[num1][rev] 的读取使用了被篡改的 num1，读到的是错误的源状态。
- 循环变量被翻转后，整个遍历顺序被打乱——当 op = 1 时，内层循环对 num1 = 0 的两次迭代实际上分别使用了 num1 = 0 和 num1 = 1，完全混乱。

修复

```
ll to_num1 = num1 ^ op, to_rev = rev ^ (op == 0 ? num1 : 0);
```

用一行表达式直接计算目标状态，彻底避免对循环变量的修改。

教训

永远不要在循环体内修改循环变量的值（除非是有意的 ++i 式跳步）。当需要基于循环变量做变换时，应使用独立的临时变量存储变换结果。

2 错误二：next_state 中截断条件写反

错误代码

```
for (int num1 = 0; num1 < 2; ++num1) {
    for (int rev = 0; rev < 2; ++rev) {
        // ... 不截断的转移 ...
        if (rev == 0) {   // 错误：应该是 rev == 1
            nd[op][0] = min(nd[op][0],
                d[num1][rev] == INF ? INF : d[num1][rev] + 1);
        }
    }
}
```

错误分析

截断操作的含义是”把当前未结束子段关闭为一个完整的茄子序列段”。一个子段是茄子序列当且仅当其逆序对数为奇数，即内部状态中 rev = 1。因此只有 rev == 1 的子段才有资格被截断。

代码中 if (rev == 0) 恰好写反了：它允许逆序对数为偶数（非茄子序列）的子段被截断，却禁止了真正合法的子段被截断。

修复

```
if (rev == 1) {   // 正确：只有茄子序列（rev 为奇）才能截断
    nd[op][0] = min(nd[op][0],
        d[num1][rev] == INF ? INF : d[num1][rev] + 1);
}
```

教训

涉及”合法性判定”的条件分支，写完后必须回到定义逐字核对。本题中”茄子序列 ⇔ 逆序对为奇 ⇔ `rev == 1`”，写成 `rev == 0` 属于最基本的条件取反错误。

3 错误三：BFS 建自动机时 nxt 表不完整

错误代码

```
for (int i = 0; i < 2; ++i) {
    auto v = u.next_state(i);
    if (mp.contains(v)) {
        continue;   // 错误：跳过了，但没有记录转移边
    }
    mp[v] = idx;
    nxt[mp[u]][i] = idx;
    ++idx;
    q.push(v);
}
```

错误分析

当目标状态 v 已经在 BFS 中被访问过时，代码直接 `continue`，**没有设置 `nxt[mp[u]][i] = mp[v]`**。这导致所有”转移到已访问状态”的边全部留为初始值 -1。

自动机共 40 个状态、80 条有向边，但其中只有 39 条（BFS 树边）被正确记录，剩下 41 条（回边和交叉边）全部丢失。后续的子序列 DP 在遇到这些 -1 的转移时会直接跳过，导致大量合法的子序列转移被忽略。

修复

```
for (int i = 0; i < 2; ++i) {
    auto v = u.next_state(i);
    if (!mp.contains(v)) {
        mp[v] = idx;
        idx_state[idx] = v;
        ++idx;
        q.push(v);
    }
    nxt[mp[u]][i] = mp[v];   // 无论 v 是否新发现，都要记录转移
}
```

教训

BFS 建图时，”发现新节点”和”记录边”是两个独立的操作。即使目标节点已经在队列中，从当前节点到它的**转移边**仍然必须被记录。这是 BFS 建自动机的经典易错点。

4 错误四：子序列 DP 缺少取模

错误代码

```
for (int status = 0; status <= 45; ++status) {
    ll to_status = aut.nxt[status][ch];
    if (to_status == -1) continue;
```

```
    ndp[to_status] += dp[status];   // 错误：没有取模
}
```

错误分析

题目要求答案对 998244353 取模。`dp[status]` 本身已经是模意义下的值（$< 998244353 < 10^9$），但多次累加后 `ndp[to_status]` 可能超过 `long long` 的范围。更关键的是，不取模会导致后续与 f 值相乘时溢出，产生完全错误的结果。

修复

```
ndp[to_status] += dp[status];
if (ndp[to_status] >= mod) ndp[to_status] -= mod;
```

由于每次只加一个 $<$ mod 的值，和一定 $< 2 \times$ mod，用一次减法代替取模运算，在 10^7 的循环量级下更快。

教训

模意义下的 DP，每一步累加都必须取模。即使中间结果不溢出 `long long`，不取模也会导致后续乘法溢出或数值语义错误。

5 错误五：答案计算对 f 值求和而非取最小值

错误代码

```
for (int num1 = 0; num1 < 2; ++num1) {
    if (d[num1][1] == INF) continue;
    ll val = d[num1][1] + 1;
    ans += val * dp[status];   // 错误：把两个候选值都加进去了
    ans %= mod;
}
```

错误分析

对于一个自动机状态 $D = (d_{00}, d_{10}, d_{01}, d_{11})$，其对应的 f 值应为

$$f = \min(d_{01} + 1,\ d_{11} + 1)$$

即在所有合法（`rev` $= 1$）的末段中，选择已结束段数最少的方案，再加上末段本身的 1 段。

但代码中用 `for` 循环遍历 `num1 = 0, 1`，把 $d_{01} + 1$ 和 $d_{11} + 1$ **都**乘以 `dp[status]` 加到了答案里。当两者同时有限时（例如 $d_{01} = 0, d_{11} = 1$），正确的 $f = 1$，但代码计算出 $(0 + 1) + (1 + 1) = 3$，严重偏大。

修复

```
ll lans = INF;
for (int num1 = 0; num1 < 2; ++num1) {
    if (d[num1][1] == INF) continue;
    ll val = d[num1][1] + 1;
    lans = min(lans, val);   // 取最小值
}
if (lans == INF) continue;
ans += lans % mod * dp[status] % mod;
ans %= mod;
```

教训

f 的定义是”最小划分段数”，涉及最小值。在最终统计时必须对所有候选方案取 min，而非求和。这是把”最优化问题”和”计数问题”的操作搞混了。

6 错误六：答案计算缺少 $+\infty$ 守卫

错误代码

```
ll lans = INF;
for (int num1 = 0; num1 < 2; ++num1) {
    if (d[num1][1] == INF) continue;
    ll val = d[num1][1] + 1;
    lans = min(lans, val);
}
// 缺少：if (lans == INF) continue;
ans += lans % mod * dp[status] % mod;
ans %= mod;
```

错误分析

当某个自动机状态 D 的 d_{01} 和 d_{11} 均为 $+\infty$ 时，意味着落在该状态的子序列无法完成合法划分，其 f 值为 0，不应对答案有任何贡献。

但如果缺少 `if (lans == INF) continue;`，`lans` 仍为初始值 `INF`（$= 2^{61} - 1$），这个巨大的数值会被乘以 `dp[status]` 后加入答案，导致结果完全错误。

修复

```
if (lans == INF) continue;   // f=0 的状态不贡献答案
```

教训

使用哨兵值（如 `INF`）初始化 `min` 变量时，计算结束后必须检查结果是否仍为哨兵值。如果是，说明没有任何有效候选，必须跳过后续计算，否则哨兵值会作为”正常数值”参与运算。

7 错误七（TLE）：循环内 vector 拷贝构造导致 10^7 次堆分配

错误代码

```
for (int i = 0; i < N; ++i) {
    ll ch = s[i] - '0';
    vector<ll> ndp = dp;   // 拷贝构造：每次都 new 一块内存
    for (int status = 0; status <= 45; ++status) {
        // ...
    }
    swap(ndp, dp);
}   // ndp 析构：每次都 delete 一块内存
```

错误分析

`vector<ll> ndp = dp;` 是拷贝构造，每次调用都会在堆上分配 $50 \times 8 = 400$ 字节的新内存，循环结束时 `ndp` 析构又将其释放。N 最大为 10^7，因此总共触发 10^7 次 `malloc` + 10^7 次 `free`，系统调用开销远超 DP 本身的计算量。

修复

将 `ndp` 的声明提到循环外部，循环内用赋值代替构造：

```
vector<ll> ndp;              // 提到循环外
for (int i = 0; i < N; ++i) {
    ll ch = s[i] - '0';
    ndp = dp;                // 赋值：size 相同时不会重新分配内存
    for (int status = 0; status <= 45; ++status) {
        // ...
```

```
    }
    dp.swap(ndp);              // O(1)，只交换内部指针
}
```

当 ndp 和 dp 的 size 相同时，ndp = dp 只做 memcpy，不触发堆分配。而 swap 只交换三个指针，开销为 $O(1)$。整个循环从 10^7 次堆操作降为 0 次。

教训

在热循环（$\geq 10^6$ 次）中，绝对不能使用会触发堆内存分配的操作。vector 的拷贝构造、push_back 触发扩容、= {} 赋空值释放内存，都是常见的隐性性能杀手。应优先使用 clear() 保留容量、预分配 + 赋值复用、或直接使用定长数组。

8 错误总览

编号	错误描述	类型	后果
1	next_state 修改循环变量 num1	逻辑	WA
2	截断条件 rev==0 应为 rev==1	逻辑	WA
3	BFS 未记录到已访问状态的转移边	遗漏	WA
4	子序列 DP 累加后未取模	遗漏	WA
5	f 值对两个候选求和而非取 min	语义	WA
6	缺少 $+\infty$ 守卫导致哨兵值参与计算	遗漏	WA
7	循环内 vector 拷贝构造触发 10^7 次堆分配	性能	TLE