白茄子

题目大意

给定一个只包含 `0` 和 `1` 的字符串 S。

如果一个 01 序列的逆序对数量为奇数，那么称它为“茄子序列”。

对于任意一个 01 序列 T，定义 $f(T)$ 为最小的正整数 k，使得我们能够把 T 划分成 k 个连续子段，并且每个子段都是茄子序列。如果无论如何划分都做不到，则定义 $f(T)=0$。

题目要求原串 S 的所有非空子序列的 f 值之和，对 998244353 取模。

解题思路

总长度上界达到 10^7，显然不可能枚举子序列，更不可能对每个子序列再跑一次分段 DP。

这道题真正要做的事情只有两步：

- 先把“一个固定的 01 串，它的 f 值是多少”压成一个很小的自动机状态。
- 再在原串上做“子序列自动机 DP”，统计有多少个子序列落到每个状态。

下面从头推导这个状态。

1. 先只看一个子段，什么信息才是必要的

如果我们只关心一个连续子段是否为茄子序列，那么真正有用的只有两个奇偶量：

- a：当前子段中 `1` 的个数奇偶。
- p：当前子段中逆序对个数奇偶。

原因很简单。往当前子段末尾再加入一个字符时：

- 如果加入的是 `1`，那么 `1` 的个数奇偶翻转，但不会新增任何逆序对。
- 如果加入的是 `0`，那么它会与前面所有的 `1` 形成逆序对，所以逆序对奇偶需要额外异或上“当前 `1` 的个数奇偶”。

于是转移就是：

$$(a,p)\xrightarrow{1}(a\oplus 1,\,p)$$

$$(a,p)\xrightarrow{0}(a,\,p\oplus a)$$

因此，一个“尚未结束的当前子段”只有四种内部状态：

$$(0,0),\ (1,0),\ (0,1),\ (1,1)$$

而一个子段是否合法，只看最后的 p 是否为 1。也就是说，合法状态恰好是：

$$(0,1),\ (1,1)$$

2. 从单段状态，升级到整串最优分段 DP

上一小节讨论的，其实只是“**一个单独的段**内部需要记录什么状态”。

现在我们真正要算的是整个序列 T 的 $f(T)$，也就是“最少能切成多少个合法段”。看起来像是突然从“单段”跳到了“多段”，但中间其实只差一步非常自然的抽象。

当我们从左到右扫描 T 的某个前缀时，当前局面一定可以描述成：

- 前面若干段已经彻底结束了，而且这些段都合法。
- 最后还挂着一段没有结束，后面的字符只能继续接在这一段后面。

所以，过去的切分细节里，真正会影响未来的只有两件事：

- 这“最后一段”现在处于什么内部状态。
- 在它前面，已经结束了多少个合法段。

换句话说，我们可以把所有“最后一段状态相同”的方案合并，只保留其中“已经结束段数最小”的那个数字。

这正是你说的那种理解：**把状态相同的段，抽象成一个状态配合一个数字**。

于是我们定义：

$$d_{xy}$$

表示当前已经读完某个前缀后，最后那个未结束子段的内部状态是 (x, y) 时，前面已经结束的合法子段数的最小值。

这里要特别注意，d_{xy} 记录的是“**整条前缀的最优摘要**”，而不是“某一个单独子段的值”。四个 d 合在一起，才描述了当前前缀的整体情况。

另外，状态 $(0,0)$ 也不等于“空段”，它只是表示当前未结束子段的内部统计量恰好回到了 $(0,0)$。例如一个子段读成 `11` 之后，也会落在这个状态。

初始时我们还没有读任何字符，相当于当前只有一个空子段，它的状态是 $(0,0)$，因此：

$$d_{00} = 0, \qquad d_{10} = d_{01} = d_{11} = +\infty$$

现在读下一个字符时，先不要急着把它接上去，而是先问一句：**要不要在这个字符前面切一刀？**

如果当前未结束子段已经合法，也就是它处于 $(0,1)$ 或 $(1,1)$，那么我们就可以选择在这里截断，让它变成一个完整答案段，然后重新开始构造下一段。

因此，在读入下一个字符之前，能够以状态 $(0,0)$ 去承接这个字符的最优代价是：

$$r = \min\left(d_{00},\, d_{01} + 1,\, d_{11} + 1\right)$$

这三项分别对应：

- 不截断，继续沿用原来状态就是 $(0,0)$ 的未结束子段。
- 把一个合法的 $(0,1)$ 子段截断掉，于是已结束段数加一，再新开一段。
- 把一个合法的 $(1,1)$ 子段截断掉，于是已结束段数加一，再新开一段。

接下来，才是把这个新字符真正接到“当前最后一段”后面。由于上一小节已经求出了**单段内部状态**的转移，所以这里直接套用即可。

如果读入的是 `0`：

$$\begin{cases} d'_{00} = r \\ d'_{10} = d_{11} \\ d'_{01} = d_{01} \\ d'_{11} = d_{10} \end{cases}$$

如果读入的是 `1`：

$$\begin{cases} d'_{00} = d_{10} \\ d'_{10} = r \\ d'_{01} = d_{11} \\ d'_{11} = d_{01} \end{cases}$$

这样一来，“单段状态”就自然升级成了“整串前缀的最优分段 DP”。

整串处理结束以后，最后那个未结束子段本身也必须合法，所以只有状态 $(0,1)$ 和 $(1,1)$ 可以作为最终答案。于是：

$$f(T) = \begin{cases} 0, & \min(d_{01}, d_{11}) = +\infty \\ \min(d_{01}, d_{11}) + 1, & \text{否则} \end{cases}$$

这里最后的 $+1$，就是把最后那个合法但尚未结算的子段也计入答案。

3. 把“状态 + 数字”的整体压成自动机状态

到这里，我们不再只看“一个段是什么状态”，而是在看“**整个前缀的最优摘要**”。

这个摘要可以写成四元组：

$$D = (d_{00}, d_{10}, d_{01}, d_{11})$$

它的含义可以理解成：

- 最后一段如果处于 $(0,0)$，最少已经结束了多少段。

- 最后一段如果处于 $(1,0)$，最少已经结束了多少段。
- 最后一段如果处于 $(0,1)$，最少已经结束了多少段。
- 最后一段如果处于 $(1,1)$，最少已经结束了多少段。

所以这四个数，本质上就是“**四种单段状态，各自配了一个最优数字**”。

这也就是为什么第三部分可以直接接着第二部分往下讲。因为第二部分已经把“多段问题”压成了一个四元组 D，而第三部分做的事情，只是进一步说明：**如果两个前缀的四元组 D 完全一样，那么它们后面无论再接什么字符，行为都会完全一样**。

原因很直接。无论读入 `0` 还是 `1`，新的四元组只由旧的四元组通过固定公式算出来；最后的 $f(T)$ 也只由这个四元组决定。既然如此，前缀本身长什么样已经不重要了，重要的只有这个四元组。

于是我们就可以把四元组 D 直接当成自动机状态。

接下来还有一个问题：这样的状态会不会很多？

乍一看，$d_{00}, d_{10}, d_{01}, d_{11}$ 都可能越来越大，状态数似乎会爆炸。但这个 DP 有一个很强的性质：**所有可达状态的最小有限值始终都是 0**。

理由并不复杂：

- 如果当前 $d_{00} = 0$，那么 $r = 0$，新状态里自然还会保留一个 0。
- 如果当前 $d_{00} \neq 0$，由于最小有限值是 0，那么 d_{10}, d_{01}, d_{11} 中至少有一个是 0。而无论读入 `0` 还是 `1`，这些位置里的某些值都会被直接复制到新状态中，所以新状态里仍然会有一个 0。

这件事非常关键。它说明我们不需要额外记录“整体平移了多少”，状态本身就是绝对的。

于是，从初始状态

$$(0, +\infty, +\infty, +\infty)$$

出发，按照上面的 `0/1` 转移去 BFS 枚举所有**实际可达状态**，总共只会得到 40 个。

不过总长是 10^7，常数仍然值得继续优化。所以我们再做一次标准 DFA 最小化：

- 先按当前输出值 $f(T)$ 分类。
- 再按读入 `0` 和 `1` 后会跳到哪个等价类继续细分。

最终只剩下 15 个状态。

于是，任意一个子序列都会被压缩到这 15 个状态之一，而这个状态就已经唯一决定了它的 f 值。

4. 对所有子序列做自动机 DP

现在问题就简单了。

设最小化后的自动机状态数为 m，定义：

$$\texttt{cnt}[q]$$

表示当前已经处理完原串某个前缀后，有多少个子序列会落在状态 q。

初始时只有空子序列存在，它落在自动机初态：

$$\texttt{cnt}[\text{start}] = 1$$

接下来从左到右扫描原串的每个字符 c。对任意已有子序列，都有两种选择：

- 不选这个字符，那么状态不变。
- 选这个字符，那么状态从 q 转移到 $\texttt{nxt}[q][c]$。

因此每次转移都只是一个经典的“选或不选”子序列 DP：

- 先把“不选”的贡献整体继承下来。
- 再把“选”的贡献加到对应的后继状态上。

处理完整个原串以后，所有子序列都已经被分到了某个自动机状态里。由于状态已经唯一决定了 f 值，所以答案就是：

$$\sum_{q=0}^{m-1} \texttt{cnt}[q] \times \texttt{val}[q]$$

注意空子序列也在里面，但它始终停在初态，而初态的 $\texttt{val} = 0$，所以不会产生任何贡献，完全不需要额外减去。

代码实现

下面给出完整实现。代码会在程序启动时自动构造那 40 个原始状态，再最小化成 15 个状态，之后对每个测试串做线性 DP。

```cpp
#include <bits/stdc++.h>
using namespace std;

static constexpr int MOD = 998244353;
static constexpr int INF = (int)1e9;

using State = array<int, 4>; // 顺序为 d00, d10, d01, d11

struct Automaton {
    int start = 0;
    vector<array<int, 2>> nxt;
    vector<int> val; // 该状态对应的 f(T)
};

int add_one(int x) {
    return x >= INF ? INF : x + 1;
}

// 在“固定串最少分段 DP”里，读入一个新字符后的状态转移
State move_state(const State& d, int bit) {
    int restart = min({d[0], add_one(d[2]), add_one(d[3])});
    if (bit == 0) {
        return State{restart, d[3], d[2], d[1]};
    } else {
        return State{d[1], restart, d[3], d[2]};
    }
}

// 当前状态本身对应的 f(T)
int get_value(const State& d) {
    int best = min(d[2], d[3]);
    return best >= INF ? 0 : best + 1;
}

// 从初态出发，BFS 枚举所有可达的原始状态
Automaton build_raw_automaton() {
    vector<State> states;
    vector<array<int, 2>> nxt;
    vector<int> val;
    map<State, int> id;
    queue<int> q;

    auto get_id = [&](const State& s) -> int {
        auto it = id.find(s);
        if (it != id.end()) return it->second;
        int nid = (int)states.size();
        id[s] = nid;
        states.push_back(s);
        nxt.push_back({0, 0});
        val.push_back(0);
        q.push(nid);
        return nid;
    };

    Automaton raw;
    raw.start = get_id(State{0, INF, INF, INF});

    while (!q.empty()) {
        int u = q.front();
```

```
        q.pop();

        val[u] = get_value(states[u]);
        for (int bit = 0; bit < 2; ++bit) {
            State ns = move_state(states[u], bit);
            nxt[u][bit] = get_id(ns);
        }
    }

    raw.nxt = move(nxt);
    raw.val = move(val);
    return raw;
}

// 标准 DFA 最小化。因为状态数只有常数个，这里直接反复细分即可
Automaton minimize_automaton(const Automaton& raw) {
    int n = (int)raw.nxt.size();
    vector<int> cls(n);

    {
        map<int, int> mp;
        for (int i = 0; i < n; ++i) {
            if (!mp.count(raw.val[i])) {
                mp[raw.val[i]] = (int)mp.size();
            }
            cls[i] = mp[raw.val[i]];
        }
    }

    while (true) {
        map<tuple<int, int, int>, int> mp;
        vector<int> ncls(n);

        for (int i = 0; i < n; ++i) {
            auto sig = make_tuple(
                raw.val[i],
                cls[raw.nxt[i][0]],
                cls[raw.nxt[i][1]]
            );
            if (!mp.count(sig)) {
                mp[sig] = (int)mp.size();
            }
            ncls[i] = mp[sig];
        }

        if (ncls == cls) break;
        cls.swap(ncls);
    }

    int m = *max_element(cls.begin(), cls.end()) + 1;
    vector<int> repr(m, -1);
    for (int i = 0; i < n; ++i) {
        if (repr[cls[i]] == -1) repr[cls[i]] = i;
    }

    Automaton aut;
    aut.start = cls[raw.start];
    aut.nxt.assign(m, {0, 0});
    aut.val.assign(m, 0);

    for (int c = 0; c < m; ++c) {
        int i = repr[c];
        aut.val[c] = raw.val[i];
```

```
        aut.nxt[c][0] = cls[raw.nxt[i][0]];
        aut.nxt[c][1] = cls[raw.nxt[i][1]];
    }

    return aut;
}

int solve_one(const string& s, const Automaton& aut) {
    int m = (int)aut.nxt.size();
    vector<int> dp(m, 0), ndp(m, 0);

    // 空子序列
    dp[aut.start] = 1;

    for (char ch : s) {
        int bit = ch - '0';

        // 不选当前字符
        ndp = dp;

        // 选当前字符
        for (int q = 0; q < m; ++q) {
            if (dp[q] == 0) continue;
            int nq = aut.nxt[q][bit];
            ndp[nq] += dp[q];
            if (ndp[nq] >= MOD) ndp[nq] -= MOD;
        }

        dp.swap(ndp);
    }

    long long ans = 0;
    for (int q = 0; q < m; ++q) {
        ans = (ans + 1LL * dp[q] * aut.val[q]) % MOD;
    }
    return (int)ans;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    Automaton raw = build_raw_automaton();
    Automaton aut = minimize_automaton(raw);

    int T;
    cin >> T;
    while (T--) {
        string s;
        cin >> s;
        cout << solve_one(s, aut) << '\n';
    }
    return 0;
}
```

复杂度分析

- **预处理复杂度**：自动机建图和最小化都只在常数状态上进行，复杂度是 $\mathcal{O}(1)$。
- **单组时间复杂度**：设最小化后状态数为 $m = 15$，扫描每个字符时做一次大小为 m 的 DP，故时间复杂度为 $\mathcal{O}(15 \times |S|)$，也就是线性复杂度。
- **总时间复杂度**：$\mathcal{O}(15 \times \sum |S|)$。

- **空间复杂度**：自动机和 DP 数组都只有常数大小，空间复杂度为 $\mathcal{O}(1)$。

小结

这道题最难的地方，不是最后的子序列 DP，而是先把“最少分段数”这个看起来很全局的量压成一个很小的自动机状态。

一旦完成了这一步，后面的部分就会变成非常标准的套路：**子序列计数 + 自动机转移**。整个算法的本质，就是先把复杂条件局部化，再把所有子序列统一交给自动机去分类统计。