# 题目

给定一个字符串列表 dict ，其中所有字符串的长度都相同。

当存在两个字符串在相同索引处只有一个字符不同时，返回 True ，否则返回 False 。

示例 1：

输入：dict = ["abcd","acbd", "aacd"]

输出：true

解释：字符串 "abcd" 和 "aacd" 只在索引 1 处有一个不同的字符。

示例 2：

输入：dict = ["ab","cd","yz"]

输出：false

示例 3：

输入：dict = ["abcd","cccc","abyd","abab"]

输出：true

提示：

dict 中的字符数小于或等于 10^5 。

dict[i].length == dict[j].length

dict[i] 是互不相同的。

dict[i] 只包含小写英文字母。

进阶：你可以以 O(n\*m) 的复杂度解决问题吗？其中 n 是列表 dict 的长度，m 是字符串的长度。

# 分析

## 方法一：模式串匹配

要解决这个问题，我们需要判断字符串列表中是否存在两个字符串，它们在相同索引处仅有一个字符不同。核心挑战是在保证正确性的前提下，优化时间复杂度（避免暴力对比的O(n²m)），最终实现O(nm)的高效解法。

思路：

问题的关键是如何快速识别“仅有一个字符不同”的字符串对。直接暴力对比所有字符串对（O(n²m)）会因n最大为1e5而超时，因此需要更聪明的方法：

两个字符串s和t仅有一个字符不同，意味着：若将它们在某一索引k处的字符替换为一个特殊标记（如\*），得到的“模式串”会完全相同。

例如：s=abcd、t=aacd，在索引1处替换为\*后，两者的模式串均为a\*cd。

具体步骤

1、生成模式串：遍历每个字符串s，对其每个索引k（0≤k<m，m为字符串长度），将s[k]替换为特殊标记（如\*），生成m个模式串。

2、记录模式串：用哈希表存储所有生成的模式串。若某一模式串被重复记录，说明存在两个字符串通过替换同一索引的字符得到了该模式串——这两个字符串即为“仅有一个字符不同”的目标对。

3、判断结果：若遍历过程中发现重复的模式串，直接返回true；遍历结束未发现则返回false。

代码：

class Solution {

public:

bool differByOne(vector<string>& dict) {

if (dict.empty() || dict.size() == 1) return false; // 空列表或仅1个字符串，无对比对象

int n = dict.size(); // 字符串数量

int m = dict[0].size(); // 每个字符串的长度（题目保证所有字符串长度相同）

unordered\_set<string> patternSet; // 存储所有生成的模式串

for (const string& s : dict) {

// 对当前字符串的每个索引k，生成模式串（替换s[k]为'\*'）

for (int k = 0; k < m; ++k) {

// 生成模式串：前k个字符 + '\*' + 从k+1开始的剩余字符

string pattern = s.substr(0, k) + "\*" + s.substr(k + 1);

// 若模式串已存在，说明找到目标字符串对

if (patternSet.count(pattern)) {

return true;

}

// 若不存在，加入哈希表

patternSet.insert(pattern);

}

}

// 遍历结束未发现重复模式串，返回false

return false;

}

};

代码解释：

1、边界处理：若列表为空或仅有1个字符串，直接返回false（无两个字符串可对比）。

2、遍历每个字符串：对列表中的每个字符串s，生成其所有可能的“单字符替换为\*”的模式串。

- 生成模式串的方式：用substr(0, k)获取前k个字符，用substr(k+1)获取k之后的字符，中间插入\*，形成a\*cd这类模式。

3、哈希表判断重复：

- 若生成的模式串已在patternSet中，说明之前有另一个字符串通过替换同一索引的字符得到了相同模式——这两个字符串仅有一个字符不同，返回true。

- 若模式串不存在，将其加入哈希表，供后续字符串对比。

4、最终结果：遍历所有字符串和所有索引后，未发现重复模式串则返回false。

复杂度分析：

- 时间复杂度：O(nm)，其中n是字符串数量，m是字符串长度。

每个字符串生成m个模式串，共生成n×m个模式串。

哈希表的count和insert操作平均时间为O(1)，因此总时间为O(nm)，完全满足题目进阶要求。

- 空间复杂度：O(nm)，最坏情况下所有模式串均不重复，哈希表需存储n×m个模式串（每个模式串长度为m，空间可视为O(nm)）。

关键优势

1、避免暴力对比：无需遍历所有字符串对（O(n²)），直接通过模式串关联“仅有一个字符不同”的字符串，效率大幅提升。

2、正确性保障：仅当两个字符串在同一索引处有且仅有一个字符不同时，才会生成相同的模式串，不存在误判。

3、适配约束：即使n=1e5、m=100，n×m=1e7的操作量也在时间限制内，实用性强。

上述代码执行会出现内存超限。

## 方法二：哈希值

当出现“超出内存限制”时，核心原因是直接存储完整模式串（如a\*cd）会占用大量空间——对于n=1e5、m=100的场景，需存储1e7个长度为m的字符串，总内存占用可达数百MB甚至GB，超出题目限制。

解决方案是用“哈希值”替代完整模式串：不存储整个模式串，而是计算每个模式串的哈希值并存储，通过对比哈希值判断模式串是否重复。哈希值（如64位整数）的存储空间远小于完整字符串，可大幅降低内存消耗。

优化思路：模式串哈希化

两个字符串生成的“同一模式”（如abcd和aacd在索引1替换为\*的模式），其哈希值必然相同；反之，若两个模式的哈希值相同（忽略极低的哈希冲突），则认为模式相同。

为高效计算模式串的哈希值，我们使用前缀哈希+后缀哈希的预处理策略：

1、预处理前缀哈希数组prefix：prefix[i][k]表示第i个字符串前k个字符的哈希值（从0到k-1）。

2、预处理后缀哈希数组suffix：suffix[i][k]表示第i个字符串从k到末尾的哈希值（从k到m-1）。

3、计算模式串哈希：对于第i个字符串的第k个模式（替换s[i][k]为\*），其哈希值 = 前缀哈希（前k个字符） + 后缀哈希（从k+1开始），通过组合这两个哈希值得到唯一标识。

为避免哈希冲突，可使用双哈希（两个不同的基数和模），但实际场景中单哈希（64位无符号整数，利用自然溢出）已足够应对题目约束。

优化后代码（无内存溢出）：

class Solution {

public:

bool differByOne(vector<string>& dict) {

if (dict.empty() || dict.size() == 1) return false;

int n = dict.size(); // 字符串数量

int m = dict[0].size(); // 字符串长度

const uint64\_t BASE = 911382629; // 哈希基数（质数，减少冲突）

// 1. 预处理每个字符串的前缀哈希和后缀哈希

vector<vector<uint64\_t>> prefix(n, vector<uint64\_t>(m + 1, 0)); // prefix[i][k]：s[i]前k个字符的哈希（0~k-1）

vector<vector<uint64\_t>> suffix(n, vector<uint64\_t>(m + 1, 0)); // suffix[i][k]：s[i]从k开始的哈希（k~m-1）

vector<uint64\_t> power(m + 1, 1); // 预计算BASE的幂次：power[k] = BASE^k（用于后缀哈希对齐）

// 预计算BASE的幂次（power[0]=1, power[1]=BASE, ..., power[m]=BASE^m）

for (int k = 1; k <= m; ++k) {

power[k] = power[k - 1] \* BASE;

}

// 计算每个字符串的前缀哈希和后缀哈希

for (int i = 0; i < n; ++i) {

const string& s = dict[i];

// 前缀哈希：prefix[i][k] = s[0]\*BASE^(k-1) + s[1]\*BASE^(k-2) + ... + s[k-1]

for (int k = 1; k <= m; ++k) {

prefix[i][k] = prefix[i][k - 1] \* BASE + (s[k - 1] - 'a' + 1); // +1避免0值（如'a'对应1，而非0）

}

// 后缀哈希：suffix[i][k] = s[k]\*BASE^(m-k-1) + s[k+1]\*BASE^(m-k-2) + ... + s[m-1]

for (int k = m - 1; k >= 0; --k) {

suffix[i][k] = suffix[i][k + 1] + (s[k] - 'a' + 1) \* power[m - k - 1];

}

}

// 2. 遍历每个字符串的每个模式，用哈希值判断是否重复

unordered\_set<uint64\_t> hashSet;

for (int i = 0; i < n; ++i) {

for (int k = 0; k < m; ++k) {

// 计算模式k的哈希值：前缀哈希（0~k-1） \* BASE^(m-k) + 后缀哈希（k+1~m-1）

// （乘BASE^(m-k)是为了将前缀哈希对齐到与后缀哈希相同的位数，避免不同模式的哈希碰撞）

uint64\_t patternHash = prefix[i][k] \* power[m - k] + suffix[i][k + 1];

if (hashSet.count(patternHash)) {

return true; // 找到重复模式，存在仅一个字符不同的字符串对

}

hashSet.insert(patternHash);

}

}

return false;

}

};

优化点解释

1、哈希值替代完整模式串：

每个模式串的哈希值是一个64位无符号整数（uint64\_t），仅占8字节；而原模式串（如长度100的字符串）占100字节。

对于n=1e5、m=100的场景，哈希表存储1e7个哈希值，总内存仅1e7 \* 8字节 = 80MB，远低于原方案的数百MB。

2、前缀+后缀哈希高效计算：

预处理前缀和后缀哈希后，每个模式的哈希值可通过O(1)计算得到，避免了每次生成模式串的O(m)时间开销（虽时间复杂度仍为O(nm)，但实际运行更快）。

预计算power数组（BASE的幂次），用于后缀哈希的位数对齐，确保不同位置的模式哈希值唯一。

3、避免0值哈希：

计算哈希时将字符转换为(s[k]-'a'+1)（如'a'→1，'b'→2），而非直接用(s[k]-'a')（避免全'a'字符串的前缀哈希为0，导致不同模式的哈希冲突）。

为什么不会内存溢出？

预处理数组的内存：prefix和suffix均为n×(m+1)的uint64\_t数组，对于n=1e5、m=100，总内存为2 \* 1e5 \* 101 \* 8字节 ≈ 161.6MB。

哈希表的内存：存储1e7个uint64\_t，总内存80MB。

总内存消耗：约240MB，远低于多数OJ的内存限制（通常为512MB或1GB）。

极端场景适配

若m极大（如m=1e3），可进一步优化：不存储完整的prefix和suffix数组，而是对每个字符串实时计算前缀哈希和后缀哈希（遍历字符串时同步计算，计算完模式哈希后丢弃），可将预处理数组的内存从O(nm)降至O(m)，进一步减少内存占用。