# 题目

给你一个字符串s，请你将s分割成一些子串，使每个子串都是 回文串 。返回s所有可能的分割方案。

回文串是正着读和反着读都一样的字符串。

示例 1：

输入：s = "aab"

输出：[["a","a","b"],["aa","b"]]

示例 2：

输入：s = "a"

输出：[["a"]]

提示：

1 <= s.length <= 16

s仅由小写英文字母组成

# 分析

要解决字符串分割为所有回文子串的问题，我们可以采用回溯法结合回文串预处理的策略。回溯法用于探索所有可能的分割方案，预处理则通过动态规划快速判断任意子串是否为回文，优化时间效率。

解题思路

1、核心逻辑：

分割问题本质是“选择分割点”：从字符串起始位置开始，依次尝试在每个位置分割，若分割出的子串是回文，则继续递归分割剩余部分；若不是，则跳过该分割点。

当分割到字符串末尾时，当前的分割方案即为一个有效解，加入结果列表。

2、回文串预处理（动态规划）：

- 提前构建一个二维数组is\_pal，其中is\_pal[i][j]表示子串s[i..j]（从索引i到j，闭区间）是否为回文串。

- 状态转移方程：

- 若i == j（单个字符）：is\_pal[i][j] = true；

- 若j == i + 1（两个字符）：is\_pal[i][j] = (s[i] == s[j])；

- 若j > i + 1（多个字符）：is\_pal[i][j] = (s[i] == s[j]) && is\_pal[i+1][j-1]（两端字符相等且中间子串是回文）。

- 预处理顺序：按子串长度从短到长计算（先处理长度1，再长度2，直到长度n），确保计算is\_pal[i][j]时，is\_pal[i+1][j-1]已确定。

3、回溯过程：

- 用path记录当前的分割方案（如["a","a"]），start记录当前分割的起始位置。

- 从start开始遍历所有可能的分割终点end：

- 若s[start..end]是回文（is\_pal[start][end]为true），则将该子串加入path，递归分割start = end + 1的剩余部分；

- 递归结束后，执行“回溯”：将path中最后一个子串删除，尝试下一个分割点。

- 当start == n（n为字符串长度）时，path已包含一个完整的分割方案，加入结果列表。

代码：

class Solution {

public:

vector<vector<string>> partition(string s) {

int n = s.size();

// 预处理：is\_pal[i][j] 表示 s[i..j] 是否为回文串

vector<vector<bool>> is\_pal(n, vector<bool>(n, false));

for (int len = 1; len <= n; ++len) { // 按子串长度从1到n遍历

for (int i = 0; i + len - 1 < n; ++i) { // i是起始索引，j是结束索引

int j = i + len - 1;

if (i == j) {

is\_pal[i][j] = true;

} else if (j == i + 1) {

is\_pal[i][j] = (s[i] == s[j]);

} else {

is\_pal[i][j] = (s[i] == s[j]) && is\_pal[i+1][j-1];

}

}

}

vector<vector<string>> result; // 存储所有分割方案

vector<string> path; // 存储当前分割方案

backtrack(s, 0, is\_pal, path, result);

return result;

}

private:

// 回溯函数：start是当前分割的起始位置

void backtrack(const string& s, int start, const vector<vector<bool>>& is\_pal,

vector<string>& path, vector<vector<string>>& result) {

int n = s.size();

// 分割到字符串末尾，当前方案有效，加入结果

if (start == n) {

result.push\_back(path);

return;

}

// 尝试从start到end的所有分割点

for (int end = start; end < n; ++end) {

if (is\_pal[start][end]) { // 若s[start..end]是回文，继续分割

path.push\_back(s.substr(start, end - start + 1)); // 加入当前子串

backtrack(s, end + 1, is\_pal, path, result); // 递归分割剩余部分

path.pop\_back(); // 回溯：删除最后一个子串，尝试下一个分割点

}

}

}

};

解释

1、回文预处理：

- 例如对于s = "aab"，预处理后is\_pal数组的关键值为：

- is\_pal[0][0] = true（"a"）、is\_pal[0][1] = true（"aa"）、is\_pal[0][2] = false（"aab"）；

- is\_pal[1][1] = true（"a"）、is\_pal[1][2] = false（"ab"）；

- is\_pal[2][2] = true（"b"）。

2、回溯过程（以`s = "aab"`为例）：

- 初始start = 0，path = []：

- end = 0：s[0..0]`是回文，path = ["a"]，递归start = 1；

- start = 1，end = 1：s[1..1]是回文，path = ["a","a"]，递归start = 2；

- start = 2，end = 2：s[2..2]是回文，path = ["a","a","b"]，递归start = 3（等于n=3），加入结果；

- 回溯：path变为["a","a"]，end遍历结束，返回上一层。

- 回溯：path变为["a"]，end = 2：s[1..2]不是回文，跳过；返回上一层。

- 回溯：path变为[]，end = 1：s[0..1]是回文，path = ["aa"]，递归start = 2；

- start = 2，end = 2：s[2..2]是回文，path = ["aa","b"]，递归start = 3，加入结果；

- 回溯：path变为["aa"]，end遍历结束，返回上一层。

- end = 2：s[0..2]不是回文，跳过；回溯结束。

- 最终结果为[["a","a","b"], ["aa","b"]]，与示例一致。

复杂度分析

- 时间复杂度：O(n \* 2^n)，其中`n`是字符串长度。

预处理回文数组：O(n^2)；

回溯过程：最多有2^(n-1)种分割方案（每个位置可选择分割或不分割），每种方案需`O(n)`时间复制到结果列表，因此总时间为O(n \* 2^n)。

- 空间复杂度：O(n^2)，主要用于存储回文数组is\_pal（O(n^2)）和递归栈（最坏情况下O(n)，可忽略）。

该方法通过预处理优化了回文判断，同时利用回溯法完整探索所有分割方案，高效解决了问题。