# 题目

给你一个字符串 s，找到 s 中最长的回文子串。

**示例 1：**

输入：s = "babad"

输出："bab"

解释："aba" 同样是符合题意的答案。

**示例 2：**

输入：s = "cbbd"

输出："bb"

**示例 3：**

输入：s = "a"

输出："a"

**示例 4：**

输入：s = "ac"

输出："a"

**提示：**

1 <= s.length <= 1000

s 仅由数字和英文字母（大写和/或小写）组成

# 分析

## 方法一：暴力破解

class Solution {

public:

string longestPalindrome(string s) {

string res="";//存放结果

string temp="";//存放子串

for(int i=0;i<s.length();i++)

{

for(int j=i;j<s.length();j++)

{

temp=temp+s[j];

string tem=temp;//tem存放子串反转结果

std::reverse(tem.begin(),tem.end());//反转

if(temp==tem)

res=res.length()>temp.length()?res:temp;

}

temp="";

}

return res;

}

};

## 方法二：动态规划

**思路：**

初始状态：

dp[i][i]=1; //单个字符是回文串

dp[i][i+1]=1 if s[i]=s[i+1]; //连续两个相同字符是回文串

**代码：**

class Solution {

public:

string longestPalindrome(string s) {

int len=s.size();

if(len==0||len==1)

return s;

int start=0; //回文串起始位置

int max=1; //回文串最大长度

vector<vector<int>> dp(len,vector<int>(len));//定义二维动态数组

for(int i=0;i<len;i++)//初始化状态

{

dp[i][i]=1;

if(i<len-1&&s[i]==s[i+1])

{

dp[i][i+1]=1;

max=2;

start=i;

}

}

for(int l=3;l<=len;l++)

//l表示检索的子串长度，等于3表示先检索长度为3的子串

{

for(int i=0;i+l-1<len;i++)

{

int j=l+i-1;//终止字符位置

if(s[i]==s[j]&&dp[i+1][j-1]==1)//状态转移

{

dp[i][j]=1;

start=i;

max=l;

}

}

}

return s.substr(start,max);//获取最长回文子串

}

};

**官方答案：**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class Solution {

public:

string longestPalindrome(string s) {

int n = s.size();

if (n < 2) {

return s;

}

int maxLen = 1;

int begin = 0;

// dp[i][j] 表示 s[i..j] 是否是回文串

vector<vector<int>> dp(n, vector<int>(n));

// 初始化：所有长度为 1 的子串都是回文串

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

dp[i][i] = true;

}

// 递推开始

// 先枚举子串长度

for (int L = 2; L <= n; L++) {

// 枚举左边界，左边界的上限设置可以宽松一些

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

// 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得

int j = L + i - 1;

// 如果右边界越界，就可以退出当前循环

if (j >= n) {

break;

}

if (s[i] != s[j]) {

dp[i][j] = false;

} else {

if (j - i < 3) {

dp[i][j] = true;

} else {

dp[i][j] = dp[i + 1][j - 1];

}

}

// 只要dp[i][L] == true成立，就表示子串s[i..L]是回文，此时记录回文长度和起始位置

if (dp[i][j] && j - i + 1 > maxLen) {

maxLen = j - i + 1;

begin = i;

}

}

}

return s.substr(begin, maxLen);

}

};

**复杂度分析：**

时间复杂度：O(n^2)，其中n是字符串的长度。动态规划的状态总数为O(n^2)，对于每个状态，我们需要转移的时间为O(1)。

空间复杂度：O(n^2)，即存储动态规划状态需要的空间。