Longest Palindromic Substring

Medium

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.

**Example 1:**

Input: "babad"  
Output: "bab"  
Note: "aba" is also a valid answer.

**Example 2:**

Input: "cbbd"  
Output: "bb"

**解法1**

暴力破解。会超时

string longestPalindrome(string s) {  
 if(s.size() == 0)return "";  
 string res = "";  
 for(int i = 0; i < s.size(); ++i){  
 for(int j = i; j < s.size(); ++j){  
 int p = i, q = j;  
 bool flag = true;  
 while(p <= q){  
 if(s[p] == s[q]){  
 p++, q--;  
 }else{  
 flag = false;  
 break;  
 }  
 }  
 if(flag && j - i + 1 > res.size())res = s.substr(i, j - i + 1);  
 }  
 }  
 return res;  
}

**解法2**

动态规划

To improve over the brute force solution, we first observe how we can avoid unnecessary re-computation while validating palindromes. Consider the case “ababa”. If we already knew that “bab” is a palindrome, it is obvious that “ababa” must be a palindrome since the two left and right end letters are the same.

We defineowing:

Therefore,

The base cases are:

This yields a straight forward DP solution, which we first initialize the one and two letters palindromes, and work our way up finding all three letters palindromes, and so on…

**Complexity Analysis**

* Time complexity : .
* Space complexity : .

string longestPalindrome(string s) {  
 if(s.size() == 0)return "";  
 int len = s.size();  
 bool dp[len][len];  
 fill(dp[0], dp[0] + len \* len, false);  
 int maxlen = 1, start = 0;  
 for(int i = 0; i < len; ++i){  
 dp[i][i] = true;  
 if(i < len - 1 && s[i] == s[i + 1]){  
 dp[i][i + 1] = true;  
 maxlen = 2;  
 start = i;  
 }  
 }  
 for(int dl = 2; dl < len; ++dl){  
 for(int i = 0; i < len; ++i){  
 if(i + dl <len){  
 if(s[i] == s[i + dl] && dp[i + 1][i + dl - 1] == true){  
 dp[i][i + dl] = true;  
 if(dl + 1 > maxlen){  
 maxlen = dl + 1;  
 start = i;  
 }  
 }  
 }else{  
 break;  
 }  
 }  
 }  
 return s.substr(start, maxlen);  
}

==Note:==

题目中的dp数组是一个上三角矩阵，可以进行压缩存储，能节省一半的数组空间

**解法3**

中心扩展法。最开始想到了这种办法，但是没有考虑到中心是两个字符的情况。代码中用函数expand()实现扩展

class Solution {  
public:  
 string longestPalindrome(string s) {  
 if(s.size() == 0)return "";  
 int maxlen = 1, start = 0;  
 for(int i = 0; i < s.size(); ++i){  
 int len1 = expand(s, i, i);  
 int len2 = expand(s, i, i + 1);  
 int len = max(len1, len2);  
 if(len > maxlen){  
 start = i - (len - 1) / 2;  
 maxlen = len;  
 }  
 }  
 return s.substr(start, maxlen);  
 }  
 private : int expand(const string &s, int l, int r){  
 while(l >= 0 && r < s.size() && s[l] == s[r]){  
 l--;  
 r++;  
 }  
 return r - l - 1;  
 }  
};

==Note:==

在expand()函数中，string传引用和传形参两种方式在时间和空间上的消耗有显著差别。总体来说，中心扩展法的性能优于动态规划

**解法4**

Manacher（马拉车）

string longestPalindrome(string s) {  
 if(s.size() == 0)return "";  
 string temps = "$#";  
 for(int i = 0; i < s.size(); ++i){  
 temps += s[i];  
 temps += '#';  
 }  
 int len = temps.size();  
 int rl[len];  
 int id = 0, mx = 0;  
 int start = 0, maxlen = 0;  
 for(int i = 1; i < len; ++i){  
 if(i < mx){  
 rl[i] = min(rl[2 \* id - i], mx - i);  
 }else{  
 rl[i] = 1;  
 }  
 while(i - rl[i] >= 0 && i + rl[i] < len && temps[i - rl[i]] == temps[i + rl[i]])rl[i]++;  
 if(rl[i] + i > mx){  
 id = i;  
 mx = i + rl[i];  
 }  
 if(rl[i] > maxlen){  
 start = (i - rl[i]) / 2;  
 maxlen = rl[i] - 1;  
 }  
 }  
 return s.substr(start, maxlen);  
}

性能最好的算法