<https://leetcode.com/problems/longest-palindromic-substring/>

Given a string s, return *the longest* *palindromic substring*ins.

**Example 1:**

Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer.

**Example 2:**

Input: s = "cbbd"

Output: "bb"

**Constraints:**

1 <= s.length <= 1000

s consist of only digits and English letters.

**Attempt 1: 2023-06-27**

**Solution 1:  Brute Force (10 min, TLE 93/141)**

**Style 1: i is start index, j is (real end index but plus 1) to get substring need s.substring(i, j)**

class Solution {

    public String longestPalindrome(String s) {

        int len = s.length();

        int maxLen = 0;

        int maxStart = 0;

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

            for(int j = i + 1; j <= len; j++) {

                if(isPalindrome(s, i, j - 1)) {

                    if(j - i > maxLen) {

                        maxLen = j - i;

                        maxStart = i;

                    }

                }

            }

        }

        return s.substring(maxStart, maxStart + maxLen);

    }

    private boolean isPalindrome(String s, int i, int j) {

        while(i < j) {

            if(s.charAt(i) != s.charAt(j)) {

                return false;

            }

            i++;

            j--;

        }

        return true;

    }

}

Time Complexity : O(N^3), Here three nested loop creates the time complexity. Where N is the size of the string(s).

Space Complexity : O(1), Constant space.

Solved using string(Three Nested Loop). Brute Force Approach.

Note : this may give TLE.

**Style 2: i is start index, j is (real end index but plus 1) to get substring need s.substring(i, j)**

class Solution {

    public String longestPalindrome(String s) {

        int len = s.length();

        int maxLen = 0;

        int maxStart = 0;

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

            for(int j = i; j < len; j++) {

                if(isPalindrome(s, i, j)) {

                    if(j - i + 1 > maxLen) {

                        maxLen = j - i + 1;

                        maxStart = i;

                    }

                }

            }

        }

        return s.substring(maxStart, maxStart + maxLen);

    }

    private boolean isPalindrome(String s, int i, int j) {

        while(i < j) {

            if(s.charAt(i) != s.charAt(j)) {

                return false;

            }

            i++;

            j--;

        }

        return true;

    }

}

Time Complexity : O(N^3), Here three nested loop creates the time complexity. Where N is the size of the string(s).

Space Complexity : O(1), Constant space.

Solved using string(Three Nested Loop). Brute Force Approach.

Note : this may give TLE.

**Refer to**

<https://leetcode.com/problems/longest-palindromic-substring/solutions/3202985/best-c-3-solution-dp-string-brute-force-optimize-one-stop-solution/>

**Solution 2:  Spread From Center (10 min)**

class Solution {

    public String longestPalindrome(String s) {

        int len = s.length();

        int maxLen = 0;

        String result = "";

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

            String s1 = expand(s, i, i);

            String s2 = expand(s, i, i + 1);

            if(s1.length() > result.length()) {

                result = s1;

            }

            if(s2.length() > result.length()) {

                result = s2;

            }

        }

        return result;

    }

    private String expand(String s, int i, int j) {

        while(i >= 0 && j < s.length()) {

            if(s.charAt(i) != s.charAt(j)) {

                break;

            }

            i--;

            j++;

        }

        return s.substring(i + 1, j);

    }

}

Complexity Analysis

Given n as the length of s,

Time complexity: O(n^2)

There are 2n−1=O(n) centers. For each center, we call expand, which costs up to O(n).

Although the time complexity is the same as in the DP approach, the average/practical runtime of the algorithm is much faster. This is because most centers will not produce long palindromes, so most of the O(n) calls to expand will cost far less than n iterations.

The worst case scenario is when every character in the string is the same.

Space complexity: O(1)

We don't use any extra space other than a few integers. This is a big improvement on the DP approach.

**Refer to**

<https://leetcode.com/problems/longest-palindromic-substring/editorial/>

**Solution 3:  DP (30 min)**

**Style 1:**

class Solution {

    public String longestPalindrome(String s) {

        int len = s.length();

        boolean[][] dp = new boolean[len][len];

        int maxLen = 0;

        int maxStart = 0;

        for(int i = len - 1; i >= 0; i--) {

            for(int j = i; j < len; j++) {

                // 1.j - i == 0, only a character is a palindrome,

                // 2.j - i == 1 and s.charAt(i) == s.charAt(j), ij is a palindrome,

                // 3.j - i == 2 and s.charAt(i) == s.charAt(j), no matter what between i and j, i#j is a palindrome

                // 4.and if j - i > 2, then the internal string between i and j must be palindrome

                if(j - i <= 2) {

                    dp[i][j] = (s.charAt(i) == s.charAt(j));

                } else {

                    dp[i][j] = (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]);

                }

                if(dp[i][j]) {

                    if(j - i + 1 > maxLen) {

                        maxLen = j - i + 1;

                        maxStart = i;

                    }

                }

            }

        }

        return s.substring(maxStart, maxStart + maxLen);

    }

}

Time complexity: O(n^2)

Space complexity: O(n^2)

**Refer to**

<https://leetcode.com/problems/longest-palindromic-substring/solutions/2921/Share-my-Java-solution-using-dynamic-programming/comments/147908/>

dp(i, j) represents whether s(i ... j) can form a palindromic substring, dp(i, j) is true when s(i) equals to s(j) and s(i+1 ... j-1) is a palindromic substring. When we found a palindrome, check if it's the longest one. Time complexity O(n^2).

public class Solution {

    public static String longestPalindrome(String s) {

        int n = s.length();

        String res = null;

        int palindromeStartsAt = 0, maxLen = 0;

        boolean[][] dp = new boolean[n][n];

        // dp[i][j] indicates whether substring s starting at index i and ending at j is palindrome

        for(int i = n-1; i >= 0; i--) { // keep increasing the possible palindrome string

            for(int j = i; j < n; j++) { // find the max palindrome within this window of (i,j)

                //check if substring between (i,j) is palindrome

                dp[i][j] = (s.charAt(i) == s.charAt(j)) // chars at i and j should match

                          &&

                          ( j-i < 3  // if window is less than or equal to 3, just end chars should match

                            || dp[i+1][j-1]  ); // if window is > 3, substring (i+1, j-1) should be palindrome too

                //update max palindrome string

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

                    palindromeStartsAt = i;

                    maxLen = j-i+1;

                }

            }

        }

        return s.substring(palindromeStartsAt, palindromeStartsAt+maxLen);

    }

}

**Why we loop down i and loop up j ?**

<https://leetcode.com/problems/longest-palindromic-substring/solutions/2921/Share-my-Java-solution-using-dynamic-programming/comments/264539>

A little explanation for why the indices in the for loops are set the way they are (I was really confused for a long time):

j must be greater than or equal i at all times. Why? i is the start index of the substring, j is the end index of the substring. It makes no sense for i to be greater than j. Visualization helps me, so if you visualize the dp 2d array, think of a diagonal that cuts from top left to bottom right. We are only filling the top right half of dp.

Why are we counting down for i, but counting up for j? Each sub-problem dp[i][j] depends on dp[i+1][j-1] (dp[i+1][j-1] must be true and s.charAt(i) must equal s.charAt(j) for dp[i][j] to be true).

**What is j - i <= 2 ?**

<https://leetcode.com/problems/longest-palindromic-substring/solutions/2921/Share-my-Java-solution-using-dynamic-programming/comments/144790>

Really hard to read this dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 3 || dp[i + 1][j - 1]); j-i == 0, only a character is a palindrome, j-i == 1 and s.charAt(i) == s.charAt(j), ij is a palindrome, j-i == 2 and s.charAt(i) == s.charAt(j), no matter what between i and j, i#j is a palindrome and if j - i > 2, then the internal string between i and j must be palindrome

**Style 2:**

**Refer to**

<https://leetcode.com/problems/longest-palindromic-substring/editorial/>

class Solution {

    public static String longestPalindrome(String s) {

        int length = s.length();

        int maxLength = 1;

        int longestBegin = 0;

        boolean[][] table = new boolean[1000][1000];

        // All single character (substring length = 1) are

        // naturally palindrome

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

            table[i][i] = true;

        }

        // Be careful on boundary conditions, if missing "-1" will

        // show error as java.lang.StringIndexOutOfBoundsException:

        // String index out of range: 5 when input "babad", this

        // is because we assume current substring length is 2,

        // need to make sure s.charAt(i + 1) in boundary

        for(int i = 0; i < length - 1; i++) {

            if(s.charAt(i) == s.charAt(i + 1)) {

                table[i][i + 1] = true;

                maxLength = 2;

                longestBegin = i;

            }

        }

        // Be careful on boundary conditions, len can equal to length

        // if missing "=", error will show as e.g input "CCC", expect

        // output "CCC", error output "CC"

        for(int len = 3; len <= length; len++) {

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

                int j = i + len - 1;

                if(s.charAt(i) == s.charAt(j) && table[i + 1][j - 1]) {

                    table[i][j] = true;

                    maxLength = len;

                    longestBegin = i;

                }

            }

        }

        return s.substring(longestBegin, longestBegin + maxLength);

    }

}