Longest Palindromic Substring

Problem

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”

Solution

Approach 1: Brute Force

The obvious brute force solution is to pick all possible starting and ending positions for a substring, and verify if it is a palindrome.

Complexity Analysis

- Time complexity: $O(n^3)$. Assume that n is the length of the input string, there are a total of $\frac{n(n-1)}{2}$ such substrings (excluding the trivial solution where a character itset is a palindrome). Since verifying each substring takes $O(n)$ time, the run time complexity is $O(n^3)$.
- Space complexity: $O(1)$.

Approach 2: Dynamic Programming

We define $P(i,j)$ as following:

$$P(i,j) = \begin{cases} true, & if\ the\ substring\ S_i \dots S_j\ is\ a\ palindrome \\ false, & otherwise. \end{cases}$$

Therefore,

$$P(i,j) = (P(i+1, j-1)\ and\ S_i == S_j)$$

The base cases are:

$$P(i,i) = true$$

$$P(i,i+1) = (S_i == S_{i+1})$$

Java

```java
class Solution {
    public String longestPalindrome(String s) {
        if(s.length() <= 0)
            return "";
        int strlen = s.length();
        int left=0, right = 0, maxlen = 0;
        Boolean[][] palindromic = new Boolean[strlen][strlen];
        for(int i=0;i<strlen;i++){
            for(int j=0;j<=i;j++){
                if(i==j)
                    palindromic[i][j] = true;
                else if(i-j==1)
                    palindromic[i][j] = (s.charAt(i) == s.charAt(j));
                else
                    palindromic[i][j] = (s.charAt(i) == s.charAt(j)) && (palindromic[i-1]
[j+1]);
                if (palindromic[i][j] && maxlen<(i-j+1)){
                    maxlen = i-j+1;
                    left = j;
                    right = i+1;
                }
            }
        }
        return s.substring(left, right);
    }
}
```

Python

```python
import numpy as np

class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        if len(s) <= 0:
            return ""
        strlen = len(s)
        matrix = np.zeros((strlen, strlen))
        maxlen = 0
        left = 0
        right = 0
        for i in range(strlen):
            for j in range(i+1):
                if i == j:
                    matrix[i, j] = 1
                elif i - j == 1:
                    matrix[i, j] = s[i] == s[j]
                else:
                    matrix[i, j] = (s[i] == s[j]) and (matrix[i - 1, j + 1])
                if matrix[i, j] and maxlen < (i - j + 1):
                    maxlen = i - j + 1
                    left = j
                    right = i + 1
        return s[left:right]
```

Complexity Analysis

- Time complexity: $O(n^2)$.

- Space complexity: $O(n^2)$.

Approach 3: Expand Around Center

In fact, we could solve it in $O(n^2)$ time using only constant space.
A palindrome mirrors around its center. Therefore, a palindrome can be expanded from its center, and tere are only $2n-1$ such centers, because the center of a palindrome can be in between two letters.

Java

```
public String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";
    int start = 0, end = 0;
    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);
        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substring(start, end + 1);
}

private int expandAroundCenter(String s, int left, int right) {
    int L = left, R = right;
    while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {
        L--;
        R++;
    }
    return R - L - 1;
}
```

Python

```
class Solution:
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        if len(s) <= 0:
            return ""
        left = 0
        right = 0
        for i in range(len(s)):
            len1 = self.expandAroundCenter(s, i, i)
            len2 = self.expandAroundCenter(s, i, i+1)
            maxlen = max(len1, len2)
            if maxlen > right - left:
                left = i - (maxlen - 1) // 2
                right = i + maxlen // 2

        return s[left:right+1]

    def expandAroundCenter(self, s, left, right):
        L = left
        R = right
        while(L >= 0 and R < len(s) and s[L] == s[R]):
            L -= 1
            R += 1
        return R - L - 1
```

Complexity Analysis

- Time complexity: $O(n^2)$. Since expanding a palindrome around its center could take $O(n)$ time.
- Space complexity: $O(1)$.

Approach 4: Manacher’s Algorithm

There is even an $O(n)$ algorithm called Manacher’s algorithm, a non-trival algorithm.

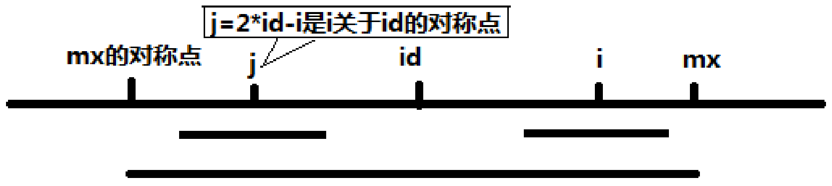
Intuition

Using an array P to store the length of palindrome centers at S_i.
First transform the input string S to another string T by inserting a special character ‘#’ in betweeen letters.
See the following example:

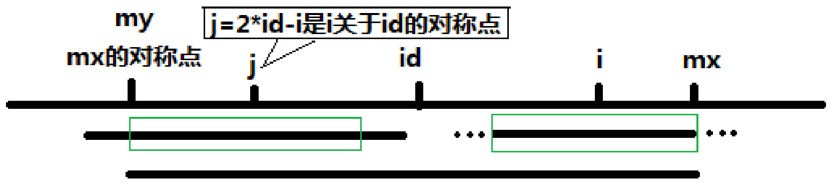
T=	#	a	#	b	#	a	#	a	#	b	#	a	#
P=	0	1	0	3	0	1	6	1	0	3	0	1	0

How to compute the value in P:

We choose two auxiliary variable id and mx, where id indicates the center of the longest palindromic substring and mx indicates $id + P[id]$
$If \quad mx - i > P[j]: \quad P[i] = P[j]$



$ELSE: \quad P[i] \geq mx - i$



$If \quad mx \leq i: \quad Let\ P[i] = 1$

Java

```
// Transform S into T.
// For example, S = "abba", T = "^#a#b#b#a#$".
```

// ^ and signs are sentinels appended to each end to avoid bounds checking string preProcess(string s) { int n = s.length(); if (n == 0) return “^”;
string ret = “^”;
for (int i = 0; i < n; i++)
ret += “#” + s.substr(i, 1);

```
  ret += "#$";
  return ret;
}

string longestPalindrome(string s) {
  string T = preProcess(s);
  int n = T.length();
  int *P = new int[n];
  int C = 0, R = 0;
  for (int i = 1; i < n-1; i++) {
    int i_mirror = 2*C-i; // equals to i' = C - (i-C)

    P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;

    // Attempt to expand palindrome centered at i
    while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
      P[i]++;

    // If palindrome centered at i expand past R,
    // adjust center based on expanded palindrome.
    if (i + P[i] > R) {
      C = i;
      R = i + P[i];
    }
  }

  // Find the maximum element in P.
  int maxLen = 0;
  int centerIndex = 0;
  for (int i = 1; i < n-1; i++) {
    if (P[i] > maxLen) {
      maxLen = P[i];
      centerIndex = i;
    }
  }
  delete[] P;

  return s.substr((centerIndex - 1 - maxLen)/2, maxLen);
}
```