LeetCode 5

https://leetcode.com/problems/longest-palindromic-substring/description/

Yifeng Zeng

Description

5. Longest Palindromic Substring

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

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

Example:
Input: "cbbd"
Output: "bb"
```

Idea Report

My primitive idea is to do a search to see if all the possible substrings is palindrom. We have a global result, and once the current string we are search is shorter than the global result, we stop searching. And we have a memo[i][j] to save the searched length of s.substring(i, j + 1). With this memorized DFS it is really slow because we use O(n) to check every possible string (isPalindrom()).

Code:

```
class Solution {
  // DFS + memo, check if whole substring is palindrom, AC, slow
  public String longestPalindrome(String s) {

          String[] res = new String[1];
          res[0] = "";
          int[][] memo = new int[s.length()][s.length()];
          helper(s, 0, s.length() - 1, res, memo);
          return res[0];
      }

      private void helper(String s, int start, int end, String[] res,
                          int[][] memo) {
          if (end - start + 1 <= res[0].length() || start > end) {
              return;
          }
          if (memo[start][end] != 0) {
              return;
          }

          if (isPalindrom(s, start, end)) {
              memo[start][end] = 1;
              res[0] = s.substring(start, end + 1);
              return;
          } else {
              memo[start][end] = -1;
          }

          helper(s, start + 1, end, res, memo);
          helper(s, start, end - 1, res, memo);
      }

      private boolean isPalindrom(String s, int i, int j) {
          while (i < j) {
              if (s.charAt(i) != s.charAt(j)) {
                  return false;
              }
              i++;
              j--;
          }
          return true;
      }
}
```

We can use a top down method to check character at i and j are the same. If they are the same, we can recursively see if i + 1 and j - 1 are the same. Also use a memo[i][j] to save the searced result.

Code:

```
class Solution {
  // DFS + memo, top down, check s[i] == s[j] ? and recursion, AC
  public String longestPalindrome(String s) {
      int len = s.length();
      // memo[i][j] is s.substring(i, j+1)
      // -1 false, 0 unvisited, >=1 length
      int[][] memo = new int[len][len];
      String[] res = new String[1];
      res[0] = "";

      // DFS
      helper(s, 0, len - 1, memo, res);
      return res[0];
  }

  private boolean helper(String s, int i, int j, int[][] memo, String[] res) {
      if (i >= j) {
          if (i == j) {
              memo[i][j] = 1;
              if (memo[i][j] > res[0].length()) {
                  res[0] = s.substring(i, j + 1);
              }
          }
          return true;
      }

      if (memo[i][j] != 0) {
          return memo[i][j] > 1;
      }

      if (s.charAt(i) == s.charAt(j) && helper(s, i + 1, j - 1, memo, res)) {
          memo[i][j] = j - i + 1;
          if (memo[i][j] > res[0].length()) {
              res[0] = s.substring(i, j + 1);
          }
          return true;
      }

      memo[i][j] = -1;
      helper(s, i + 1, j, memo, res);
      helper(s, i, j - 1, memo, res);

      return false;
  }
}
```

Since we used top down, we can also try bottom up, which becomes a DP problem. f[i][j] means the length of the substring(i, j + 1). If it is -1, it is not a palindrom, if it is 0, it means we have not searched yet. If it is > 0, it means the palindrom substring length. This method is searching from the end of the string to begining.

Code:

```
class Solution {
    // DP bottom up, AC
    public String longestPalindrome(String s) {
        int len = s.length();
        // -1 false, 0 unvisited, >=1 length
        int[][] f = new int[len + 1][len + 1];
        for (int i = 0; i <= len; i++) {
            f[i][i] = 1;
        }

        String res = "";
        for (int i = len - 1; i >= 0; i--) {
            for (int j = i; j < len; j++) {
                if (s.charAt(i) == s.charAt(j)
                    && ((j - i <= 2) || (f[i + 1][j - 1]) > 0)) {
                    f[i][j] = j - i + 1;
                    if (f[i][j] > res.length()) {
                        res = s.substring(i, j + 1);
                    }
                }
            }
        }
        return res;
    }
}
```

We can also search from the begining to the end of the string in the above DP method.

Code:

```
class Solution {
  // DP bottom up, AC
  public String longestPalindrome(String s) {
      int len = s.length();
      // -1 false, 0 unvisited, >=1 length
      int[][] f = new int[len + 1][len + 1];
      String res = "";
      for (int i = 0; i < len; i++) {
          for (int j = 0; j <= i; j++) {
              if (s.charAt(j) == s.charAt(i)
                  && (i - 1 - (j + 1) + 1 <= 1 || f[j + 1][i - 1] > 0)) {
                  f[j][i] = i - j + 1;
                  if (f[j][i] > res.length()) {
                      res = s.substring(j, i + 1);
                  }
              }
          }
      }
      return res;
  }
}
```

Summary

- Memoized DFS
- DP