161. One Edit Distance

```
class Solution(object):
    def isOneEditDistance(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """
        len_s, len_t = len(s), len(t)
        count = 0
        if len_s == len_t:
            for i in range(len_s):
                if s[i] != t[i]:
                    count += 1
                    if count > 1:
                        return False
            return count == 1
        elif len_s + 1 == len_t:
            i = 0
            for j in range(len_s):
                if s[i] != t[j]:
                    count += 1
                    if count > 1:
                        return False
                else:
                    i += 1
            return True if i == len_s or s[i] == t[len_s] else False
        elif len_s == len_t + 1:
            j = 0
            for i in range(len_t):
                if s[i] != t[j]:
                    count += 1
                    if count > 1:
                        return False
                else:
                    j += 1
            return True if j == len_t or s[len_t] == t[j] else False
        return False
```

72. Edit Distance

Dp, O(n**2)

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
```

```
        len_1, len_2 = len(word1), len(word2)
        memo = [[0 for j in range(len_2 + 1)] for i in range(len_1 + 1)]
        for i in range(1, len_1 + 1):
            memo[i][0] = i
        for j in range(1, len_2 + 1):
            memo[0][j] = j
        for i in range(len_1):
            for j in range(len_2):
                if word1[i] == word2[j]:
                    memo[i + 1][j + 1] = memo[i][j]
                else:
                    memo[i + 1][j + 1] = min(memo[i][j + 1], memo[i + 1][j], memo[i][j]) + 1
        return memo[-1][-1]
```

71. Simplify Path

```
class Solution:
    def simplifyPath(self, path: str) -> str:
        paths = path.split('/')
        result = []
        for p in paths:
            if not p or p == '.':
                continue
            elif p == '..':
                if result:
                    result.pop()
            else:
                result.append(p)
        return '/' + '/'.join(result)
```

208. Implement Trie (Prefix Tree)

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = {}

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
```

```
        """
        node = self.trie
        for c in word:
            if c not in node:
                node[c] = {}
            node = node[c]
        node['#'] = True

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        node = self.trie
        for c in word:
            if c not in node:
                return False
            node = node[c]
        return '#' in node

    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given prefix.
        """
        node = self.trie
        for c in prefix:
            if c not in node:
                return False
            node = node[c]
        return True
```

Rating bar

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="./index.css"/>
    <title>Document</title>
</head>
<body>
```

```
    <div class="rating">

<span>☆</span><span>☆</span><span>☆</span><span>☆</span><span>☆</span>
    </div>
</body>
</html>
span:hover:before,
span:hover ~ span:before {
    content: "\2605";
    position: absolute;
}

.rating {
    direction: rtl;
}
```