题目地址(821. 字符的最短距离)

https://leetcode-cn.com/problems/shortest-distance-to-a-character

入选理由

1. 仍然是一道简单题，不过比昨天的题目难度增加一点

2. 虽然这是一个字符串的题目，但其实字符串和数组没有本质差别，这在讲义中也提到了。

题目描述

给定一个字符串 S 和一个字符 C。返回一个代表字符串 S 中每个字符到字符串 S 中的字符 C 的最短距离的数组。

示例 1:

```
输入: S = "loveleetcode", C = 'e'
输出: [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]
说明:

- 字符串 S 的长度范围为 [1, 10000]。
- C 是一个单字符，且保证是字符串 S 里的字符。
- S 和 C 中的所有字母均为小写字母。
```

标签

- 字符串

难度

- 简单

前置知识

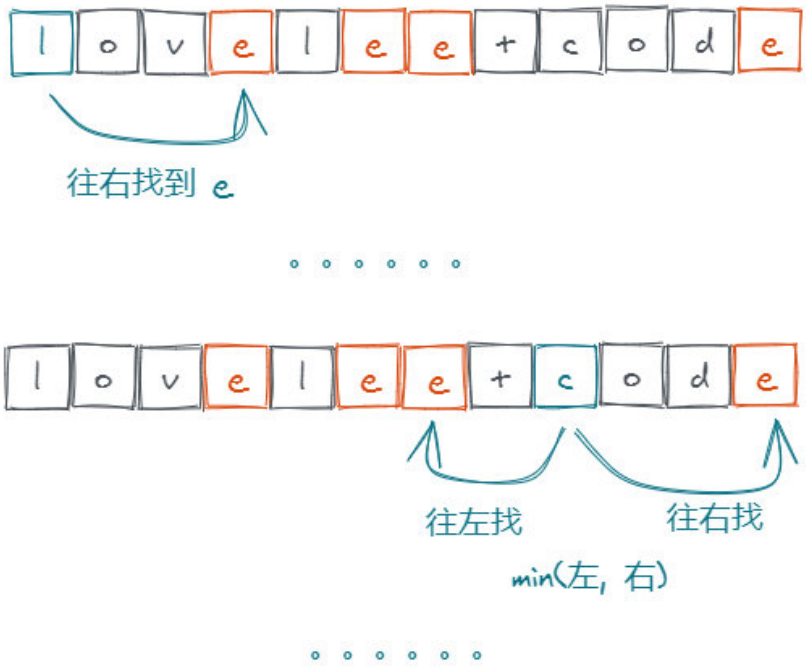
- 数组的遍历(正向遍历和反向遍历)

解法 1：中心扩展法

思路

这是最符合直觉的思路，对每个字符分别进行如下处理：

- 从当前下标出发，分别向左、右两个方向去寻找目标字符 `C` 。
- 只在一个方向找到的话，直接计算字符距离。
- 两个方向都找到的话，取两个距离的最小值。



复杂度分析

- 时间复杂度：$O(N^2)$，N 为 S 的长度，两层循环。
- 空间复杂度：$O(1)$。

代码 (JS/C++)

JavaScript Code

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
  // 结果数组 res
  var res = Array(S.length).fill(0);
```

```
  for (let i = 0; i < S.length; i++) {
    // 如果当前是目标字符，就什么都不用做
    if (S[i] === C) continue;

    // 定义两个指针 l, r 分别向左、右两个方向寻找目标字符 C，取最短距离
    let l = i,
      r = i,
      shortest = Infinity;

    while (l >= 0) {
      if (S[l] === C) {
        shortest = Math.min(shortest, i - l);
        break;
      }
      l--;
    }

    while (r < S.length) {
      if (S[r] === C) {
        shortest = Math.min(shortest, r - i);
        break;
      }
      r++;
    }

    res[i] = shortest;
  }
  return res;
};
```

C++ Code

```
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        vector<int> res(S.length());

        for (int i = 0; i < S.length(); i++) {
            if (S[i] == C) continue;

            int left = i;
            int right = i;
            int dist = 0;

            while (left >= 0 || right <= S.length() - 1) {
                if (S[left] == C) {
                    dist = i - left;
                    break;
                }
```

```
                if (S[right] == C) {
                    dist = right - i;
                    break;
                }

                if (left > 0) left--;
                if (right < S.length() - 1) right++;
            }

            res[i] = dist;
        }

        return res;
    }
};
```

解法 2：空间换时间

思路

空间换时间是编程中很常见的一种 trade-off (反过来，时间换空间也是)。

因为目标字符 `C` 在 `S` 中的位置是不变的，所以我们可以提前将 `C` 的所有下标记录在一个数组 `cIndices` 中。

然后遍历字符串 `S` 中的每个字符，到 `cIndices` 中找到距离当前位置最近的下标，计算距离。

复杂度分析

- 时间复杂度：$O(N * K)$，N 是 S 的长度，K 是字符 `C` 在字符串中出现的次数，$K <= N$。
- 空间复杂度：$O(K)$，K 为字符 `C` 出现的次数，这是记录字符 `C` 出现下标的辅助数组消耗的空间。

代码 (JS/C++)

JavaScript Code

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
  // 记录 C 字符在 S 字符串中出现的所有下标
  var cIndices = [];
  for (let i = 0; i < S.length; i++) {
    if (S[i] === C) cIndices.push(i);
```

```
  }

  // 结果数组 res
  var res = Array(S.length).fill(Infinity);

  for (let i = 0; i < S.length; i++) {
    // 目标字符，距离是 0
    if (S[i] === C) {
      res[i] = 0;
      continue;
    }

    // 非目标字符，到下标数组中找最近的下标
    for (const cIndex of cIndices) {
      const dist = Math.abs(cIndex - i);

      // 小小剪枝一下
      // 注：因为 cIndices 中的下标是递增的，后面的 dist 也会越来越大，可以排除
      if (dist >= res[i]) break;

      res[i] = dist;
    }
  }
  return res;
};
```

C++ Code

```
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        int n = S.length();
        vector<int> c_indices;
        // Initialize a vector of size n with default value n.
        vector<int> res(n, n);

        for (int i = 0; i < n; i++) {
            if (S[i] == C) c_indices.push_back(i);
        }

        for (int i = 0; i < n; i++) {
            if (S[i] == C) {
                res[i] = 0;
                continue;
            }

            for (int j = 0; j < c_indices.size(); j++) {
                int dist = abs(c_indices[j] - i);
                if (dist > res[i]) break;
```

```
                res[i] = dist;
            }
        }

        return res;
    }
};
```

解法 3：贪心

思路

其实对于每个字符来说，它只关心离它最近的那个 `C` 字符，其他的它都不管。所以这里还可以用贪心的思路：

1. 先 从左往右 遍历字符串 `S`，用一个数组 left 记录每个字符 左侧 出现的最后一个 `C` 字符的下标；
2. 再 从右往左 遍历字符串 `S`，用一个数组 right 记录每个字符 右侧 出现的最后一个 `C` 字符的下标；
3. 然后同时遍历这两个数组，计算距离最小值。

优化 1

再多想一步，其实第二个数组并不需要。因为对于左右两侧的 `C` 字符，我们也只关心其中距离更近的那一个，所以第二次遍历的时候可以看情况覆盖掉第一个数组的值：

1. 字符左侧没有出现过 `C` 字符
2. `i - left > right - i` (i 为当前字符下标，left 为字符左侧最近的 `C` 下标，right 为字符右侧最近的 `C` 下标)

如果出现以上两种情况，就可以进行覆盖，最后再遍历一次数组计算距离。

优化 2

如果我们是直接记录 `C` 与当前字符的距离，而不是记录 `C` 的下标，还可以省掉最后一次遍历计算距离的过程。

复杂度分析

- 时间复杂度：$O(N)$，N 是 S 的长度。
- 空间复杂度：$O(1)$。

代码 (JS/C++/Python)

JavaScript Code

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
  var res = Array(S.length);

  // 第一次遍历：从左往右
  // 找到出现在左侧的 C 字符的最后下标
  for (let i = 0; i < S.length; i++) {
    if (S[i] === C) res[i] = i;
    // 如果左侧没有出现 C 字符的话，用 Infinity 进行标记
    else res[i] = res[i - 1] === void 0 ? Infinity : res[i - 1];
  }

  // 第二次遍历：从右往左
  // 找出现在右侧的 C 字符的最后下标
  // 如果左侧没有出现过 C 字符，或者右侧出现的 C 字符距离更近，就更新 res[i]
  for (let i = S.length - 1; i >= 0; i--) {
    if (res[i] === Infinity || res[i + 1] - i < i - res[i]) res[i] = res[i + 1];
  }

  // 计算距离
  for (let i = 0; i < res.length; i++) {
    res[i] = Math.abs(res[i] - i);
  }
  return res;
};
```

直接计算距离：

JavaScript Code

```
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
  var res = Array(S.length);

  for (let i = 0; i < S.length; i++) {
    if (S[i] === C) res[i] = 0;
    // 记录距离：res[i - 1] + 1
    else res[i] = res[i - 1] === void 0 ? Infinity : res[i - 1] + 1;
  }
```

```
  for (let i = S.length - 1; i >= 0; i--) {
    // 更新距离：res[i + 1] + 1
    if (res[i] === Infinity || res[i + 1] + 1 < res[i]) res[i] = res[i + 1] + 1;
  }

  return res;
};
```

C++ Code

```
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        int n = S.length();
        vector<int> dist(n, n);

        for (int i = 0; i < n; i++) {
            if (S[i] == C) dist[i] = 0;
            else if (i > 0) dist[i] = dist[i - 1] + 1;
        }

        for (int i = n - 1; i >= 0; i--) {
            if (dist[i] == n
                || (i < n - 1 && dist[i + 1] + 1 < dist[i]))
                    dist[i] = dist[i + 1] + 1;
        }

        return dist;
    }
};
```

Python Code

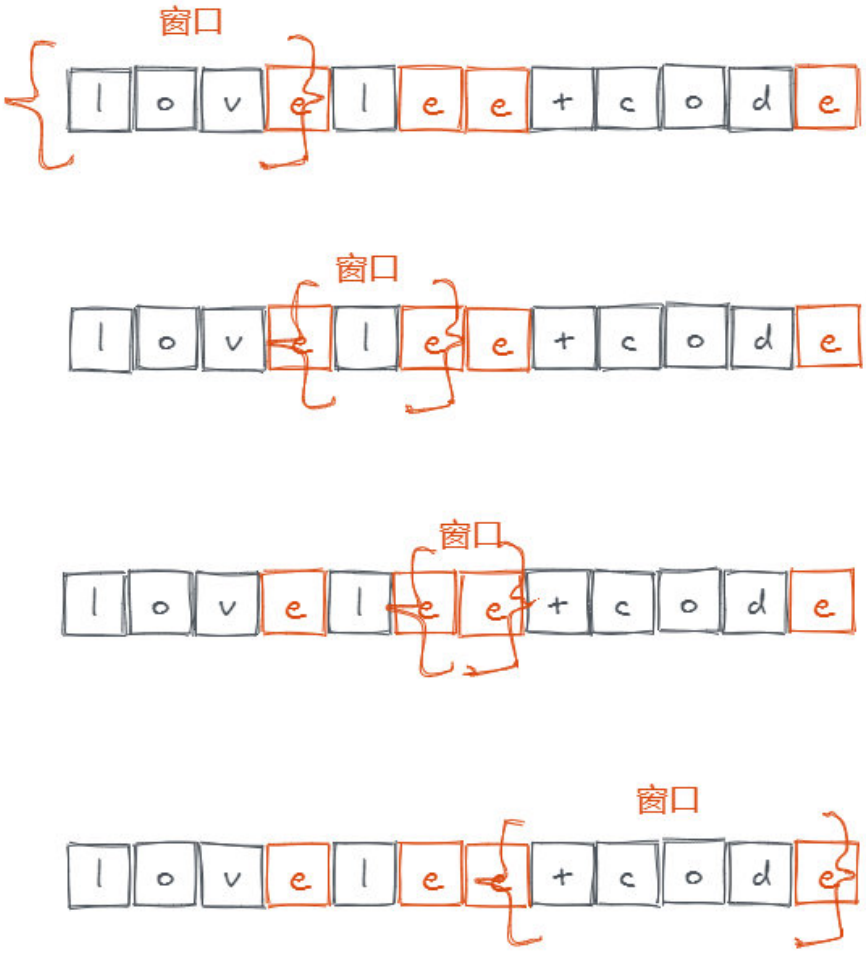
```
class Solution(object):
    def shortestToChar(self, s, c):
        """
        :type s: str
        :type c: str
        :rtype: List[int]
        """
        n = len(s)
        res = [0 if s[i] == c else None for i in range(n)]

        for i in range(1, n):
            if res[i] != 0 and res[i - 1] is not None:
                res[i] = res[i - 1] + 1
```

```
        for i in range(n - 2, -1, -1):
            if res[i] is None or res[i + 1] + 1 < res[i]:
                res[i] = res[i + 1] + 1
        return res
```

解法 4：窗口

思路

把 `C` 看成分界线，将 `S` 划分成一个个窗口。然后对每个窗口进行遍历，分别计算每个字符到窗口边界的距离最小值。



复杂度分析

- 时间复杂度：$O(N)$，N 是 S 的长度。
- 空间复杂度：$O(1)$。

代码 (JS/C++/Python)

JavaScript Code

```javascript
/**
 * @param {string} S
 * @param {character} C
 * @return {number[]}
 */
var shortestToChar = function (S, C) {
  // 窗口左边界，如果没有就初始化为 Infinity，初始化为 S.length 也可以
  let l = S[0] === C ? 0 : Infinity,
    // 窗口右边界
    r = S.indexOf(C, 1);

  const res = Array(S.length);

  for (let i = 0; i < S.length; i++) {
    // 计算字符到当前窗口左右边界的最小距离
    res[i] = Math.min(Math.abs(i - l), Math.abs(r - i));

    // 遍历完了当前窗口的字符后，将整个窗口右移
    if (i === r) {
      l = r;
      r = S.indexOf(C, l + 1);
    }
  }

  return res;
};
```

C++ Code

```cpp
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        int n = S.length();

        int l = S[0] == C ? 0 : n;
        int r = S.find(C, 1);

        vector<int> dist(n);

        for (int i = 0; i < n; i++) {
            dist[i] = min(abs(i - l), abs(r - i));
            if (i == r) {
                l = r;
                r = S.find(C, r + 1);
            }
```

```
        }

        return dist;
    }
};
```

Python Code

```
class Solution(object):
    def shortestToChar(self, s, c):
        """
        :type s: str
        :type c: str
        :rtype: List[int]
        """
        n = len(s)
        res = [0 for _ in range(n)]

        l = 0 if s[0] == c else n
        r = s.find(c, 1)

        for i in range(n):
            res[i] = min(abs(i - l), abs(r - i))
            if i == r:
                l = r
                r = s.find(c, l + 1)
        return res
```

上一页 下一页