821. Shortest Distance to a Character
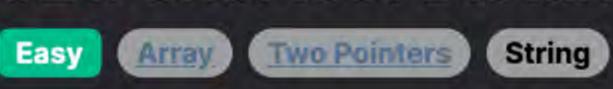
`Easy` `Array` `Two Pointers` `String`

821. Shortest Distance to a Character

Problem Description

In this problem, we are given a string s and a character c, with the requirement that c occurs at least once in s. We need to return an array of integers answer such that for every index i in the string s, answer[i] represents the shortest distance from the character at s[i] to the nearest occurrence of the character c. The distance between two indices is defined as the absolute difference between those indices.

Intuition

The intuition behind the solution is to iterate through the string s twice to find the distance to the nearest occurrence of character c. On the first pass, we move from left to right, keeping track of the most recent position of c (using a variable pre initialized with a value smaller than any valid index). For every character in s, we update ans[i] to be the distance from the current index i to the closest occurrence of c encountered so far.

The key point to notice is that this only accounts for the nearest occurrence of c that is either at the current index i or to its left since we have moved from left to right.

To find the overall nearest occurrence, we need to also check for the closest c to the right of our current position. Hence, we do a second pass from right to left. This time, we use a variable suf initialized with a value larger than any valid index to track the latest occurrence of c from the right. As we move leftward, we update ans[i] with the minimum of its current value and the distance to the nearest c from the right.

By comparing ans[i] obtained from the leftward and rightward passes for each index i, we ensure that the closest c in either direction is considered. This allows us to compute the shortest distance for each character in s to the nearest c.

By using two passes, we make the algorithm efficient with linear time complexity, since each pass processes every element exactly once.

Solution Approach

The solution approach involves a two-pass algorithm to compute the minimum distance to the closest occurrence of the character c for each index in the string s. Here is the step-by-step implementation explained:

1. **Initialize the data structures and variables**: We initialize an array ans with the same length as string s, filled with n, the length of s. This ensures that at the beginning, the minimum distances are set to the largest possible value. We also initialize two variables, pre and suf, to -inf and inf respectively. These will hold the indices of the most recent occurrence of c from the left pass and the right pass, ensuring that they are initially set to invalid positions outside the range of valid indices.

2. **First pass (left to right)**: We loop through s from left to right using i to track the current index. If we find an occurrence of c, we update pre to the current index i. For each index i, we compute the distance from i to pre and update ans[i] to be the minimum of its current value and i - pre.

   This leftward pass sets ans[i] to the correct value if the nearest c is to the left or at index i.

3. **Second pass (right to left)**: We perform a backward loop through s from the last index to the first, again tracking the current index with i. If s[i] is c, we update suf to i. Then for each index i, we calculate the distance to suf, and we update ans[i] to be the minimum of its current value and suf - i.

   This rightward pass corrects ans[i] if the nearest occurrence of c is to the right of i.

By combining the results of both passes, ans[i] ends up with the minimum distance to the closest c from either direction. Since we are moving in both directions once, our time complexity remains O(n), where n is the length of s.

4. **Return the result**: Finally, after both passes are completed, we return the array ans which contains the minimum distance to the closest c for each index in the string s.

In terms of data structures, we only use a list to store the result. The algorithm heavily relies on the two-pointer technique, where the pointers (pre and suf) move through the string in opposite directions, updating the distances based on the latest seen occurrences of the target character c.

Example Walkthrough

Let's consider an example where the string s is "leetcode" and the character c is 'e'.

Step 1: Initialize the data structures and variables

We start by creating the ans array of the same length as s, which is 8 in this case. The ans array is initially filled with the largest possible minimum distances, which would be 8 (the length of s).

```
1   ans = [8, 8, 8, 8, 8, 8, 8, 8]
```

We'll have two variables pre = -inf and suf = inf to track the most recent occurrence of c from both ends.

Step 2: First pass (left to right)

1. Start from index i = 0, since s[0] != 'e', we skip updating pre and ans[1] = max(8, i - (-inf)), which remains 8.
2. Move to index i = 1, s[1] != 'e', we update pre = 1, and ans[1] = min(8, 1 - 1) = 0.
3. Continue to index i = 2, s[2] != 'e', so ans[2] = min(8, 2 - 1) = 1.
4. Repeat until the end of s, creating the array after the left to right pass:

```
1   ans = [8, 0, 1, 2, 8, 1, 2, 3]
```

Step 3: Second pass (right to left)

1. Start from the last index i = 7, s[7] != 'e', and update ans[7] = min(3, inf - 7) = 3.
2. Move to index i = 6, s[6] != 'e', and ans[6] = min(2, 7 - 6) = 1.
3. Proceed to index i = 4, s[4] = 'e', we update suf = 4, and ans[4] = min(0, 4 - 4) = 0.
4. Continue moving left and updating ans until i = 0, resulting in:

```
1   ans = [2, 0, 1, 0, 0, 1, 1, 3]
```

Now, ans[i] represents the shortest distance to the nearest e for every character.

Step 4: Return the result

Finally, we return the updated ans array:

```
1   ans = [2, 0, 1, 0, 0, 1, 1, 3]
```

This ans array provides the shortest distance from each character in s to the nearest occurrence of c, as desired. Each element in ans is computed by considering the closest occurrence of c from both directions in the string. The solution is efficient with a linear time complexity due to the two-pass algorithm.

Python Solution

```python
1   class Solution:
2       def shortestToChar(self, s: str, c: str) -> List[int]:
3           # Find the length of the input string
4           string_length = len(s)
5
6           # Initialize the answer list with a default high value
7           answer = [string_length] * string_length
8
9           # Initialize the previous occurrence index of character 'c' to minus infinity
10          previous_occurrence = float('-inf')
11
12          # Forward pass: Find the distance to the closest occurrence of 'c' to the left
13          for index, character in enumerate(s):
14              if character == c:
15                  # Update the previous occurrence index when we find 'c'
16                  previous_occurrence = index
17              # Update the answer list with the minimum distance so far
18              answer[index] = min(answer[index], index - previous_occurrence)
19
20          # Initialize the next occurrence index of character 'c' to infinity
21          next_occurrence = float('inf')
22
23          # Backward pass: Find the distance to the closest occurrence of 'c' to the right
24          for index in range(string_length - 1, -1, -1):
25              if s[index] == c:
26                  # Update the next occurrence index when we find 'c'
27                  next_occurrence = index
28              # Update the answer list with the minimum distance from either direction
29              answer[index] = min(answer[index], next_occurrence - index)
30
31          # Return the populated list of minimum distances to 'c'
32          return answer
33
```

Java Solution

```java
1   class Solution {
2       public int[] shortestToChar(String s, char targetChar) {
3           // Get the length of the string to create and fill the output array
4           int strLength = s.length();
5           int[] shortestDistances = new int[strLength];
6
7           // The variable 'inf' represents an effective infinity for our purposes
8           final int inf = 1 << 30; // 2^30 is much greater than the maximum possible string length
9           Arrays.fill(shortestDistances, inf); // Fill the array with 'infinite' distance initially
10
11          // First pass: move from left to right,
12          // updating the closest target character seen so far
13          for (int i = 0, closestLeft = -inf; i < strLength; ++i) {
14              // If the current position is of the closest target character if found
15              if (s.charAt(i) == targetChar) {
16                  closestLeft = i;
17              }
18              // Update the shortest distance for position i
19              shortestDistances[i] = i - closestLeft;
20          }
21
22          // Second pass: move from right to left,
23          // updating the closest target character seen so far
24          for (int i = strLength - 1, closestRight = inf; i >= 0; --i) {
25              // Update the position of the closest target character if found
26              if (s.charAt(i) == targetChar) {
27                  closestRight = i;
28              }
29              // Update the shortest distance for position i only if it's smaller than the current value
30              shortestDistances[i] = Math.min(shortestDistances[i], closestRight - i);
31          }
32
33          // Return the array of shortest distances to the target character
34          return shortestDistances;
35      }
36  }
37
```

C++ Solution

```cpp
1   #include <vector>
2   #include <string>
3   #include <algorithm> // Include necessary headers
4
5   class Solution {
6   public:
7       std::vector<int> shortestToChar(std::string s, char c) {
8           int n = s.size(); // Get the size of the string
9           const int INF = 3 << 30; // Define an 'infinity' value for initial distances
10          std::vector<int> distances(n, INF); // Create a vector to store distances initialized with 'infinity'
11
12          // Forward pass: Find closest 'c' before the current position
13          // Initialize previous position of character 'c' to a very negative number
14          for (int i = 0, prevCPosition = -INF; i < n; ++i) {
15              if (s[i] == c) {
16                  prevCPosition = i; // Update position when we find character 'c'
17              }
18              // Calculate distance to the closest 'c' so far from the left
19              distances[i] = std::min(distances[i], i - prevCPosition);
20          }
21
22          // Backward pass: Find closest 'c' after the current position
23          // Set suffix position of character 'c' very high initially
24          for (int i = n - 1, nextCPosition = INF; i >= 0; --i) {
25              if (s[i] == c) {
26                  nextCPosition = i; // Update position when we find character 'c'
27              }
28              // Calculate distance to the closest 'c' so far from the right
29              // We use the min() function again to ensure the closest 'c' in either direction
30              distances[i] = std::min(distances[i], nextCPosition - i);
31          }
32
33          return distances; // Return the computed vector of distances
34      }
35  };
36
```

Typescript Solution

```typescript
1   function shortestToChar(s: string, c: string): number[] {
2       // Get the length of the string for iteration purposes.
3       const length = s.length;
4       // Define an 'infinite' large number for initial distances.
5       const infinity = 2 ** 30;
6       // Initialize an array to store the shortest distances to character 'c'.
7       const shortestDistances: number[] = new Array(length).fill(infinity);
8
9       // Forward pass: find distances from left-side occurrences of 'c'.
10      for (let i = 0, lastOccurrenceLeft = -infinity; i < length; ++i) {
11          if (s[i] === c) {
12              // Update last occurrence of 'c' when found.
13              lastOccurrenceLeft = i;
14          }
15          // Calculate distance from the last occurrence found on the left.
16          shortestDistances[i] = i - lastOccurrenceLeft;
17      }
18
19      // Backward pass: find and compare distances from right-side occurrences of 'c'.
20      for (let i = length - 1, lastOccurrenceRight = infinity; i >= 0; --i) {
21          if (s[i] === c) {
22              // Update last occurrence of 'c' when found.
23              lastOccurrenceRight = i;
24          }
25          // Store the minimum distance between the previous value and
26          // the distance from the last occurrence found on the right.
27          shortestDistances[i] = Math.min(shortestDistances[i], lastOccurrenceRight - i);
28      }
29
30      // Return the array containing shortest distances to character 'c'.
31      return shortestDistances;
32  }
33
```

Time and Space Complexity

The time complexity of the code is O(n), where n is the length of the string s. This is because the code processes each character in the string twice: once in the forward direction and once in the backward direction.

As for the space complexity, the space used by the program is also O(n). An additional array ans of size n is used to store the minimum distance to the character c for each position in the string.

In detail, the algorithm involves the following steps:

1. It initializes the ans array with n, where n is the size of the string s. This action itself has a space complexity of O(n).
2. It iterates through the string once from left to right (forward traversal) to calculate and update the minimum distance from each character to the previous occurrence of c. This has a time complexity of O(n) for the forward traversal.
3. It then iterates through the string again from right to left (backward traversal) to update the minimum distance based on the following occurrences of c. This backward traversal also takes O(n) time.
4. No additional data structures are used that are dependent on the size of the input, so the space complexity remains O(n).

Based on these operations, the total time complexity is O(n) (forward O(n) + backward O(n)) is still O(n)), and the total space complexity remains O(n) given the above analysis.