<https://leetcode.com/problems/longest-substring-of-one-repeating-character/description/>

You are given a **0-indexed** string s. You are also given a **0-indexed** string queryCharacters of length k and a **0-indexed** array of integer **indices** queryIndices of length k, both of which are used to describe k queries.

The ith query updates the character in s at index queryIndices[i] to the character queryCharacters[i].

Return an array lengths of length k where lengths[i] is the **length** of the **longest substring** of s consisting of **only one repeating** character **after** the ith query is performed.

**Example 1:**

**Input:** s = "babacc", queryCharacters = "bcb", queryIndices = [1,3,3]

**Output:** [3,3,4]

**Explanation:**

- 1st query updates s = "b**b**bacc". The longest substring consisting of one repeating character is "bbb" with length 3.

- 2nd query updates s = "bbb**c**cc". The longest substring consisting of one repeating character can be "bbb" or "ccc" with length 3.

- 3rd query updates s = "bbb**b**cc". The longest substring consisting of one repeating character is "bbbb" with length 4.Thus, we return [3,3,4].

**Example 2:**

**Input:** s = "abyzz", queryCharacters = "aa", queryIndices = [2,1]

**Output:** [2,3]

**Explanation:**

- 1st query updates s = "ab**a**zz". The longest substring consisting of one repeating character is "zz" with length 2.

- 2nd query updates s = "a**a**azz". The longest substring consisting of one repeating character is "aaa" with length 3.Thus, we return [2,3].

**Constraints:**

1 <= s.length <= 10^5

s consists of lowercase English letters.

k == queryCharacters.length == queryIndices.length

1 <= k <= 10^5

queryCharacters consists of lowercase English letters.

0 <= queryIndices[i] < s.length

**Attempt 1: 2025-01-20**

**Solution 1: Segment Tree (1800 min)**

class Solution {

    public int[] longestRepeating(String s, String queryCharacters, int[] queryIndices) {

        int n = queryIndices.length;

        int[] result = new int[n];

        SegmentTree segmentTree = new SegmentTree(s);

        for(int i = 0; i < n; i++) {

            segmentTree.update(0, queryIndices[i], 0, s.length() - 1, queryCharacters.charAt(i));

            result[i] = segmentTree.getMaxLen();

        }

        return result;

    }

}

class SegmentTree {

    Node[] tree;

    StringBuilder sb;

    public SegmentTree(String s) {

        sb = new StringBuilder(s);

        tree = new Node[4 \* s.length()];

        buildTree(0, 0, sb.length() - 1);

    }

    private Node merge(Node left, Node right, int lo, int mid, int hi) {

        // When merging two child nodes, the parent node's prefix and suffix are

        // determined based on the prefixes and suffixes of the children.

        int maxLen = Math.max(left.maxLen, right.maxLen);

        int prefix\_start = left.prefix\_start;

        int prefix\_end = left.prefix\_end;

        int suffix\_start = right.suffix\_start;

        int suffix\_end = right.suffix\_end;

        // Note: 'segment' equals to 'node' here

        // There is one more condition when merge left and right two segments, if

        // their connection part (left segment's last character will connect to right

        // segment's first character when merge) has the same character (For example,

        // if the last character of the left segment is the same as the first character

        // of the right segment, the maxLength might increase, and the prefix or

        // suffix might extend.), then we should consider this connection part's length,

        // and compare to original left segment's and right segment's maximum repeating

        // character's length. So we actually compare three candidate repeating

        // character's length in one merge:

        // (1) left segment's maximum repeating character's length (either its prefix or suffix)

        // (2) right segment's maximum repeating character's length (either its prefix or suffix)

        // (3) left and right segment's connection part maximum repeating character's length if exist

        if(sb.charAt(mid) == sb.charAt(mid + 1)) {

            maxLen = Math.max(maxLen, right.prefix\_end - left.suffix\_start + 1);

            // If the prefix of the left segment extends to the middle

            // (left.prefix\_end - left.prefix\_start + 1 == mid - lo + 1), then

            // the prefix of the parent node can be extended to include

            // the prefix of the right segment.

            // Note: Based on definition of 'prefix' as below:

            // The prefix of a segment refers to the longest sequence of repeating

            // characters starting from the beginning of the segment.

            // prefix\_start is the starting index of this prefix segment.

            // prefix\_end is the ending index of this prefix segment.

            // The equality: left.prefix\_end - left.prefix\_start + 1 == mid - lo + 1

            // means the whole left segment is same repeating character,

            // and because this character same as right segment's first

            // character, that's why left segment can extends to includes

            // right segment's prefix

            if(left.prefix\_end - left.prefix\_start + 1 == mid - lo + 1) {

                prefix\_end = right.prefix\_end;

            }

            // Similarly, if the suffix of the right segment extends to the middle

            // (right.suffix\_end - right.suffix\_start + 1 == hi - mid), then the suffix of

            // the parent node can be extended to include the suffix of the left segment.

            if(right.suffix\_end - right.suffix\_start + 1 == hi - mid) {

                suffix\_start = left.suffix\_start;

            }

        }

        return new Node(maxLen, prefix\_start, prefix\_end, suffix\_start, suffix\_end);

    }

    private void buildTree(int treeIndex, int lo, int hi) {

        if(lo == hi) {

            // For each character in the string, the leaf nodes will have prefix\_start,

            // prefix\_end, suffix\_start, and suffix\_end set to the index of that character.

            tree[treeIndex] = new Node(1, lo, lo, hi, hi);

            return;

        }

        int mid = lo + (hi - lo) / 2;

        buildTree(2 \* treeIndex + 1, lo, mid);

        buildTree(2 \* treeIndex + 2, mid + 1, hi);

        tree[treeIndex] = merge(tree[2 \* treeIndex + 1], tree[2 \* treeIndex + 2], lo, mid, hi);

    }

    public void update(int treeIndex, int index, int lo, int hi, char c) {

        if(lo == hi) {

            sb.setCharAt(index, c);

            tree[treeIndex] = new Node(1, lo, lo, hi, hi);

            return;

        }

        int mid = lo + (hi - lo) / 2;

        if(index > mid) {

            update(2 \* treeIndex + 2, index, mid + 1, hi, c);

        } else {

            update(2 \* treeIndex + 1, index, lo, mid, c);

        }

        tree[treeIndex] = merge(tree[2 \* treeIndex + 1], tree[2 \* treeIndex + 2], lo, mid, hi);

    }

    public int getMaxLen() {

        return tree[0].maxLen;

    }

}

class Node {

    int maxLen;

    // The prefix of a segment refers to the longest sequence of repeating

    // characters starting from the beginning of the segment.

    // prefix\_start is the starting index of this prefix segment.

    // prefix\_end is the ending index of this prefix segment.

    int prefix\_start;

    int prefix\_end;

    // The suffix of a segment refers to the longest sequence of repeating

    // characters ending at the end of the segment.

    // suffix\_start is the starting index of this suffix segment.

    // suffix\_end is the ending index of this suffix segment.

    int suffix\_start;

    int suffix\_end;

    public Node(int maxLen, int prefix\_start, int prefix\_end, int suffix\_start, int suffix\_end) {

        this.maxLen = maxLen;

        this.prefix\_start = prefix\_start;

        this.prefix\_end = prefix\_end;

        this.suffix\_start = suffix\_start;

        this.suffix\_end = suffix\_end;

    }

}

Time Complexity: (O(n + m log n))

Space Complexity: (O(n + m))

Let's analyze the time complexity and space complexity of the provided solution for the problem. The solution uses a Segment Tree to efficiently handle updates and queries.

**Time Complexity**

**1. Building the Segment Tree:**

The buildTree function is called recursively to construct the Segment Tree.

At each level of the tree, the work done is proportional to the number of nodes at that level.

Since the Segment Tree is a complete binary tree, the number of levels is (O(\log n)), where (n) is the length of the string (s).

The total number of nodes in the tree is (O(n)), so the time complexity of building the tree is:

**2. Updating the Segment Tree:**

The update function is called for each query to update a character in the string.

Each update traverses the tree from the root to a leaf, which takes (O(\log n)) time.

After updating the leaf, the merge function is called at each level of the tree to update the internal nodes.

The merge function itself takes (O(1)) time because it performs a constant number of operations.

Therefore, the time complexity for each update is:

**3. Querying the Maximum Length:**

The getMaxLen function simply returns the maxLen value stored in the root of the Segment Tree.

This operation takes (O(1)) time.

**4. Overall Time Complexity:**

Let (n) be the length of the string (s), and (m) be the number of queries.

Building the Segment Tree takes (O(n)).

Each update takes (O(\log n)), and there are (m) updates, so the total time for updates is:

Querying the maximum length takes (O(1)) per query, so the total time for queries is:

Therefore, the overall time complexity is:

**Space Complexity**

**1. Segment Tree Storage:**

The Segment Tree is stored in an array of size (4n), where (n) is the length of the string (s).

This is because a Segment Tree for an array of size (n) requires (4n) space in the worst case to handle all possible segments.

Therefore, the space complexity for the Segment Tree is:

**2. String Storage:**

The StringBuilder sb is used to store the string (s), which takes (O(n)) space.

**3. Result Array:**

The result array stores the output for each query, which takes (O(m)) space.

**4. Overall Space Complexity:**

The total space complexity is dominated by the Segment Tree and the StringBuilder:

**Summary of Complexity**

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space Complexity |
| Building Segment Tree | (O(n)) | (O(n)) |
| Updating Segment Tree | (O(m log n)) | (O(n)) |
| Querying Maximum Length | (O(m)) | (O(m)) |
| **Overall** | (O(n + m log n)) | (O(n + m)) |

**Explanation of Key Points**

Why (O(n)) for Building the Segment Tree?

The Segment Tree is built recursively, and each node is processed exactly once.

Since there are (O(n)) nodes in the tree, the time complexity is (O(n)).

Why (O(log n)) for Updates?

Each update traverses the tree from the root to a leaf, which takes (O(\log n)) time.

The merge function is called at each level of the tree, but it takes (O(1)) time per call.

Why (O(n)) Space for the Segment Tree?

The Segment Tree is stored in an array of size (4n) to handle all possible segments.

Why (O(m)) Space for the Result Array?

The result array stores the output for each of the (m) queries.

**Final Answer**

Time Complexity: (O(n + m log n))

Space Complexity: (O(n + m))

This solution is efficient and scales well for large inputs, making it suitable for competitive programming and real-world applications. Let me know if you need further clarification! 😊

**Refer to**

<https://leetcode.com/problems/longest-substring-of-one-repeating-character/solutions/1866110/java-clean-segment-tree-solution/>

**Add comments by chatGPT**

class Node {

int maxLength; // Maximum length of repeating characters in the segment

int prefStart, prefEnd; // Start and end indices of the prefix segment (continuous repeating characters from the start)

int suffStart, suffEnd; // Start and end indices of the suffix segment (continuous repeating characters till the end)

Node(int maxLength, int prefStart, int prefEnd, int suffStart, int suffEnd) {

this.maxLength = maxLength;

this.prefStart = prefStart;

this.prefEnd = prefEnd;

this.suffStart = suffStart;

this.suffEnd = suffEnd;

}

}

class SegmentTree {

private Node[] tree; // Segment tree nodes

private StringBuilder s; // Input string represented as a mutable StringBuilder

public SegmentTree(String s) {

this.s = new StringBuilder(s); // Initialize with input string

tree = new Node[4 \* s.length()]; // Allocate sufficient space for the segment tree

build(0, 0, s.length() - 1); // Build the segment tree

}

// Merges two child nodes into a parent node

private Node merge(Node left, Node right, int tl, int tm, int tr) {

// Start with the maximum of the left and right nodes

int maxLength = Math.max(left.maxLength, right.maxLength);

// The prefix of the parent node starts from the prefix of the left node

int prefStart = left.prefStart;

int prefEnd = left.prefEnd;

// The suffix of the parent node ends with the suffix of the right node

int suffStart = right.suffStart;

int suffEnd = right.suffEnd;

// Check if the middle character (tm) connects the left and right segments

if (s.charAt(tm) == s.charAt(tm + 1)) {

// Update maxLength to include the combined segment if connected

maxLength = Math.max(maxLength, right.prefEnd - left.suffStart + 1);

// If the left prefix extends to the middle, merge with the right prefix

if (left.prefEnd - left.prefStart + 1 == tm - tl + 1) {

prefEnd = right.prefEnd;

}

// If the right suffix extends to the middle, merge with the left suffix

if (right.suffEnd - right.suffStart + 1 == tr - tm) {

suffStart = left.suffStart;

}

}

return new Node(maxLength, prefStart, prefEnd, suffStart, suffEnd);

}

// Builds the segment tree recursively

private void build(int pos, int tl, int tr) {

if (tl == tr) { // Leaf node, represents a single character

tree[pos] = new Node(1, tl, tl, tr, tr);

} else {

int tm = tl + (tr - tl) / 2; // Midpoint of the current segment

// Recursively build the left and right children

build(2 \* pos + 1, tl, tm);

build(2 \* pos + 2, tm + 1, tr);

// Merge the left and right children into the current node

tree[pos] = merge(tree[2 \* pos + 1], tree[2 \* pos + 2], tl, tm, tr);

}

}

// Updates the segment tree when a character at index idx is changed to ch

public void update(int pos, int tl, int tr, int idx, char ch) {

if (tl == tr) { // Leaf node, directly update

s.setCharAt(idx, ch); // Update the string

tree[pos] = new Node(1, tl, tl, tr, tr); // Update the node with the new character

} else {

int tm = tl + (tr - tl) / 2; // Midpoint of the current segment

// Update either the left or right child based on the index

if (idx <= tm) {

update(2 \* pos + 1, tl, tm, idx, ch);

} else {

update(2 \* pos + 2, tm + 1, tr, idx, ch);

}

// Merge the updated children into the current node

tree[pos] = merge(tree[2 \* pos + 1], tree[2 \* pos + 2], tl, tm, tr);

}

}

// Returns the maximum length of repeating characters in the string

public int getMaxLength() {

return tree[0].maxLength;

}

}

class Solution {

public int[] longestRepeating(String s, String queryCharacters, int[] queryIndices) {

int n = queryIndices.length;

int[] result = new int[n];

// Initialize the segment tree with the input string

SegmentTree segmentTree = new SegmentTree(s);

// Process each query

for (int i = 0; i < n; i++) {

// Update the segment tree with the new character at the specified index

segmentTree.update(0, 0, s.length() - 1, queryIndices[i], queryCharacters.charAt(i));

// Record the maximum length of repeating characters after the update

result[i] = segmentTree.getMaxLength();

}

return result;

}

}

**Explanation of the merge Method:**

Inputs:

**left** and **right**: Represent the segment nodes for the left and right child of the current node.

**tl, tm, tr:** **Indices representing the current segment's left, middle, and right boundaries.**

Logic:

The **maxLength** is initially set as the maximum of the left and right child nodes.

The prefix (prefStart, prefEnd) and suffix (suffStart, suffEnd) are set to the left and right nodes' respective values.

**Connecting Segments:**

**If the characters at the middle of the left (tm) and right (tm + 1) nodes match:**

**The maxLength is updated to include the combined length of the overlapping segment.**

**The prefix and suffix are extended if they span across the entire left or right node.**

Output:

A new Node is created with updated properties and returned to the parent node.

This design ensures that the segment tree efficiently tracks the maximum repeating substring while handling updates dynamically.

**Refer to Deepseek**

**Q1: For Leetcode 2213.Longest Substring of One Repeating Character**

I have below correct solution, but i don't very understand the variable definition in Segment Tree defined class Node as below:

int prefStart, prefEnd;

int suffStart, suffEnd;

Can you help me to understand what is 'prefix' and 'suffix' relate to Segment Node ? and what are above variables actual meaning ? Any good example to explain how these variables work ?

A: In the context of the Segment Tree used for solving Leetcode 2213 (Longest Substring of One Repeating Character), the Node class contains several variables that are crucial for maintaining the state of the segments in the tree. Specifically, the variables prefStart, prefEnd, suffStart, and suffEnd are used to track the prefix and suffix segments of the current node. Let's break down what these variables represent and how they are used:

**Prefix and Suffix in Segment Tree**

**Prefix (prefStart, prefEnd)**:

**The prefix of a segment refers to the longest sequence of repeating characters starting from the beginning of the segment.**

**prefStart is the starting index of this prefix segment.**

**prefEnd is the ending index of this prefix segment.**

**Suffix (suffStart, suffEnd)**:

**The suffix of a segment refers to the longest sequence of repeating characters ending at the end of the segment.**

**suffStart is the starting index of this suffix segment.**

**suffEnd is the ending index of this suffix segment.**

**Example to Illustrate**

Consider the string s = "aaabbb" and let's see how the Segment Tree nodes might look:

**Leaf Nodes**:

**For each character in the string,** **the leaf nodes will have prefStart, prefEnd, suffStart, and suffEnd set to the index of that character.**

For example, for the character at index 0 ('a'), the node will be:

new Node(1, 0, 0, 0, 0);

**Internal Nodes:**

**When merging two child nodes, the parent node's prefix and suffix are determined based on the prefixes and suffixes of the children.**

For example, consider merging two segments "aaa" and "bbb":

The left segment ("aaa") has:

prefStart = 0, prefEnd = 2

suffStart = 0, suffEnd = 2

The right segment ("bbb") has:

prefStart = 3, prefEnd = 5

suffStart = 3, suffEnd = 5

The merged segment ("aaabbb") will have:

prefStart = 0, prefEnd = 2 (from the left segment)

suffStart = 3, suffEnd = 5 (from the right segment)

maxLength = 3 (since the longest repeating character sequence is either "aaa" or "bbb")

**How These Variables Work in the Code**

**Building the Segment Tree:**

When building the tree, each leaf node is initialized with the character at that index.

Internal nodes are built by merging their left and right children using the merge function.

**Merging Nodes:**

The merge function combines two child nodes into a parent node.

It checks if the characters at the boundary of the two segments are the same. If they are, it updates the maxLength and adjusts the prefix and suffix accordingly.

**For example, if the last character of the left segment is the same as the first character of the right segment, the maxLength might increase, and the prefix or suffix might extend.**

**Updating the Tree:**

When a character is updated, the tree is updated from the leaf node up to the root.

The update function ensures that all relevant nodes are updated to reflect the new character.

**Querying the Tree:**

The getMaxLength function simply returns the maxLength of the root node, which represents the longest substring of repeating characters in the entire string.

**Summary**

Prefix (prefStart, prefEnd): Tracks the longest sequence of repeating characters from the start of the segment.

Suffix (suffStart, suffEnd): Tracks the longest sequence of repeating characters to the end of the segment.

These variables help in efficiently merging segments and maintaining the correct state of the Segment Tree, allowing for quick updates and queries.

By understanding these variables and how they are used in the merge function, you can see how the Segment Tree efficiently maintains the necessary information to solve the problem.

**Q2: can you use below input to simulate from beginning to the end how the above code works ?**

String queryCharacters = "bcb";

int[] queryIndices = new int[] {1,3,3};

String s = "babacc";

Let's simulate the execution of the code step by step using the provided input:

**Input:**

s = "babacc"

queryCharacters = "bcb"

queryIndices = [1, 3, 3]

**Step 1: Initialize the Segment Tree**

The SegmentTree is initialized with the string s = "babacc". The tree is built recursively, starting from the root and dividing the string into smaller segments.

**Initial String:**

Index: 0 1 2 3 4 5

Char: b a b a c c

**Segment Tree Construction:**

**Leaf Nodes:**

Each leaf node represents a single character in the string.

For example:

Node for index 0 ('b'): new Node(1, 0, 0, 0, 0)

Node for index 1 ('a'): new Node(1, 1, 1, 1, 1)

Node for index 2 ('b'): new Node(1, 2, 2, 2, 2)

Node for index 3 ('a'): new Node(1, 3, 3, 3, 3)

Node for index 4 ('c'): new Node(1, 4, 4, 4, 4)

Node for index 5 ('c'): new Node(1, 5, 5, 5, 5)

**Internal Nodes:**

Internal nodes are built by merging their left and right children.

For example, the node representing the segment [0, 1] ("ba") will have:

prefStart = 0, prefEnd = 0 (prefix is 'b')

suffStart = 1, suffEnd = 1 (suffix is 'a')

maxLength = 1 (no repeating characters)

Similarly, the node representing the segment [2, 3] ("ba") will have:

prefStart = 2, prefEnd = 2 (prefix is 'b')

suffStart = 3, suffEnd = 3 (suffix is 'a')

maxLength = 1 (no repeating characters)

**Root Node:**

The root node represents the entire string [0, 5] ("babacc").

After merging all segments, the root node will have:

prefStart = 0, prefEnd = 0 (prefix is 'b')

suffStart = 4, suffEnd = 5 (suffix is 'c')

maxLength = 2 (longest repeating substring is "cc" at indices [4, 5])

**Step 2: Process Queries**

Now, we process each query one by one.

**Query 1: Update index 1 to 'b'**

**Update**: s[1] = 'b'

**New String**: s = "bbbacc"

**Update Leaf Node**:

The leaf node at index 1 is updated to new Node(1, 1, 1, 1, 1).

**Update Internal Nodes**:

The parent nodes are updated recursively.

For example, the node representing [0, 1] ("bb") will now have:

prefStart = 0, prefEnd = 1 (prefix is "bb")

suffStart = 0, suffEnd = 1 (suffix is "bb")

maxLength = 2 (longest repeating substring is "bb")

**Update Root Node**:

The root node now represents "bbbacc".

After merging, the root node will have:

prefStart = 0, prefEnd = 2 (prefix is "bbb")

suffStart = 4, suffEnd = 5 (suffix is "cc")

maxLength = 3 (longest repeating substring is "bbb" at indices [0, 2])

**Query 2: Update index 3 to 'c'**

**Update**: s[3] = 'c'

**New String**: s = "bbbccc"

**Update Leaf Node**:

The leaf node at index 3 is updated to new Node(1, 3, 3, 3, 3).

**Update Internal Nodes**:

The parent nodes are updated recursively.

For example, the node representing [2, 3] ("bc") will now have:

prefStart = 2, prefEnd = 2 (prefix is 'b')

suffStart = 3, suffEnd = 3 (suffix is 'c')

maxLength = 1 (no repeating characters)

**Update Root Node**:

The root node now represents "bbbccc".

After merging, the root node will have:

prefStart = 0, prefEnd = 2 (prefix is "bbb")

suffStart = 3, suffEnd = 5 (suffix is "ccc")

maxLength = 3 (longest repeating substring is either "bbb" or "ccc")

**Query 3: Update index 3 to 'b'**

**Update**: s[3] = 'b'

**New String**\*: s = "bbbbcc"

**Update Leaf Node**:

The leaf node at index 3 is updated to new Node(1, 3, 3, 3, 3).

**Update Internal Nodes**:

The parent nodes are updated recursively.

For example, the node representing [2, 3] ("bb") will now have:

prefStart = 2, prefEnd = 3 (prefix is "bb")

suffStart = 2, suffEnd = 3 (suffix is "bb")

maxLength = 2 (longest repeating substring is "bb")

**Update Root Node**:

The root node now represents "bbbbcc".

After merging, the root node will have:

prefStart = 0, prefEnd = 3 (prefix is "bbbb")

suffStart = 4, suffEnd = 5 (suffix is "cc")

maxLength = 4 (longest repeating substring is "bbbb" at indices [0, 3])

**Final Result:**

After processing all queries, the result array will contain the maximum length of repeating characters after each update:

int[] result = {3, 3, 4};

**Summary:**

The Segment Tree efficiently maintains the state of the string and updates the maximum length of repeating characters after each query.

The prefStart, prefEnd, suffStart, and suffEnd variables help in tracking the prefix and suffix segments, which are crucial for merging segments and updating the tree.

**Q3: Can you still use same example input to explain how below 'merge' method works ?**

String queryCharacters = "bcb";

int[] queryIndices = new int[] {1,3,3};

String s = "babacc";

private Node merge(Node left, Node right, int tl, int tm, int tr) {

int maxLength = Math.max(left.maxLength, right.maxLength);

int prefStart = left.prefStart;

int prefEnd = left.prefEnd;

int suffStart = right.suffStart;

int suffEnd = right.suffEnd;

// Check if the middle character connects the left and right segments

if (s.charAt(tm) == s.charAt(tm + 1)) {

maxLength = Math.max(maxLength, right.prefEnd - left.suffStart + 1);

if (left.prefEnd - left.prefStart + 1 == tm - tl + 1) {

prefEnd = right.prefEnd;

}

if (right.suffEnd - right.suffStart + 1 == tr - tm) {

suffStart = left.suffStart;

}

}

return new Node(maxLength, prefStart, prefEnd, suffStart, suffEnd);

}

Certainly! Let's walk through how the merge method works using the provided example input:

**Input:**

s = "babacc"

queryCharacters = "bcb"

queryIndices = [1, 3, 3]

**Initial Segment Tree Construction:**

Before processing any queries, the Segment Tree is built for the initial string s = "babacc". Let's focus on how the merge method is used during the construction of the tree.

**Step-by-Step Merge Process:**

**Leaf Nodes:**

Each leaf node represents a single character in the string.

For example:

Node for index 0 ('b'): new Node(1, 0, 0, 0, 0)

Node for index 1 ('a'): new Node(1, 1, 1, 1, 1)

Node for index 2 ('b'): new Node(1, 2, 2, 2, 2)

Node for index 3 ('a'): new Node(1, 3, 3, 3, 3)

Node for index 4 ('c'): new Node(1, 4, 4, 4, 4)

Node for index 5 ('c'): new Node(1, 5, 5, 5, 5)

**Internal Nodes:**

Internal nodes are built by merging their left and right children using the merge method.

**Example Merge: Segment [0, 1] ("ba")**

**Left Child**: Node for index 0 ('b'): new Node(1, 0, 0, 0, 0)

**Right Child:** Node for index 1 ('a'): new Node(1, 1, 1, 1, 1)

**Merge Parameters:**

tl = 0, tm = 0, tr = 1

**Merge Steps:**

**Initial Values:**

maxLength = Math.max(1, 1) = 1

prefStart = 0, prefEnd = 0

suffStart = 1, suffEnd = 1

**Check Middle Connection:**

s.charAt(tm) = s.charAt(0) = 'b'

s.charAt(tm + 1) = s.charAt(1) = 'a'

Since 'b' != 'a', the middle characters do not connect.

**Result:**

The merged node is new Node(1, 0, 0, 1, 1).

**Example Merge: Segment [4, 5] ("cc")**

**Left Child:** Node for index 4 ('c'): new Node(1, 4, 4, 4, 4)

**Right Child:** Node for index 5 ('c'): new Node(1, 5, 5, 5, 5)

**Merge Parameters:**

tl = 4, tm = 4, tr = 5

**Merge Steps:**

**Initial Values:**

maxLength = Math.max(1, 1) = 1

prefStart = 4, prefEnd = 4

suffStart = 5, suffEnd = 5

**Check Middle Connection:**

s.charAt(tm) = s.charAt(4) = 'c'

s.charAt(tm + 1) = s.charAt(5) = 'c'

Since 'c' == 'c', the middle characters connect.

**Update Values:**

maxLength = Math.max(1, 5 - 4 + 1) = 2

Since the left prefix extends to the middle (4 - 4 + 1 == 1), prefEnd = 5.

Since the right suffix extends to the middle (5 - 5 + 1 == 1), suffStart = 4.

**Result:**

The merged node is new Node(2, 4, 5, 4, 5).

**Processing Queries:**

Now, let's process each query and see how the merge method is used during updates.

**Query 1: Update index 1 to 'b'**

Update: s[1] = 'b'

New String: s = "bbbacc"

**Update Leaf Node:**

The leaf node at index 1 is updated to new Node(1, 1, 1, 1, 1).

**Update Internal Nodes:**

The parent nodes are updated recursively using the merge method.

**Example Merge: Segment [0, 1] ("bb")**

**Left Child**: Node for index 0 ('b'): new Node(1, 0, 0, 0, 0)

**Right Child**: Node for index 1 ('b'): new Node(1, 1, 1, 1, 1)

**Merge Parameters:**

tl = 0, tm = 0, tr = 1

**Merge Steps**:

**Initial Values:**

maxLength = Math.max(1, 1) = 1

prefStart = 0, prefEnd = 0

suffStart = 1, suffEnd = 1

**Check Middle Connection:**

s.charAt(tm) = s.charAt(0) = 'b'

s.charAt(tm + 1) = s.charAt(1) = 'b'

Since 'b' == 'b', the middle characters connect.

**Update Values:**

maxLength = Math.max(1, 1 - 0 + 1) = 2

Since the left prefix extends to the middle (0 - 0 + 1 == 1), prefEnd = 1.

Since the right suffix extends to the middle (1 - 1 + 1 == 1), suffStart = 0.

**Result:**

The merged node is new Node(2, 0, 1, 0, 1).

**Query 2: Update index 3 to 'c'**

**Update**: s[3] = 'c'

**New String**: s = "bbbccc"

**Update Leaf Node**:

The leaf node at index 3 is updated to new Node(1, 3, 3, 3, 3).

**Update Internal Nodes**:

The parent nodes are updated recursively using the merge method.

**Example Merge: Segment [2, 3] ("bc")**

**Left Child**: Node for index 2 ('b'): new Node(1, 2, 2, 2, 2)

**Right Child**: Node for index 3 ('c'): new Node(1, 3, 3, 3, 3)

**Merge Parameters**:

tl = 2, tm = 2, tr = 3

**Merge Steps**:

**Initial Values**:

maxLength = Math.max(1, 1) = 1

prefStart = 2, prefEnd = 2

suffStart = 3, suffEnd = 3

**Check Middle Connection**:

s.charAt(tm) = s.charAt(2) = 'b'

s.charAt(tm + 1) = s.charAt(3) = 'c'

Since 'b' != 'c', the middle characters do not connect.

**Result**:

The merged node is new Node(1, 2, 2, 3, 3).

**Query 3: Update index 3 to 'b'**

**Update**: s[3] = 'b'

**New String**: s = "bbbbcc"

**Update Leaf Node**:

The leaf node at index 3 is updated to new Node(1, 3, 3, 3, 3).

**Update Internal Nodes**:

The parent nodes are updated recursively using the merge method.

**Example Merge: Segment [2, 3] ("bb")**

**Left Child**: Node for index 2 ('b'): new Node(1, 2, 2, 2, 2)

**Right Child**: Node for index 3 ('b'): new Node(1, 3, 3, 3, 3)

**Merge Parameters**:

tl = 2, tm = 2, tr = 3

**Merge Steps**:

**Initial Values**:

maxLength = Math.max(1, 1) = 1

prefStart = 2, prefEnd = 2

suffStart = 3, suffEnd = 3

**Check Middle Connection**:

s.charAt(tm) = s.charAt(2) = 'b'

s.charAt(tm + 1) = s.charAt(3) = 'b'

Since 'b' == 'b', the middle characters connect.

**Update Values**:

maxLength = Math.max(1, 3 - 2 + 1) = 2

Since the left prefix extends to the middle (2 - 2 + 1 == 1), prefEnd = 3.

Since the right suffix extends to the middle (3 - 3 + 1 == 1), suffStart = 2.

**Result**:

The merged node is new Node(2, 2, 3, 2, 3).

**Summary:**

The merge method combines two child nodes into a parent node by checking if the middle characters connect.

If the middle characters connect, it updates the maxLength, prefEnd, and suffStart accordingly.

This process ensures that the Segment Tree maintains the correct state of the string after each update.

Let me know if you need further clarification! 😊

**Q4: I still don't understand why the merge method use array index 'tl, tm, tr' to correlate with the merge process of left and right node, is that necessary or we don't actually need array index 'tl, tm, tr' to help the merge process ?**

Great question! Let's dive deeper into why the merge method uses the array indices tl, tm, and tr and whether they are necessary for the merge process.

**Why Are tl, tm, and tr Used in the merge Method?**

The indices tl, tm, and tr are used in the merge method to:

**Determine the Boundaries of the Current Segment**:

tl is the left boundary of the current segment.

tr is the right boundary of the current segment.

tm is the midpoint of the current segment, calculated as tm = tl + (tr - tl) / 2.

**Check if the Middle Characters Connect**:

**The merge method checks if the characters at the boundary of the left and right segments (i.e., s.charAt(tm) and s.charAt(tm + 1)) are the same.**

**If they are the same, it means the left and right segments can be merged into a longer repeating substring.**

**Update Prefix and Suffix**:

The indices tl, tm, and tr are used to determine if the prefix or suffix of the left or right segment extends to the middle of the current segment.

For example:

**If the prefix of the left segment extends to the middle (left.prefEnd - left.prefStart + 1 == tm - tl + 1), then the prefix of the parent node can be extended to include the prefix of the right segment.**

**Similarly, if the suffix of the right segment extends to the middle (right.suffEnd - right.suffStart + 1 == tr - tm), then the suffix of the parent node can be extended to include the suffix of the left segment.**

**Are tl, tm, and tr Necessary?**

Yes, **tl, tm, and tr are necessary** for the merge process because they provide critical information about the boundaries of the current segment and the relationship between the left and right segments. Without these indices, the merge method would not be able to:

**Check if the Middle Characters Connect**:

The indices tm and tm + 1 are used to access the characters at the boundary of the left and right segments.

Without tm, the method would not know where the left segment ends and the right segment begins.

**Determine if the Prefix or Suffix Extends to the Middle**:

The indices tl, tm, and tr are used to calculate the length of the prefix and suffix segments.

Without these indices, the method would not be able to determine if the prefix or suffix of a segment extends to the middle of the current segment.

**Example to Illustrate the Need for tl, tm, and tr**

Let's revisit the example of merging the segments [0, 1] ("ba") and [2, 3] ("ba") in the string s = "baba".

**Without tl, tm, and tr:**

The merge method would not know where the left segment ends and the right segment begins.

It would not be able to check if the characters at the boundary (s.charAt(1) and s.charAt(2)) are the same.

It would not be able to determine if the prefix or suffix of a segment extends to the middle of the current segment.

**With tl, tm, and tr:**

The merge method knows that:

The left segment is [0, 1] ("ba").

The right segment is [2, 3] ("ba").

The midpoint tm is 1, so it checks if s.charAt(1) ('a') and s.charAt(2) ('b') are the same.

Since 'a' != 'b', the middle characters do not connect, and the merged node will have:

prefStart = 0, prefEnd = 0 (prefix is 'b').

suffStart = 3, suffEnd = 3 (suffix is 'a').

maxLength = 1 (no repeating characters).

**Key Takeaways:**

The indices tl, tm, and tr are **necessary** for the merge method to:

Determine the boundaries of the current segment.

Check if the middle characters connect.

Update the prefix and suffix of the parent node.

Without these indices, the merge method would not have enough information to correctly merge the left and right segments.

Let me know if you need further clarification or additional examples! 😊

**Q5: But look at another correct solution in C++ below, it looks like not use similar variables in merge method, it just check left node's last char if equals to right node's first char or not, why in this solution it not use 'tl, tm, tr' and how it overcome ?**

**Refer to**

<https://leetcode.com/problems/longest-substring-of-one-repeating-character/solutions/1879490/segment-tree/>

struct node {

char lc = 0, rc = 0;

int pref = 0, suf = 0, longest = 0, sz = 1;

void merge(node &l, node &r) {

longest = max(l.longest, r.longest);

if (l.rc == r.lc)

longest = max(longest, l.suf + r.pref);

sz = l.sz + r.sz;

lc = l.lc;

rc = r.rc;

pref = l.pref + (l.pref == l.sz && l.lc == r.lc ? r.pref : 0);

suf = r.suf + (r.suf == r.sz && r.rc == l.rc ? l.suf : 0);

}

};

int st\_set(vector<node>& st, int pos, char ch, int i, int l, int r) {

if (l <= pos && pos <= r) {

if (l != r) {

auto m = l + (r - l) / 2, li = 2 \* i + 1, ri = 2 \* i + 2;

st\_set(st, pos, ch, li, l, m);

st\_set(st, pos, ch, ri, m + 1, r);

st[i].merge(st[li], st[ri]);

}

else {

st[i].lc = st[i].rc = ch;

st[i].suf = st[i].pref = st[i].longest = 1;

}

}

return st[i].longest;

}

vector<int> longestRepeating(string s, string queryCharacters, vector<int>& queryIndices) {

vector<int> res;

int powOf2 = 1, sz = s.size();

while (powOf2 < sz)

powOf2 <<= 1;

vector<node> st(powOf2 \* 2);

for (int i = 0; i < s.size(); ++i)

st\_set(st, i, s[i], 0, 0, powOf2 - 1);

for (int j = 0; j < queryCharacters.size(); ++j)

res.push\_back(st\_set(st, queryIndices[j], queryCharacters[j], 0, 0, powOf2 - 1));

return res;

}

You're absolutely right! The C++ solution you provided does not explicitly use tl, tm, and tr in the merge method, and it still works correctly. Let's break down why this is possible and how this solution overcomes the need for tl, tm, and tr.

**Key Differences in the C++ Solution**

**Node Structure**:

The node structure in the C++ solution stores:

lc and rc: The first and last characters of the segment.

pref and suf: The lengths of the prefix and suffix of repeating characters.

longest: The length of the longest repeating substring in the segment.

sz: The size of the segment.

These variables encode all the necessary information about the segment without explicitly needing tl, tm, and tr.

**Merge Logic**:

The merge method in the C++ solution combines two child nodes (l and r) into the parent node.

It checks if the last character of the left child (l.rc) matches the first character of the right child (r.lc).

If they match, it updates the longest value to include the combined length of the suffix of the left child and the prefix of the right child (l.suf + r.pref).

It also updates the pref and suf values based on whether the prefix or suffix extends across the entire segment.

**Segment Size (sz)**:

The sz variable in the node structure represents the size of the segment.

This is used to determine if the prefix or suffix extends across the entire segment. For example:

If l.pref == l.sz, it means the entire left segment is a repeating prefix.

If r.suf == r.sz, it means the entire right segment is a repeating suffix.

**Why tl, tm, and tr Are Not Needed in the C++ Solution**

The C++ solution avoids the need for tl, tm, and tr by:

**Storing Segment Information in the Node**:

The sz variable in the node structure encodes the size of the segment, which replaces the need for tl and tr.

The lc and rc variables store the first and last characters of the segment, which replace the need for tm to check the boundary between the left and right segments.

**Using sz to Determine Prefix and Suffix**:

The pref and suf values are updated based on whether they extend across the entire segment (l.pref == l.sz or r.suf == r.sz).

This eliminates the need to calculate the midpoint (tm) or check the boundaries explicitly.

**Implicit Boundary Checking**:

The merge method implicitly checks the boundary between the left and right segments by comparing l.rc and r.lc.

This is equivalent to checking s.charAt(tm) == s.charAt(tm + 1) in the Java solution, but it avoids explicitly using tm.

**How the C++ Solution Overcomes the Need for tl, tm, and tr**

**Segment Size (sz)**:

**The sz variable in the node structure encodes the size of the segment, which is used to determine if the prefix or suffix extends across the entire segment.**

For example:

**If l.pref == l.sz, it means the entire left segment is a repeating prefix.**

**If r.suf == r.sz, it means the entire right segment is a repeating suffix.**

**First and Last Characters (lc and rc)**:

**The lc and rc variables store the first and last characters of the segment, which are used to check if the left and right segments can be merged.**

For example:

**If l.rc == r.lc, it means the last character of the left segment matches the first character of the right segment, so they can be merged.**

**Prefix and Suffix Lengths (pref and suf)**:

The pref and suf variables store the lengths of the prefix and suffix of repeating characters, which are used to update the longest value.

For example:

If l.rc == r.lc, the longest value is updated to include the combined length of the suffix of the left segment and the prefix of the right segment (l.suf + r.pref).

**Example Walkthrough**

Let's walk through an example to see how the C++ solution works without tl, tm, and tr.

**Input:**

s = "babacc"

queryCharacters = "bcb"

queryIndices = [1, 3, 3]

**Initial Segment Tree Construction:**

**Leaf Nodes**:

Each leaf node represents a single character in the string.

For example:

Node for index 0 ('b'): lc = 'b', rc = 'b', pref = 1, suf = 1, longest = 1, sz = 1

Node for index 1 ('a'): lc = 'a', rc = 'a', pref = 1, suf = 1, longest = 1, sz = 1

Node for index 2 ('b'): lc = 'b', rc = 'b', pref = 1, suf = 1, longest = 1, sz = 1

Node for index 3 ('a'): lc = 'a', rc = 'a', pref = 1, suf = 1, longest = 1, sz = 1

Node for index 4 ('c'): lc = 'c', rc = 'c', pref = 1, suf = 1, longest = 1, sz = 1

Node for index 5 ('c'): lc = 'c', rc = 'c', pref = 1, suf = 1, longest = 1, sz = 1

**Internal Nodes**:

Internal nodes are built by merging their left and right children using the merge method.

**Example Merge: Segment [4, 5] ("cc")**

**Left Child**: Node for index 4 ('c'): lc = 'c', rc = 'c', pref = 1, suf = 1, longest = 1, sz = 1

**Right Child**: Node for index 5 ('c'): lc = 'c', rc = 'c', pref = 1, suf = 1, longest = 1, sz = 1

**Merge Steps**:

**Check if l.rc == r.lc**:

l.rc = 'c', r.lc = 'c' → They match.

**Update longest**:

longest = max(l.longest, r.longest) = max(1, 1) = 1

longest = max(longest, l.suf + r.pref) = max(1, 1 + 1) = 2

**Update pref and suf**:

pref = l.pref + (l.pref == l.sz && l.lc == r.lc ? r.pref : 0) = 1 + (1 == 1 && 'c' == 'c' ? 1 : 0) = 2

suf = r.suf + (r.suf == r.sz && r.rc == l.rc ? l.suf : 0) = 1 + (1 == 1 && 'c' == 'c' ? 1 : 0) = 2

**Result**:

The merged node is lc = 'c', rc = 'c', pref = 2, suf = 2, longest = 2, sz = 2.

**Summary**

The C++ solution avoids the need for tl, tm, and tr by encoding the segment size (sz) and the first and last characters (lc and rc) in the node structure.

The merge method uses these variables to check if the left and right segments can be merged and to update the longest, pref, and suf values.

This approach is more concise and avoids explicitly calculating the midpoint or checking the boundaries.

Let me know if you need further clarification! 😊

**Q6: need more explain about below 2 lines logic inside C++ solution's merge() method:**

pref = l.pref + (l.pref == l.sz && l.lc == r.lc ? r.pref : 0);

suf = r.suf + (r.suf == r.sz && r.rc == l.rc ? l.suf : 0);

Absolutely! Let's break down the logic of these two lines in the C++ solution's merge() method. These lines are responsible for updating the \*\*prefix (pref)\*\* and \*\*suffix (suf)\*\* of the parent node based on the prefix and suffix of the left and right child nodes.

**Key Variables:**

l: The left child node.

r: The right child node.

l.pref: The length of the prefix of the left child.

r.pref: The length of the prefix of the right child.

l.suf: The length of the suffix of the left child.

r.suf: The length of the suffix of the right child.

l.sz: The size of the left child's segment.

r.sz: The size of the right child's segment.

l.lc: The first character of the left child's segment.

r.lc: The first character of the right child's segment.

l.rc: The last character of the left child's segment.

r.rc: The last character of the right child's segment.

**Line 1: Updating the Prefix (pref)**

pref = l.pref + (l.pref == l.sz && l.lc == r.lc ? r.pref : 0);

**Explanation:**

**l.pref**:

This is the length of the prefix of the left child.

For example, if the left child's segment is "aaa", then l.pref = 3.

**l.pref == l.sz**:

**This checks if the prefix of the left child extends across the entire left segment.**

**If l.pref == l.sz, it means the entire left segment is a repeating prefix.**

For example, if the left child's segment is "aaa", then l.pref == l.sz is true.

**l.lc == r.lc**:

**This checks if the first character of the left child (l.lc) matches the first character of the right child (r.lc).**

**If they match, it means the prefix of the left child can be extended into the right child.**

**Ternary Operator (? :)**:

If both conditions (l.pref == l.sz and l.lc == r.lc) are true, then the prefix of the parent node includes the prefix of the right child (r.pref).

Otherwise, it does not include the prefix of the right child (0).

**Final Calculation**:

The prefix of the parent node (pref) is the prefix of the left child (l.pref) plus the additional prefix from the right child (if applicable).

**Example:**

Left child: "aaa" (l.pref = 3, l.sz = 3, l.lc = 'a')

Right child: "aab" (r.pref = 2, r.lc = 'a')

Since l.pref == l.sz (3 == 3) and l.lc == r.lc ('a' == 'a'), the prefix of the parent node is:

pref = l.pref + r.pref = 3 + 2 = 5

The parent node's prefix is "aaaaa".

**Line 2: Updating the Suffix (suf)**

suf = r.suf + (r.suf == r.sz && r.rc == l.rc ? l.suf : 0);

**Explanation:**

**r.suf**:

This is the length of the suffix of the right child.

For example, if the right child's segment is "bbb", then r.suf = 3.

**r.suf == r.sz**:

**This checks if the suffix of the right child extends across the entire right segment.**

**If r.suf == r.sz, it means the entire right segment is a repeating suffix.**

For example, if the right child's segment is "bbb", then r.suf == r.sz is true.

**r.rc == l.rc**:

This checks if the last character of the right child (r.rc) matches the last character of the left child (l.rc).

If they match, it means the suffix of the right child can be extended into the left child.

**Ternary Operator (? :):**

If both conditions (r.suf == r.sz and r.rc == l.rc) are true, then the suffix of the parent node includes the suffix of the left child (l.suf).

Otherwise, it does not include the suffix of the left child (0).

**Final Calculation**:

The suffix of the parent node (suf) is the suffix of the right child (r.suf) plus the additional suffix from the left child (if applicable).

**Example:**

Left child: "abb" (l.suf = 2, l.rc = 'b')

Right child: "bbb" (r.suf = 3, r.sz = 3, r.rc = 'b')

Since r.suf == r.sz (3 == 3) and r.rc == l.rc ('b' == 'b'), the suffix of the parent node is:

suf = r.suf + l.suf = 3 + 2 = 5

The parent node's suffix is "bbbbb".

**Why These Lines Are Necessary**

These two lines ensure that the \*\*prefix\*\* and \*\*suffix\*\* of the parent node are correctly updated based on the properties of the left and right child nodes. Specifically:

**Prefix**:

If the left child's prefix extends across its entire segment and matches the right child's prefix, the parent's prefix is extended into the right child.

This ensures that the parent's prefix is as long as possible.

**Suffix**:

If the right child's suffix extends across its entire segment and matches the left child's suffix, the parent's suffix is extended into the left child.

This ensures that the parent's suffix is as long as possible.

**Example Walkthrough**

Let's walk through an example to see how these lines work in practice.

**Input:**

Left child: "aaab" (l.pref = 3, l.suf = 1, l.sz = 4, l.lc = 'a', l.rc = 'b')

Right child: "bbbc" (r.pref = 3, r.suf = 1, r.sz = 4, r.lc = 'b', r.rc = 'c')

**Step 1: Update Prefix (pref)**

l.pref == l.sz → 3 == 4 → **False**

l.lc == r.lc → 'a' == 'b' → **False**

Since both conditions are false, the prefix of the parent node is:

pref = l.pref + 0 = 3 + 0 = 3

**Step 2: Update Suffix (suf)**

r.suf == r.sz → 1 == 4 → **False**

r.rc == l.rc → 'c' == 'b' → **False**

Since both conditions are false, the suffix of the parent node is:

suf = r.suf + 0 = 1 + 0 = 1

**Result:**

The parent node's prefix is "aaa" (length 3).

The parent node's suffix is "c" (length 1).

**Summary**

The two lines in the merge() method ensure that the **prefix** and **suffix** of the parent node are correctly updated based on the properties of the left and right child nodes.

They use the segment size (sz) and the first/last characters (lc and rc) to determine if the prefix or suffix can be extended across the boundary between the left and right segments.

This logic is essential for maintaining the correct state of the Segment Tree during updates.

Let me know if you need further clarification! 😊

**Q7: But why this time we compare l.lc with r.lc instead of l.rc == r.lc ?**

Great question! The comparison l.lc == r.lc in the **prefix update** logic and r.rc == l.rc in the **suffix update** logic might seem a bit confusing at first. Let me explain why these specific comparisons are used and how they differ from the l.rc == r.lc comparison.

**Key Idea: Prefix and Suffix Extensions**

The goal of the merge() method is to correctly update the **prefix** and **suffix** of the parent node based on the properties of the left and right child nodes. Specifically:

**Prefix**:

The prefix of the parent node is the **longest sequence of repeating characters starting from the beginning of the segment**.

To extend the prefix, we need to check if the **entire left segment is a repeating prefix** and if it **matches the prefix of the right segment.**

**Suffix**:

The suffix of the parent node is the **longest sequence of repeating characters ending at the end of the segment.**

To extend the suffix, we need to check if the **entire right segment is a repeating suffix** and if it **matches the suffix of the left segment.**

**Why l.lc == r.lc for Prefix Update?**

The comparison l.lc == r.lc is used in the **prefix update** logic to determine if the prefix of the parent node can be extended into the right segment. Here's why:

**l.lc**:

This is the first character of the left segment.

If the entire left segment is a repeating prefix (i.e., l.pref == l.sz), then all characters in the left segment are the same as l.lc.

**r.lc**:

This is the first character of the right segment.

**If l.lc == r.lc, it means the repeating prefix of the left segment can continue into the right segment.**

**Why Not l.rc == r.lc?:**

The l.rc (last character of the left segment) is not relevant for extending the **prefix**.

The prefix is about the **beginning** of the segment, so we only care about the **first character** of the left and right segments (l.lc and r.lc).

**Example:**

Left child: "aaa" (l.pref = 3, l.sz = 3, l.lc = 'a')

Right child: "aab" (r.pref = 2, r.lc = 'a')

Since l.pref == l.sz (3 == 3) and l.lc == r.lc ('a' == 'a'), the prefix of the parent node can be extended into the right segment:

pref = l.pref + r.pref = 3 + 2 = 5

The parent node's prefix is "aaaaa".

**Why r.rc == l.rc for Suffix Update?**

The comparison r.rc == l.rc is used in the **suffix update** logic to determine if the suffix of the parent node can be extended into the left segment. Here's why:

**r.rc**:

This is the **last character** of the right segment.

If the entire right segment is a repeating suffix (i.e., r.suf == r.sz), then all characters in the right segment are the same as r.rc.

**l.rc**:

This is the **last character** of the left segment.

If r.rc == l.rc, it means the repeating suffix of the right segment can **continue into the left segment**.

**Why Not r.lc == l.rc?**:

The r.lc (first character of the right segment) is not relevant for extending the **suffix**.

The suffix is about the **end** of the segment, so we only care about the **last character** of the left and right segments (l.rc and r.rc).

**Example:**

Left child: "abb" (l.suf = 2, l.rc = 'b')

Right child: "bbb" (r.suf = 3, r.sz = 3, r.rc = 'b')

Since r.suf == r.sz (3 == 3) and r.rc == l.rc ('b' == 'b'), the suffix of the parent node can be extended into the left segment:

suf = r.suf + l.suf = 3 + 2 = 5

The parent node's suffix is "bbbbb".

**Why l.rc == r.lc Is Not Used Here**

The comparison l.rc == r.lc is **not used** in this solution because:

**Purpose**:

The l.rc == r.lc comparison is typically used to check if the **boundary between the left and right segments** can be merged into a longer repeating substring.

However, in this solution, the \*\*prefix\*\* and \*\*suffix\*\* are updated independently, and the longest value is updated separately based on l.rc == r.lc.

**Separate Logic:**

The longest value is updated in a separate step:

if (l.rc == r.lc)

longest = max(longest, l.suf + r.pref);

This ensures that the longest value accounts for any merging at the boundary between the left and right segments.

**Summary**

**l.lc == r.lc** is used for the **prefix update** because the prefix is about the **beginning** of the segment, so we compare the first characters of the left and right segments.

**r.rc == l.rc** is used for the **suffix update** because the suffix is about the **end** of the segment, so we compare the last characters of the left and right segments.

The l.rc == r.lc comparison is used separately to update the longest value, ensuring that any merging at the boundary between the left and right segments is accounted for.

This design ensures that the **prefix**, **suffix**, and **longest** **repeating substring** are all correctly maintained during the merge process.

Let me know if you need further clarification! 😊

**Refer to**

[L56.P5.1.Merge Intervals](note://EBA980E5AA5C4417B1FC624E737697B7)

[L424.P2.6.Longest Repeating Character Replacement (Ref.L340)](note://1581773836A946879EB5237C85BFF6C2)

[L1446.Consecutive Characters](note://WEB3a48306b0e7b904730e5e13662ae7c9e)

[L307.Range Sum Query - Mutable](note://5677877215264F99A1992A5E4F26DCA6)

[L327.Count of Range Sum (Ref.L715, L315, L307)](note://WEBa8df841f1750bbdee6107f732c50620f)

[L2276.Count Integers in Intervals (Ref.L57,L56,L352,L715)](note://WEB2a7d56df47f01cdc79d5e90de4bab425)

[L2407.Longest Increasing Subsequence II (Ref.L300,L307)](note://779B31B629A1437683A4E6E303BEE1D6)