### **Interview questions for BuildHer Campaign.**

### **Easy 1**

Given a string s consisting of words and spaces, return *the length of the* ***last*** *word in the string.*

A **word** is a maximal

substring consisting of non-space characters only.

**Constraints:**

* 1 <= s.length <= 104
* s consists of only English letters and spaces ' '.
* There will be at least one word in s.

**CODE:**

def length\_of\_last\_word(s):

words = s.split()

if not words:

return 0

return len(words[-1])

# Example usage:

s = "Hello World"

result = length\_of\_last\_word(s)

print(result)

**OUTPUT:** 5

**EXPLANATION:**

### Certainly! Let's break down the logic and algorithm of the code for finding the length of the last word in a given string.

### 1. Split the String:

### - The first step is to split the input string into words. This is done using the `split` method.

### - The `split` method without any arguments splits the string based on whitespace by default. This gives us a list of words.

### 2. Access the Last Word:

### - Once the string is split, we can access the last word in the list.

### - In Python, indexing with `-1` refers to the last element in a list.

### 3. Calculate and Return Length:

### - After obtaining the last word, the code calculates its length using the `len` function.

### - The length of the last word is the desired result, so the function returns this length.

### 4. Example:

### - Let's consider the string "Hello World". After splitting, we get the list `['Hello', 'World']`.

### - Accessing the last element with `[-1]` gives us `'World'`.

### - The length of the last word, in this case, is 5, which is then returned by the function.

### 

### **Medium 1**

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the BST.

**CODE:**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def lowest\_common\_ancestor(root, p, q):

if p.val > q.val:

p, q = q, p

while root:

if root.val > q.val:

root = root.left

elif root.val < p.val:

root = root.right

else:

return root

# Example usage:

root = TreeNode(5)

root.left = TreeNode(3, TreeNode(2), TreeNode(4))

root.right = TreeNode(6, None, TreeNode(7))

# Find the LCA of nodes with values 2 and 4

p = TreeNode(2)

q = TreeNode(4)

result = lowest\_common\_ancestor(root, p, q)

print(result.val)

**OUTPUT:** 3

EXPLANATION:

To find the lowest common ancestor (LCA) of two nodes in a binary search tree (BST), you can follow these steps:

1. Start from the Root:

- Begin traversing the tree starting from the root.

2. Navigate Down the Tree:

- At each node, compare the values of the given nodes `p` and `q` with the current node's value.

- If both `p` and `q` are greater than the current node's value, it means both nodes are in the right subtree. So, move to the right child.

- If both `p` and `q` are less than the current node's value, it means both nodes are in the left subtree. So, move to the left child.

3. LCA Condition:

- If one of `p` or `q` is smaller and the other is larger than the current node's value, or if one of them is equal to the current node's value, then the current node is the lowest common ancestor.

- This is because in a BST, the left subtree of a node contains values smaller than the node, and the right subtree contains values larger than the node.

4. Repeat Until LCA is Found:

- Keep traversing down the tree until you find the LCA.

### **Hard 2**

You are given a string s. You can convert s to a

palindrome by adding characters in front of it.

Return *the shortest palindrome you can find by performing this transformation*.

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of lowercase English letters only.

**CODE:**

def shortest\_palindrome(s):

if not s:

return s

extended = s + "#" + s[::-1]

lps = [0] \* len(extended)

j = 0

for i in range(1, len(extended)):

while j > 0 and extended[i] != extended[j]:

j = lps[j - 1]

if extended[i] == extended[j]:

j += 1

lps[i] = j

palindrome\_length = lps[-1]

return s[palindrome\_length:][::-1] + s

# Example usage:

s = "abcd"

result = shortest\_palindrome(s)

print(result)

**OUTPUT:** dcbabcd

**EXPLANATION:**

To find the shortest palindrome by adding characters in front of a given string `s`, you can use the concept of the KMP (Knuth–Morris–Pratt) algorithm and build a modified string for pattern matching. The idea is to find the longest palindromic prefix of the reversed string.

Here's the step-by-step explanation:

1. Create Modified String:

- Construct a new string `new\_s` by concatenating the reversed `s` with a special character (n

2. Build Prefix Array (KMP):

- Build the prefix array (also known as the LPS array - Longest Proper Prefix which is also Suffix) for the modified string `new\_s` using the KMP algorithm.

3. Find Palindromic Prefix:

- The value at the last index of the prefix array indicates the length of the longest palindromic prefix of `new\_s`. Let's call this length `len\_palindrome`.

4. Construct Result:

- The characters from index `len\_palindrome` to the end in the reversed `s` need to be added to the front of `s` to make it a palindrome. Extract these characters and concatenate them in front of `s` to get the result.