Class Notes 6

- Python

  💡 **Question 1**

  Write a function that reverses a string. The input string is given as an array of characters s.

  You must do this by modifying the input array in-place with O(1) extra memory.

  Example 1:

  Input: s = ["h","e","l","l","o"]

  Output: ["o","l","l","e","h"]

  Solution:

  Two Pointers Approach

  In this approach, two pointers are used to process two array elements

  at the same time. Usual implementation is to set one pointer in the

  beginning and one at the end and then to move them until they both meet.

  Algorithm

  - Set pointer left at index 0, and pointer right at index n - 1,

    where n is a number of elements in the array.

  - While left < right:

    - Swap s[left] and s[right].
    - Move left pointer one step right, and right pointer one step left.

- **Complexity Analysis**

  - Time complexity: O(*N*) to swap *N*/2 element.
  - Space complexity: O(1), it's a constant space solution.

```
def reverseString(s):
    left = 0
    right = len(s) - 1
    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1
```

-

  💡 **Question 2**

  Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

  Example 1:

  Input: s = "leetcode"

  Output: 0

  Solution:

  The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway. The idea is to go through the string and save in a hash map the number of times each character appears in the string.

  And then we go through the string the second time, this time we use the hash map as a reference to check if a character is unique or not.

  If the character is unique, one could just return its index.

  Complexity Analysis

  - Time complexity : O(*N*) since we go through the string of length N two times.
  - Space complexity : O(1) because English alphabet contains 26 letters.

```
def firstUniqChar(s):
    count = {}
    n = len(s)
    # build hash map: character and how often it appears
    for i in range(n):
        c = s[i]
        count[c] = count.get(c, 0) + 1
```

```
    # find the index
    for i in range(n):
        if count[s[i]] == 1:
            return i
    return -1
```

-

  💡 **Question 3**

  Given an input string s, reverse the order of the **words**.

  A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

  Return *a string of the words in reverse order concatenated by a single space.*

  Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

  Example 1:

  Input: s = "the sky is blue"

  Output: "blue is sky the"

  Solution:

  Approach 1: Built-in Split + Reverse

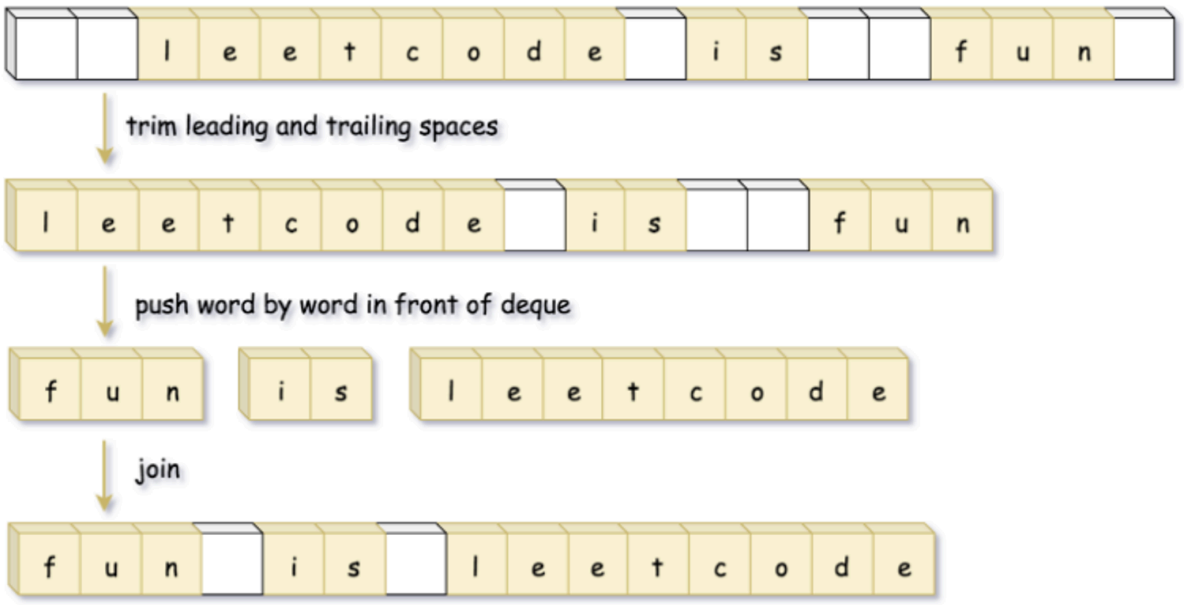
  Complexity Analysis

  - Time complexity: O(*N*), where N is a number of characters in the input string.
  - Space complexity: O(*N*), to store the result of split by spaces.

```
def reverseWords(s):
    # remove leading and trailing spaces
    s = s.strip()
    # split by multiple spaces
    wordList = s.split()
    # reverse the list of words
    wordList.reverse()
    # join the words with a space separator
```

```
    return ' '.join(wordList)
```

- **Approach 2: Deque of Words**



Complexity Analysis

- Time complexity: O(*N*).
- Space complexity: O(*N*).

```
from collections import deque

def reverseWords(s):
    left, right = 0, len(s) - 1
    # remove leading spaces
    while left <= right and s[left] == ' ':
        left += 1

    # remove trailing spaces
    while left <= right and s[right] == ' ':
        right -= 1

    d = deque()
    word = []
    # push word by word in front of deque
    while left <= right:
```

```
    c = s[left]

    if len(word) != 0 and c == ' ':
        d.appendleft(''.join(word))
        word = []
    elif c != ' ':
        word.append(c)
    left += 1

d.appendleft(''.join(word))

return ' '.join(d)
```

- 💡 **Question 4**

  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.

  Example 1:

  Input: s = "Hello World"

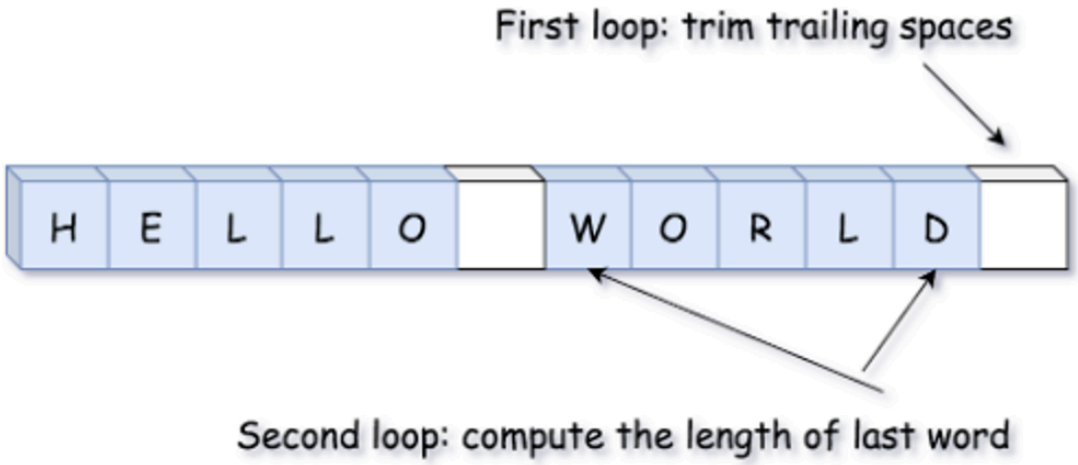
  Output: 5

  Explanation: The last word is "World" with length 5.

  Solution:

  One can break down the solution into two steps:

  - First, we would try to locate the last word, starting from the end of the string. We iterate the string in reverse order, consuming the empty spaces. When we first come across a non-space character, we know that we are at the last character of the last word.
  - Second, once we locate the last word. We count its length, starting from its last character. Again, we could use a loop here.



- **Complexity**
  - Time Complexity: O(*N*), where *N* is the length of the input string.

    In the worst case, the input string might contain only a single word, which implies that we would need to iterate through the entire string to obtain the result.
  - Space Complexity: O(1), only constant memory is consumed, regardless the input.

```
def lengthOfLastWord(s):
    # trim the trailing spaces
    p = len(s) - 1
    while p >= 0 and s[p] == ' ':
        p -= 1

    # compute the length of last word
    length = 0
    while p >= 0 and s[p] != ' ':
        p -= 1
        length += 1
    return length
```

💡 Question 5

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Solution:

Approach 1: Horizontal scanning

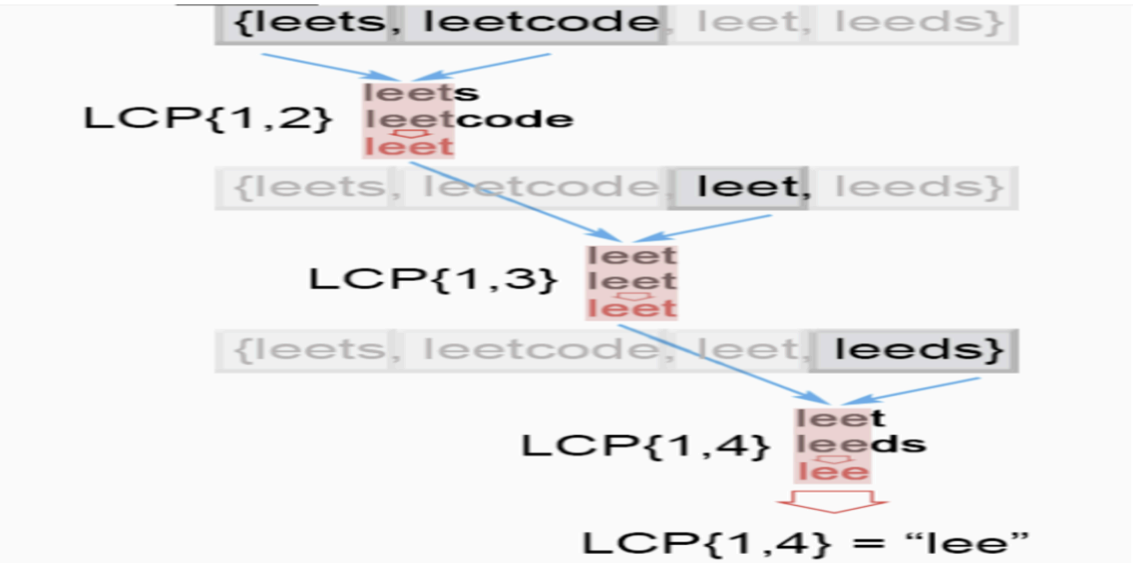
Intuition

For a start we will describe a simple way of finding the longest prefix shared by a set of strings $LCP(S_1 \dots S_n)$.
We will use the observation that :

$$LCP(S_1 \dots S_n) = LCP(LCP(LCP(S_1, S_2), S_3), \dots S_n)$$

Algorithm

To employ this idea, the algorithm iterates through the strings $[S_1 \dots S_n]$, finding at each iteration i the longest common prefix of strings $LCP(S_1 \dots S_i)$ When $LCP(S_1 \dots S_i)$ is an empty string, the algorithm ends. Otherwise after n iterations, the algorithm returns $LCP(S_1 \dots S_n)$.



Complexity Analysis

- Time complexity : $O(S)$, where S is the sum of all characters in all strings.

  In the worst case all n strings are the same. The algorithm compares the string $S1$ with the other strings $[S_2 \dots S_n]$ There are S character comparisons, where S is the sum of all characters in the input array.

- Space complexity : $O(1)$. We only used constant extra space.

```
def longestCommonPrefix(strs):
  if len(strs) == 0:
     return ""
  prefix = strs[0]
  for i in range(1, len(strs)):
     while strs[i].find(prefix) != 0:
        prefix = prefix[:-1]
        if len(prefix) == 0:
           return ""
  return prefix
```

-

  💡 **Question 6**

  Given a string s, find the length of the **longest substring** without repeating characters.

  Example 1:

  Input: s = "abcabcbb"

  Output: 3

  Explanation: The answer is "abc", with the length of 3.

Instead of using a set to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

Algorithm

The reason is that if $s[j]$ have a duplicate in the range $[i, j)$ with index j', we don't need to increase i little by little. We can skip all the elements in the range $[i, j']$ and let i to be $j' + 1$ directly.

Complexity Analysis

- Time complexity : $O(n)$. Index j will iterate n times.
- Space complexity : $O(min(m, n))$. Same as the previous approach.

```
def lengthOfLongestSubstring(s):
    n = len(s)
    ans = 0
    map = {}  # current index of character
    # try to extend the range [i, j]
    i = 0
    for j in range(n):
        if s[j] in map:
            i = max(map[s[j]], i)
        ans = max(ans, j - i + 1)
        map[s[j]] = j + 1
```

return ans

-
- JavaScript

    💡 **Question 1**

    Write a function that reverses a string. The input string is given as an array of characters s.

    You must do this by modifying the input array in-place with O(1) extra memory.

    Example 1:

    Input: s = ["h","e","l","l","o"]

    Output: ["o","l","l","e","h"]

    Solution:

    Two Pointers Approach

    In this approach, two pointers are used to process two array elements

    at the same time. Usual implementation is to set one pointer in the

    beginning and one at the end and then to move them until they both meet.

    Algorithm

    - Set pointer left at index 0, and pointer right at index n - 1,

      where n is a number of elements in the array.

    - While left < right:

        - Swap s[left] and s[right].
        - Move left pointer one step right, and right pointer one step left.

- **Complexity Analysis**

    - Time complexity: O(*N*) to swap *N*/2 element.
    - Space complexity: O(1), it's a constant space solution.

```
function reverseString(s) {
```

```
    let left = 0;
    let right = s.length - 1;
    while (left < right) {
        const tmp = s[left];
        s[left++] = s[right];
        s[right--] = tmp;
    }
}
```

-

  💡 **Question 2**

  Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

  Example 1:

  Input: s = "leetcode"

  Output: 0

  Solution:

  The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway. The idea is to go through the string and save in a hash map the number of times each character appears in the string.

  And then we go through the string the second time, this time we use the hash map as a reference to check if a character is unique or not.

  If the character is unique, one could just return its index.

  Complexity Analysis

  - Time complexity : O(*N*) since we go through the string of length N two times.
  - Space complexity : O(1) because English alphabet contains 26 letters.

```
function firstUniqChar(s) {
    const count = new Map();
    const n = s.length;
    // build hash map: character and how often it appears
    for (let i = 0; i < n; i++) {
        const c = s.charAt(i);
```

```
        count.set(c, (count.get(c) || 0) + 1);
    }

    // find the index
    for (let i = 0; i < n; i++) {
        if (count.get(s.charAt(i)) === 1) {
            return i;
        }
    }
    return -1;
}
```

- 💡 **Question 3**

  Given an input string s, reverse the order of the **words**.

  A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

  Return *a string of the words in reverse order concatenated by a single space.*

  Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

  Example 1:

  Input: s = "the sky is blue"

  Output: "blue is sky the"

  Solution:

  Approach 1: Built-in Split + Reverse

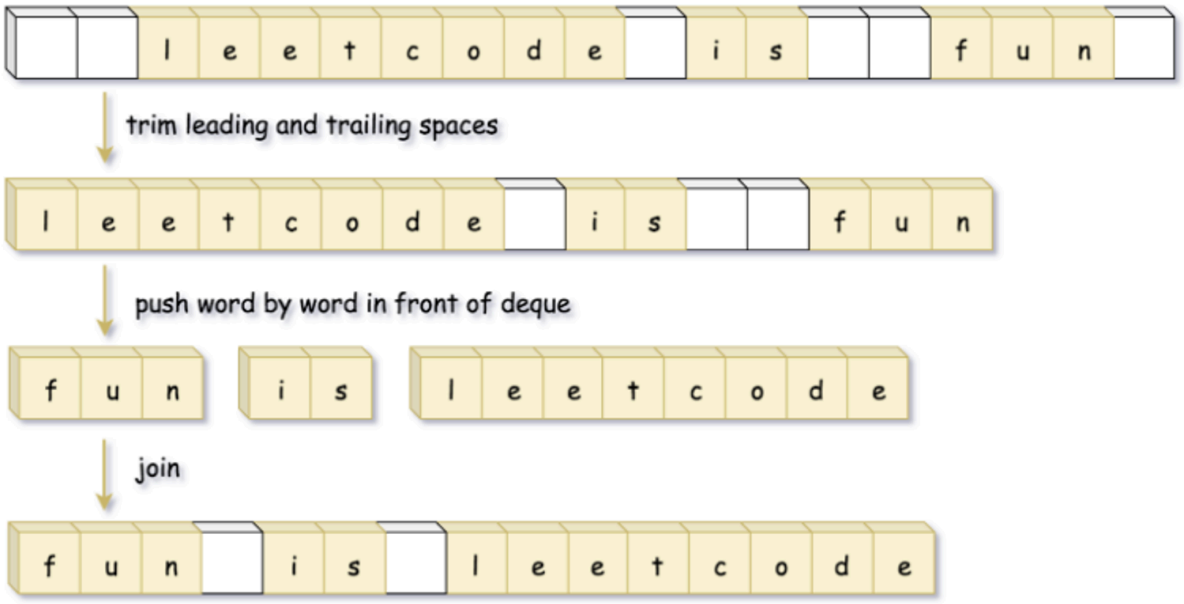
  Complexity Analysis

  - Time complexity: O(*N*), where N is a number of characters in the input string.
  - Space complexity: O(*N*), to store the result of split by spaces.

```
function reverseWords(s) {
    // remove leading and trailing spaces
    s = s.trim();
```

```
    // split by multiple spaces
    let wordList = s.split(/\\s+/);
    // reverse the array of words
    wordList.reverse();
    // join the words with a space separator
    return wordList.join(' ');
}
```

- **Approach 2: Deque of Words**



Complexity Analysis

- Time complexity: O(*N*).
- Space complexity: O(*N*).

```
function reverseWords(s) {
    let left = 0, right = s.length - 1;
    // remove leading spaces
    while (left <= right && s.charAt(left) === ' ') {
        left++;
    }

    // remove trailing spaces
    while (left <= right && s.charAt(right) === ' ') {
        right--;
```

```
    }

    const d = [];
    let word = '';
    // push word by word in front of deque
    while (left <= right) {
        const c = s.charAt(left);

        if (word.length !== 0 && c === ' ') {
            d.unshift(word);
            word = '';
        } else if (c !== ' ') {
            word += c;
        }
        left++;
    }
    d.unshift(word);

    return d.join(' ');
}
```

-

  💡 **Question 4**

  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.

  Example 1:

  Input: s = "Hello World"

  Output: 5

  Explanation: The last word is "World" with length 5.

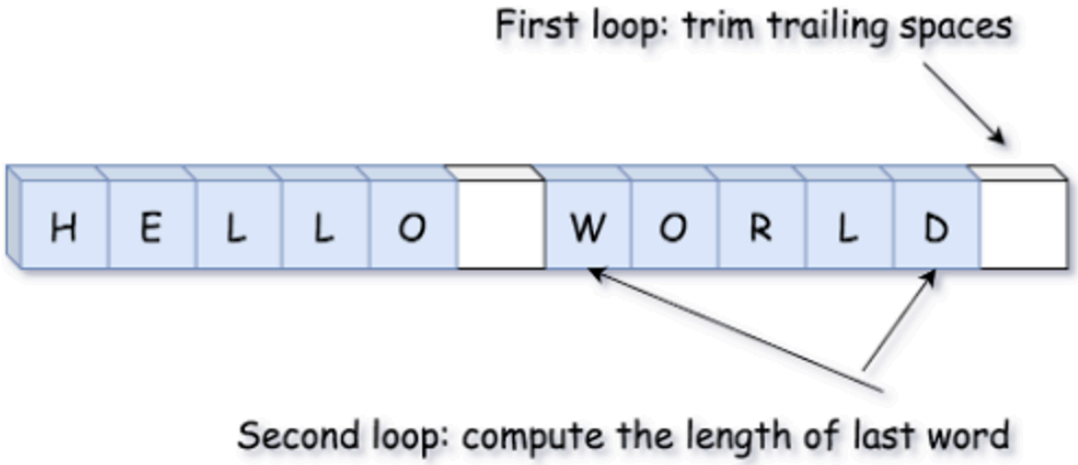
  Solution:

  One can break down the solution into two steps:

  - First, we would try to locate the last word, starting from the end of the string. We iterate the string in reverse order, consuming the empty spaces. When we first

come across a non-space character, we know that we are at the last character of the last word.

- Second, once we locate the last word. We count its length, starting from its last character. Again, we could use a loop here.



- **Complexity**

  - Time Complexity: O(*N*), where *N* is the length of the input string.

    In the worst case, the input string might contain only a single word, which implies that we would need to iterate through the entire string to obtain the result.

  - Space Complexity: O(1), only constant memory is consumed, regardless the input.

```
function lengthOfLastWord(s) {
    // trim the trailing spaces
    let p = s.length - 1;
    while (p >= 0 && s.charAt(p) === ' ') {
        p--;
    }

    // compute the length of last word
    let length = 0;
    while (p >= 0 && s.charAt(p) !== ' ') {
        p--;
        length++;
    }
    return length;
```

}

💡 Question 5

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Solution:

Approach 1: Horizontal scanning

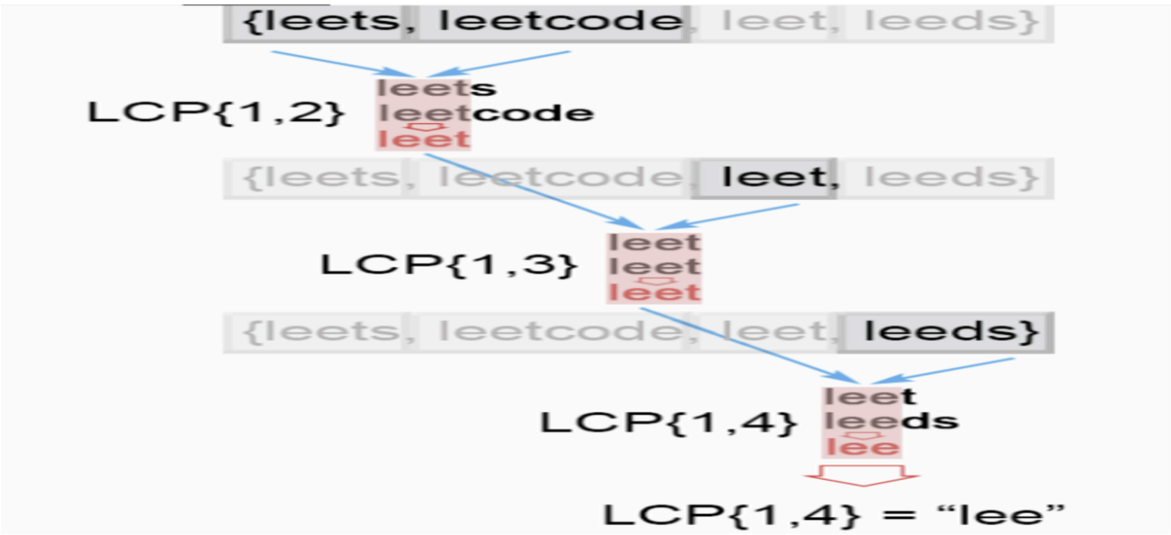
Intuition

For a start we will describe a simple way of finding the longest prefix shared by a set of strings $LCP(S_1 \dots S_n)$.
We will use the observation that :

$$LCP(S_1 \dots S_n) = LCP(LCP(LCP(S_1, S_2), S_3), \dots S_n)$$

Algorithm

To employ this idea, the algorithm iterates through the strings $[S_1 \dots S_n]$, finding at each iteration i the longest common prefix of strings $LCP(S_1 \dots S_i)$ When $LCP(S_1 \dots S_i)$ is an empty string, the algorithm ends. Otherwise after n iterations, the algorithm returns $LCP(S_1 \dots S_n)$.



Complexity Analysis

- Time complexity : $O(S)$, where S is the sum of all characters in all strings.

  In the worst case all n strings are the same. The algorithm compares the string $S1$ with the other strings $[S_2 \dots S_n]$ There are S character comparisons, where S is the sum of all characters in the input array.

- Space complexity : $O(1)$. We only used constant extra space.

```
function longestCommonPrefix(strs) {
    if (strs.length === 0) {
        return "";
    }
    let prefix = strs[0];
    for (let i = 1; i < strs.length; i++) {
        while (strs[i].indexOf(prefix) !== 0) {
            prefix = prefix.slice(0, prefix.length - 1);
            if (prefix.length === 0) {
                return "";
            }
        }
    }
    return prefix;
}
```

-

  💡 **Question 6**

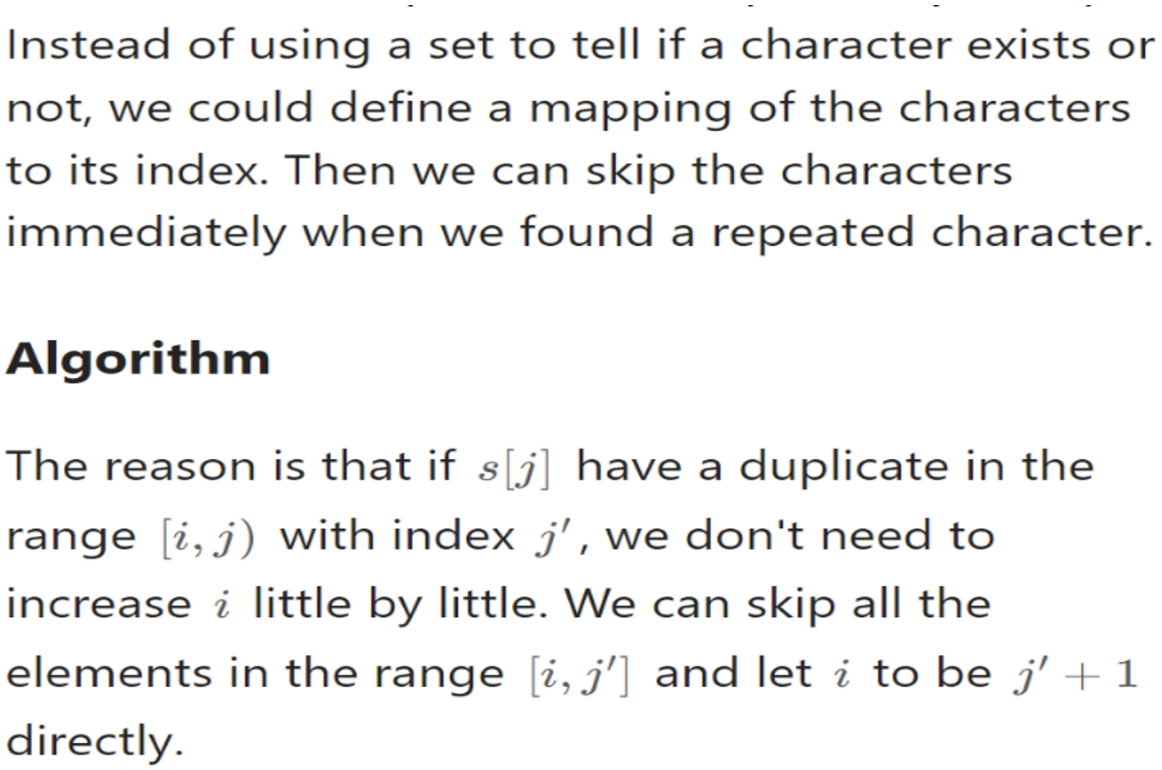
  Given a string s, find the length of the **longest substring** without repeating characters.

  Example 1:

  Input: s = "abcabcbb"

  Output: 3

  Explanation: The answer is "abc", with the length of 3.



  Complexity Analysis

  - Time complexity : $O(n)$. Index j will iterate n times.
  - Space complexity : $O(min(m, n))$. Same as the previous approach.

```
function lengthOfLongestSubstring(s) {
    const n = s.length;
```

```
    let ans = 0;
    const map = new Map(); // current index of character
    // try to extend the range [i, j]
    let i = 0;
    for (let j = 0; j < n; j++) {
        if (map.has(s.charAt(j))) {
            i = Math.max(map.get(s.charAt(j)), i);
        }
        ans = Math.max(ans, j - i + 1);
        map.set(s.charAt(j), j + 1);
    }
    return ans;
}
```

-
- Java

💡 **Question 1**

Write a function that reverses a string. The input string is given as an array of characters s.

You must do this by modifying the input array in-place with O(1) extra memory.

Example 1:

Input: s = ["h","e","l","l","o"]

Output: ["o","l","l","e","h"]

Solution:

Two Pointers Approach

In this approach, two pointers are used to process two array elements

at the same time. Usual implementation is to set one pointer in the

beginning and one at the end and then to move them until they both meet.

Algorithm

- Set pointer left at index 0, and pointer right at index n - 1,

  where n is a number of elements in the array.

- While left < right:

  - Swap s[left] and s[right].
  - Move left pointer one step right, and right pointer one step left.

- **Complexity Analysis**

  - Time complexity: O(*N*) to swap *N*/2 element.
  - Space complexity: O(1), it's a constant space solution.

```
class Solution {
   public void reverseString(char[] s) {
      int left = 0, right = s.length - 1;
      while (left < right) {
         char tmp = s[left];
         s[left++] = s[right];
         s[right--] = tmp;
      }
   }
}
```

- 💡 **Question 2**

  Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

  Example 1:

  Input: s = "leetcode"

  Output: 0

  Solution:

  The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway. The idea is to go through the string and save in a hash map the number of times each character appears in the string.

  And then we go through the string the second time, this time we use the hash map as a

reference to check if a character is unique or not.

If the character is unique, one could just return its index.

Complexity Analysis

- Time complexity : O(*N*) since we go through the string of length N two times.
- Space complexity : O(1) because English alphabet contains 26 letters.

```
class Solution {
    public int firstUniqChar(String s) {
        HashMap<Character, Integer> count = new HashMap<Character, Integer>();
        int n = s.length();
        // build hash map : character and how often it appears
        for (int i = 0; i < n; i++) {
            char c = s.charAt(i);
            count.put(c, count.getOrDefault(c, 0) + 1);
        }

        // find the index
        for (int i = 0; i < n; i++) {
            if (count.get(s.charAt(i)) == 1)
                return i;
        }
        return -1;
    }
}
```

-

💡 **Question 3**

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return *a string of the words in reverse order concatenated by a single space.*

Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Example 1:

Input: s = "the sky is blue"

Output: "blue is sky the"

Solution:

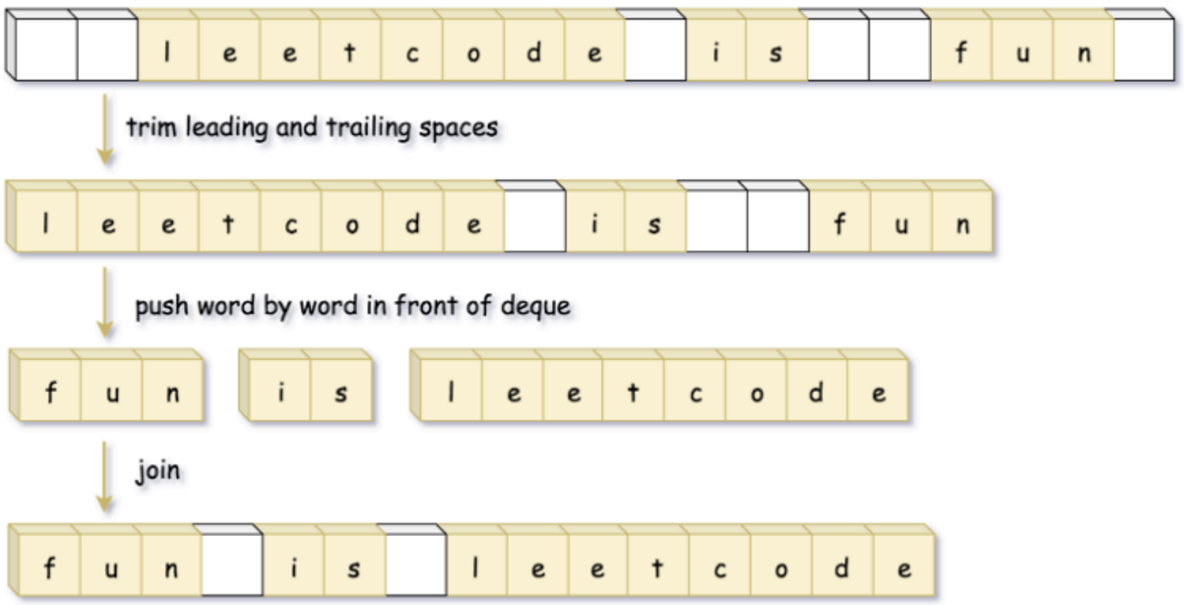
Approach 1: Built-in Split + Reverse

Complexity Analysis

- Time complexity: O(*N*), where N is a number of characters in the input string.
- Space complexity: O(*N*), to store the result of split by spaces.

```
class Solution {
  public String reverseWords(String s) {
    // remove leading spaces
    s = s.trim();
    // split by multiple spaces
    List<String> wordList = Arrays.asList(s.split("\\s+"));
    Collections.reverse(wordList);
    return String.join(" ", wordList);
  }
}
```

- **Approach 2: Deque of Words**



Complexity Analysis

- Time complexity: O(*N*).
- Space complexity: O(*N*).

```
class Solution {
  public String reverseWords(String s) {
    int left = 0, right = s.length() - 1;
    // remove leading spaces
    while (left <= right && s.charAt(left) == ' ') ++left;

    // remove trailing spaces
    while (left <= right && s.charAt(right) == ' ') --right;

    Deque<String> d = new ArrayDeque();
    StringBuilder word = new StringBuilder();
    // push word by word in front of deque
    while (left <= right) {
      char c = s.charAt(left);

      if ((word.length() != 0) && (c == ' ')) {
        d.offerFirst(word.toString());
        word.setLength(0);
      } else if (c != ' ') {
        word.append(c);
      }
      ++left;
    }
    d.offerFirst(word.toString());

    return String.join(" ", d);
  }
}
```

-

💡 Question 4

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.

Example 1:

Input: s = "Hello World"

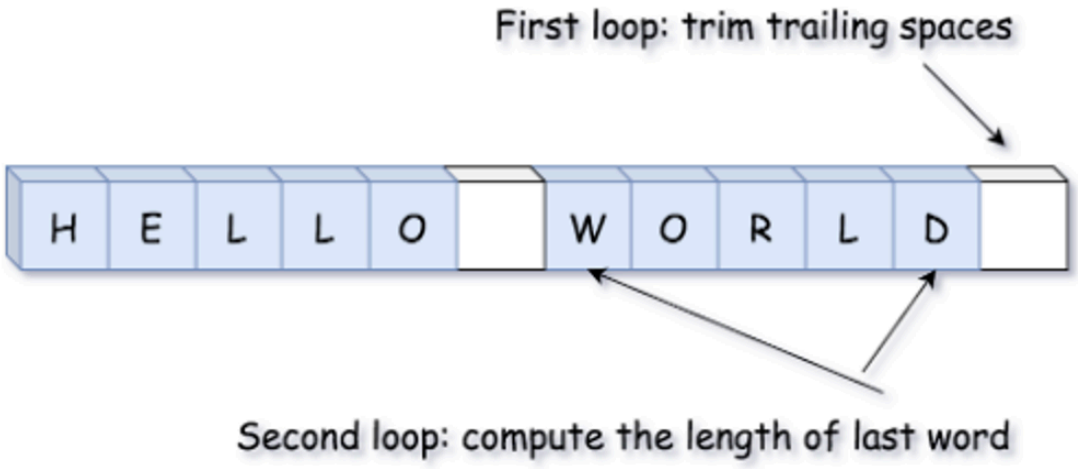
Output: 5

Explanation: The last word is "World" with length 5.

Solution:

One can break down the solution into two steps:

- First, we would try to locate the last word, starting from the end of the string. We iterate the string in reverse order, consuming the empty spaces. When we first come across a non-space character, we know that we are at the last character of the last word.
- Second, once we locate the last word. We count its length, starting from its last character. Again, we could use a loop here.



-

Complexity

- Time Complexity: O(*N*), where *N* is the length of the input string.

  In the worst case, the input string might contain only a single word, which implies that we would need to iterate through the entire string to obtain the result.

- Space Complexity: O(1), only constant memory is consumed, regardless the input.

```
class Solution {
```

```
    public int lengthOfLastWord(String s) {
        // trim the trailing spaces
        int p = s.length() - 1;
        while (p >= 0 && s.charAt(p) == ' ') {
            p--;
        }

        // compute the length of last word
        int length = 0;
        while (p >= 0 && s.charAt(p) != ' ') {
            p--;
            length++;
        }
        return length;
    }
}
```

💡 Question 5

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Solution:

Approach 1: Horizontal scanning

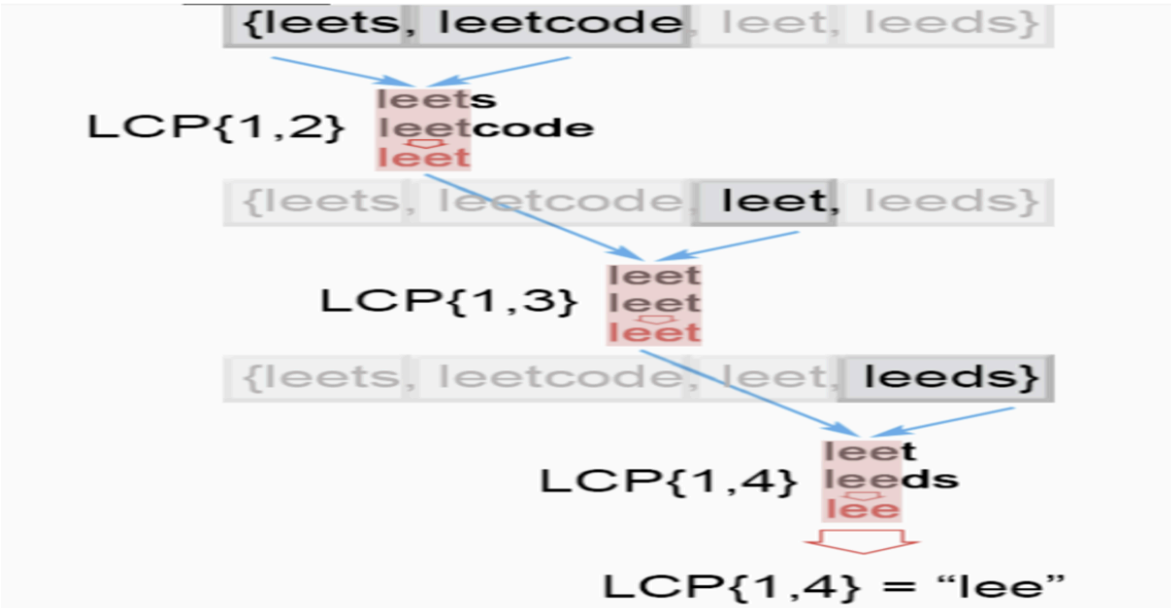
Intuition

For a start we will describe a simple way of finding the longest prefix shared by a set of strings $LCP(S_1 \dots S_n)$.
We will use the observation that :

$$LCP(S_1 \dots S_n) = LCP(LCP(LCP(S_1, S_2), S_3), \dots S_n)$$

Algorithm

To employ this idea, the algorithm iterates through the strings $[S_1 \dots S_n]$, finding at each iteration i the longest common prefix of strings $LCP(S_1 \dots S_i)$ When $LCP(S_1 \dots S_i)$ is an empty string, the algorithm ends. Otherwise after n iterations, the algorithm returns $LCP(S_1 \dots S_n)$.



Complexity Analysis

- Time complexity : $O(S)$, where S is the sum of all characters in all strings.

  In the worst case all n strings are the same. The algorithm compares the string $S1$ with the other strings $[S_2 \dots S_n]$ There are S character comparisons, where S is the sum of all characters in the input array.

- Space complexity : $O(1)$. We only used constant extra space.

```
class Solution {
  public String longestCommonPrefix(String[] strs) {
        if (strs.length == 0) return "";
        String prefix = strs[0];
        for (int i = 1; i < strs.length; i++)
          while (strs[i].indexOf(prefix) != 0) {
            prefix = prefix.substring(0, prefix.length() - 1);
            if (prefix.isEmpty()) return "";
          }
        return prefix;
            }
}
```

-

  💡 **Question 6**

  Given a string s, find the length of the **longest substring** without repeating characters.

  Example 1:

  Input: s = "abcabcbb"

  Output: 3

  Explanation: The answer is "abc", with the length of 3.

Instead of using a set to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

Algorithm

The reason is that if $s[j]$ have a duplicate in the range $[i, j)$ with index j', we don't need to increase i little by little. We can skip all the elements in the range $[i, j']$ and let i to be $j' + 1$ directly.

Complexity Analysis

- Time complexity : $O(n)$. Index j will iterate n times.
- Space complexity : $O(min(m, n))$. Same as the previous approach.

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length(), ans = 0;
        Map<Character, Integer> map = new HashMap<>(); // current index of character
        // try to extend the range [i, j]
        for (int j = 0, i = 0; j < n; j++) {
            if (map.containsKey(s.charAt(j))) {
                i = Math.max(map.get(s.charAt(j)), i);
            }
            ans = Math.max(ans, j - i + 1);
            map.put(s.charAt(j), j + 1);
        }
        return ans;
    }
}
```

-

- C++

  💡 **Question 1**

  Write a function that reverses a string. The input string is given as an array of characters s.

  You must do this by modifying the input array in-place with O(1) extra memory.

  Example 1:

  Input: s = ["h","e","l","l","o"]

  Output: ["o","l","l","e","h"]

  Solution:

  Two Pointers Approach

  In this approach, two pointers are used to process two array elements

  at the same time. Usual implementation is to set one pointer in the

  beginning and one at the end and then to move them until they both meet.

  Algorithm

  - Set pointer left at index 0, and pointer right at index n - 1,

    where n is a number of elements in the array.

  - While left < right:

    - Swap s[left] and s[right].
    - Move left pointer one step right, and right pointer one step left.

- **Complexity Analysis**

  - Time complexity: O(*N*) to swap *N*/2 element.
  - Space complexity: O(1), it's a constant space solution.

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int left = 0, right = s.size() - 1;
```

```
        while (left < right) {
            char tmp = s[left];
            s[left++] = s[right];
            s[right--] = tmp;
        }
    }
};
```

-

  💡 **Question 2**

  Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

  Example 1:

  Input: s = "leetcode"

  Output: 0

  Solution:

  The best possible solution here could be of a linear time because to ensure that the character is unique you have to check the whole string anyway. The idea is to go through the string and save in a hash map the number of times each character appears in the string.

  And then we go through the string the second time, this time we use the hash map as a reference to check if a character is unique or not.

  If the character is unique, one could just return its index.

  Complexity Analysis

  - Time complexity : O(*N*) since we go through the string of length N two times.
  - Space complexity : O(1) because English alphabet contains 26 letters.

```
class Solution {
public:
    int firstUniqChar(string s) {
        unordered_map<char, int> count;
        int n = s.length();

        // build hash map: character and how often it appears
```

```
        for (int i = 0; i < n; i++) {
            char c = s[i];
            count[c] = count[c] + 1;
        }

        // find the index
        for (int i = 0; i < n; i++) {
            if (count[s[i]] == 1) {
                return i;
            }
        }
        return -1;
    }
};
```

-

  💡 **Question 3**

  Given an input string s, reverse the order of the **words**.

  A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

  Return *a string of the words in reverse order concatenated by a single space.*

  Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

  Example 1:

  Input: s = "the sky is blue"

  Output: "blue is sky the"

  Solution:

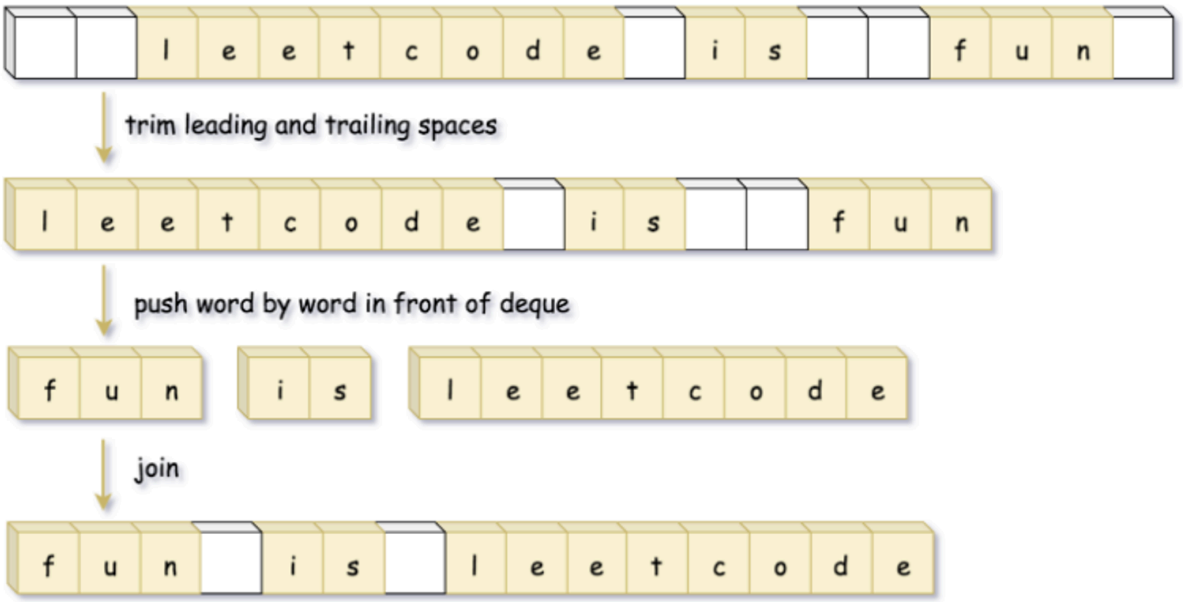
  Approach 1: Built-in Split + Reverse

  Complexity Analysis

  - Time complexity: O(*N*), where N is a number of characters in the input string.
  - Space complexity: O(*N*), to store the result of split by spaces.

```
class Solution {
public:
    string reverseWords(string &s){
        stringstream ss(s);
        string word, ans;
        while(ss >> word){
            ans = word + " " + ans;
        }
        return ans.substr(0, ans.size()-1);;
    }
};
```

- **Approach 2: Deque of Words**



Complexity Analysis

- Time complexity: O(*N*).
- Space complexity: O(*N*).

```
class Solution {
public:
    string reverseWords(string s) {
        int left = 0, right = s.length() - 1;

        // Remove leading spaces
```

```
        while (left <= right && s[left] == ' ')
            ++left;

        // Remove trailing spaces
        while (left <= right && s[right] == ' ')
            --right;

        deque<string> d;
        stringstream word;

        // Push word by word in front of deque
        while (left <= right) {
            char c = s[left];

            if (word.str().length() != 0 && c == ' ') {
                d.push_front(word.str());
                word.str(string());
            } else if (c != ' ') {
                word << c;
            }
            ++left;
        }

        d.push_front(word.str());

        stringstream result;

        // Join the words with a space
        while (!d.empty()) {
            result << d.front();
            d.pop_front();
            if (!d.empty())
                result << ' ';
        }

        return result.str();
    }
};
```

-

  💡 **Question 4**

  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.

Example 1:

Input: s = "Hello World"

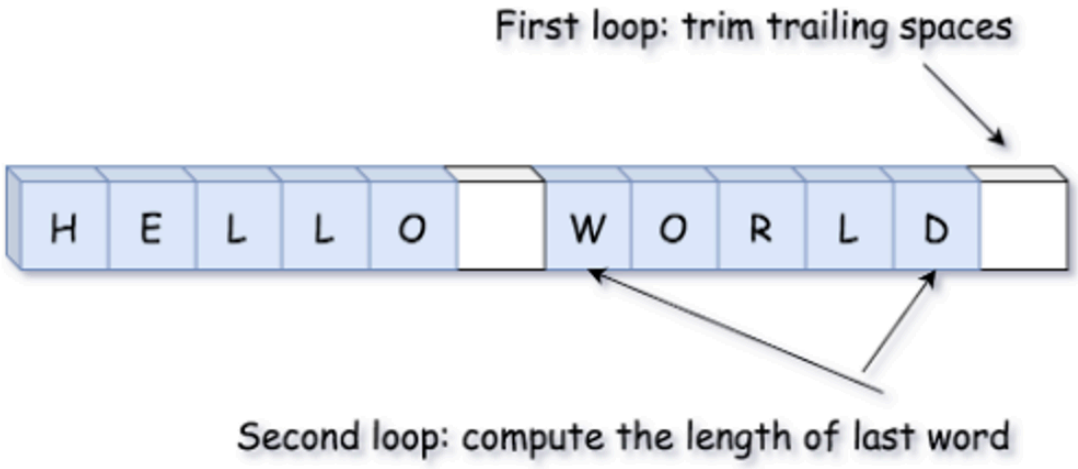
Output: 5

Explanation: The last word is "World" with length 5.

Solution:

One can break down the solution into two steps:

- First, we would try to locate the last word, starting from the end of the string. We iterate the string in reverse order, consuming the empty spaces. When we first come across a non-space character, we know that we are at the last character of the last word.
- Second, once we locate the last word. We count its length, starting from its last character. Again, we could use a loop here.



Complexity

- Time Complexity: O(*N*), where *N* is the length of the input string.

  In the worst case, the input string might contain only a single word, which implies that we would need to iterate through the entire string to obtain the result.

- Space Complexity: O(1), only constant memory is consumed, regardless the input.

```
class Solution {
public:
    int lengthOfLastWord(string s) {
        // Trim the trailing spaces
        int p = s.length() - 1;
        while (p >= 0 && s[p] == ' ') {
            p--;
        }

        // Compute the length of the last word
        int length = 0;
        while (p >= 0 && s[p] != ' ') {
            p--;
            length++;
        }
        return length;
    }
};
```

💡 Question 5

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]

Output: "fl"

Solution:

Approach 1: Horizontal scanning

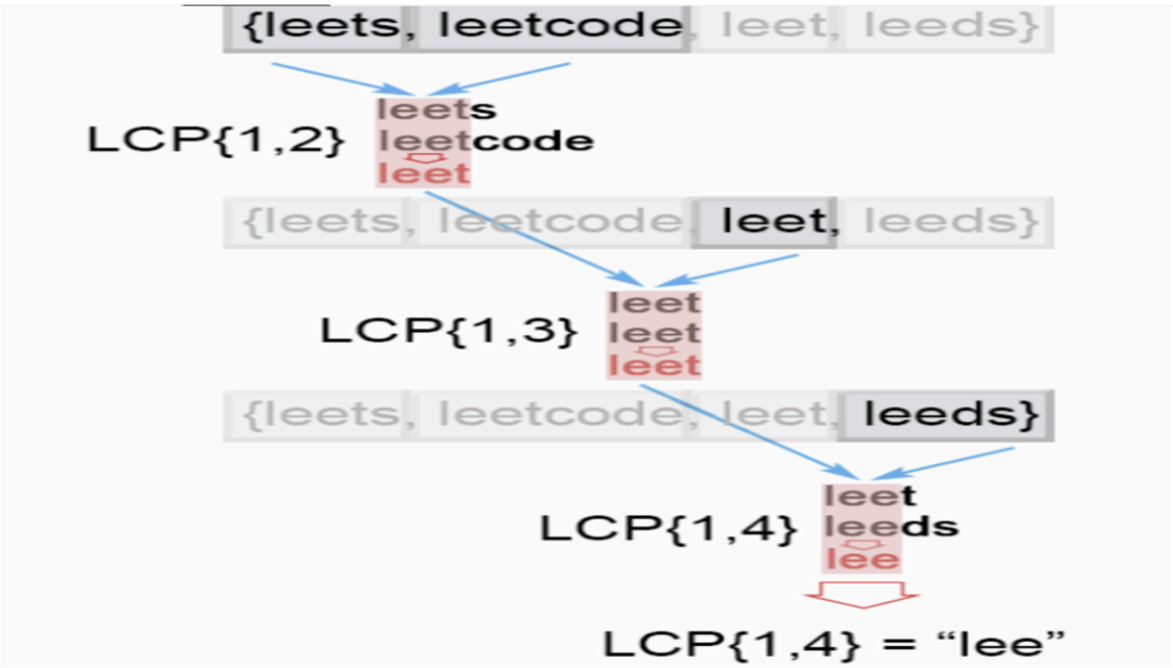
Intuition

For a start we will describe a simple way of finding the longest prefix shared by a set of strings $LCP(S_1 \dots S_n)$.
We will use the observation that :

$$LCP(S_1 \dots S_n) = LCP(LCP(LCP(S_1, S_2), S_3), \dots S_n)$$

Algorithm

To employ this idea, the algorithm iterates through the strings $[S_1 \dots S_n]$, finding at each iteration i the longest common prefix of strings $LCP(S_1 \dots S_i)$ When $LCP(S_1 \dots S_i)$ is an empty string, the algorithm ends. Otherwise after n iterations, the algorithm returns $LCP(S_1 \dots S_n)$.



Complexity Analysis

- Time complexity : $O(S)$, where S is the sum of all characters in all strings.

  In the worst case all n strings are the same. The algorithm compares the string $S1$ with the other strings $[S_2 \dots S_n]$ There are S character comparisons, where S is the sum of all characters in the input array.

- Space complexity : $O(1)$. We only used constant extra space.

```
 class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if (strs.empty())
            return "";

        string prefix = strs[0];

        for (int i = 1; i < strs.size(); i++) {
            while (strs[i].find(prefix) != 0) {
                prefix = prefix.substr(0, prefix.length() - 1);
                if (prefix.empty())
                    return "";
            }
        }
        return prefix;
    }
};
```

- 💡 **Question 6**

  Given a string s, find the length of the **longest substring** without repeating characters.

  Example 1:

  Input: s = "abcabcbb"

  Output: 3

  Explanation: The answer is "abc", with the length of 3.

Instead of using a set to tell if a character exists or not, we could define a mapping of the characters to its index. Then we can skip the characters immediately when we found a repeated character.

Algorithm

The reason is that if $s[j]$ have a duplicate in the range $[i, j)$ with index j', we don't need to increase i little by little. We can skip all the elements in the range $[i, j']$ and let i to be $j' + 1$ directly.

Complexity Analysis

- Time complexity : $O(n)$. Index j will iterate n times.
- Space complexity: $O(min(m, n))$. Same as the previous approach.

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.length();
        int ans = 0;
        unordered_map<char, int> map; // current index of character

        // Try to extend the range [i, j]
        for (int j = 0, i = 0; j < n; j++) {
            if (map.find(s[j]) != map.end()) {
                i = max(map[s[j]], i);
            }

            ans = max(ans, j - i + 1);
            map[s[j]] = j + 1;
```

```
        }

        return ans;
    }
};
```

-