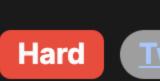
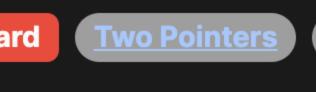
# 1163. Last Substring in Lexicographical Order





**Problem Description** 

String

Given a string s, the goal is to find the substring that is the largest in lexicographical order and return it. A lexicographical order is essentially dictionary order, so for example, "b" is greater than "a", and "ba" is greater than "ab". When determining the

lexicographical order of two strings, the comparison is made character by character starting from the first character. As soon as a difference is found, the string with the greater character at that position is considered larger. If all compared characters are equal and the strings are of different lengths, the longer string is considered to be larger.

# Intuition

could be the largest in lexicographical order. Starting with the first character, we treat it as the initial largest substring. As we move through the string with another pointer, we compare new potential substrings with the current largest one.

To solve this problem, we can leverage a two-pointer approach to iterate through the string and track potential substrings that

where the comparison difference was found. If the initial pointer's substring is larger, we simply move the second pointer forward to explore further options. By comparing characters at the offset (k) where the substrings start to differ, we can efficiently find the largest substring without needing to compare entire substrings each time. This utilizes the fact that any prefix that isn't the largest isn't worth considering because any extension to it will also not be the

Each time we find that the substring starting at the second pointer is larger, we update our initial pointer to the location just after

largest. We keep doing this until we've traversed the entire string, and our first pointer will indicate the beginning of the largest possible substring in lexicographical order, which we return by slicing the string from this position to the end. Solution Approach

## The solution is based on a smart iteration using two pointers, i and j, that scan through the string to find the largest

compare characters at an offset from the pointers i and j. Data structure wise, the only requirement is the input string s; no additional complex data structures are needed. We initialize i at 0, indicating the start of the current largest substring and set j to 1, which is the start of the substring we want to compare with the current largest. The variable k is initialized to 0, and it indicates the offset from i and j where the current

lexicographical substring. These pointers represent the start of the potentially largest substrings. The third pointer, k, is used to

comparison is happening. The algorithm proceeds as follows: While j + k is less than the length of s, we are not done comparing.

2. Compare the characters at s[i + k] and s[j + k]. 3. If s[i + k] is equal to s[j + k], we have not yet found a difference, so we increment k to continue the comparison at the next character. 4. If s[i + k] is less than s[j + k], we have found a larger substring starting at j. We update i to be i + k + 1, move j ahead to i + 1, and reset

once when it is part of a competing substring (tracked by j).

- k to 0 to start comparisons at the beginning of the new substrings. 5. If s[i + k] is greater than s[j + k], we keep our current largest substring starting at i intact, simply move j ahead by k + 1 to skip the lesser
- substring, and reset k to 0. This process ensures that we are always aware of the largest substring seen so far (marked by i) and efficiently skip over parts

of the string that cannot contribute to a larger lexicographical substring. In terms of complexity, this approach ensures a worst-case time complexity of O(n) where n is the length of the string. This is

because each character will be compared at most twice. Once when it is part of a potential largest substring (tracked by i), and

Finally, when the loop completes, the index i points to the start of the last substring in lexicographical order, and we return s[i:], the substring from i to the end of s.

This two-pointer approach with character comparison is key because it strategically narrows down the search for the largest

**Example Walkthrough** 

lexicographical substring without redundant comparisons or the use of extra space, making it both efficient and elegant.

# Compare characters at i + k and j + k, which means comparing s[0] with s[1] ("a" vs. "b"). Since "b" is greater, we update i

next characters.

substring of s is s[i:] which is "c".

to j, which is 1, so now i = 1. Reset j to i + 1 which makes j = 2 and reset k to 0.

Let's say our string s is "ababc", and we want to find the substring that is the largest in lexicographical order.

Now, our largest substring candidate starts at index 1 with "b". Compare s[i + k] with s[j + k] again, which is now "b" vs.

Initialize i, j, and k to 0, 1, and 0, respectively. Our initial largest substring is "a" at index 0.

- "a". Here, "b" is greater so we just move j ahead to j + k + 1, which is 3, and reset k to 0. Continue with the comparisons. We have "b" vs. "b" (s[1] vs. s[3]), which are equal, so we increment k to 1 to compare the •
- Next comparison is "a" vs. "c" (s[1 + k] vs. s[3 + k]). Here "c" is greater, so we find a new largest substring starting at index 3. We update i to j, which is 3, reset j to i + 1, which makes j = 4, and k to 0.
- Our last comparison would be "b" vs. "c" (s [3] vs. s [4]). "c" is greater, so i stays at 3. Now j + k is equal to the length of s, and we exit the loop. The last index i was updated to is 3, so the largest lexicographical
- Ultimately, the algorithm efficiently deduced that "c" is the largest substring without having to do an exhaustive search or comparison of all potential substrings.

# Initialize pointer (i) for the start of the current candidate substring.

# If characters at the current offset are equal, increase the offset.

if string[current start + offset] == string[compare start + offset]:

Solution Implementation

**Python** 

## # Initialize pointer (j) as the start of the next substring to compare. # Initialize (k) as the offset from both i and j during comparison. current\_start = 0

offset = 0

compare\_start = 1

offset += 1

compareIndex++;

maxCharIndex = currentIndex;

currentIndex = maxCharIndex + 1;

currentIndex += compareIndex + 1;

compareIndex = 0; // Reset compareIndex

compareIndex = 0; // Reset compareIndex

def lastSubstring(self, string: str) -> str:

# Iterate until the end of string is reached.

while compare\_start + offset < len(string):</pre>

class Solution:

```
# If the current character in the comparison substring is greater,
           # it becomes the new candidate. Update current_start to be compare_start.
            elif string[current_start + offset] < string[compare_start + offset]:</pre>
                current_start = max(current_start + offset + 1, compare_start)
                offset = 0
                # Ensure compare_start is always after current_start.
                if current_start >= compare_start:
                    compare_start = current_start + 1
            # If the current character in the candidate substring is greater,
            # continue with the next substring by moving compare_start.
            else:
                compare_start += offset + 1
                offset = 0
       # Return the last substring starting from the candidate position.
       return string[current_start:]
Java
class Solution {
    public String lastSubstring(String str) {
        int length = str.length(); // Length of the string
        int maxCharIndex = 0; // Index of the start of the last substring with the highest lexicographical order
        int currentIndex = 1; // Current index iterating through the string
        int compareIndex = 0; // Index used for comparing characters
       // Loop through the string once
       while (currentIndex + compareIndex < length) {</pre>
           // Compare characters at the current index and the current maximum substring index
            int diff = str.charAt(maxCharIndex + compareIndex) - str.charAt(currentIndex + compareIndex);
```

if (diff == 0) { // Characters are equal, move to the next character for comparison

} else if (diff < 0) { // Current character is larger, update maxCharIndex to current index</pre>

} else { // Current character is smaller, move past the current substring for comparison

```
// Create and return the substring starting from maxCharIndex to the end of the string
       return str.substring(maxCharIndex);
C++
class Solution {
public:
   // Function to find the lexicographically largest substring of 's'
   string lastSubstring(string s) {
       int strSize = s.size();  // The size of the input string
       int startIndex = 0;  // The starting index of the current candidate for the result
       // 'nextIndex' - The index of the next potential candidate
       // 'offset' - The offset from both 'startIndex' and 'nextIndex' to compare characters
       for (int nextIndex = 1, offset = 0; nextIndex + offset < strSize;) {</pre>
           // If characters at the current offset are the same, just go to next character
           if (s[startIndex + offset] == s[nextIndex + offset]) {
               ++offset;
           // If the current character at 'nextIndex' + 'offset' is greater,
           // it means this could be a new candidate for the result
           else if (s[startIndex + offset] < s[nextIndex + offset]) {</pre>
               startIndex = nextIndex; // Set the 'startIndex' to 'nextIndex'
               ++startIndex; // Increment 'startIndex' to consider next substring
               offset = 0;
                                         // Reset 'offset' since we have a new candidate
               // Make sure that 'nextIndex' is always ahead of 'startIndex'
```

// Reset 'offset' because we are comparing a new pair of indices

```
};
TypeScript
function lastSubstring(str: string): string {
   const length = str.length; // Store the length of the string
    let startIndex = 0; // Initialize the starting index of the last substring
```

else {

if (startIndex >= nextIndex) {

nextIndex += offset + 1;

offset = 0;

return s.substr(startIndex);

nextIndex = startIndex + 1;

// as it's the lexicographically largest substring

// Loop to find the last substring in lexicographical order

// If the current character at 'startIndex' + 'offset' is greater,

// Return the substring from 'startIndex' to the end of the string,

for (let currentIndex = 1, offset = 0; currentIndex + offset < length;) {</pre>

if (str[startIndex + offset] === str[currentIndex + offset]) {

// 'startIndex' remains as the candidate and move 'nextIndex' for next comparison

```
// If the characters are the same, increment the offset
              offset++;
          } else if (str[startIndex + offset] < str[currentIndex + offset]) {</pre>
              // Found a later character, update start index beyond the current comparison
              startIndex = currentIndex;
              currentIndex++;
              offset = 0; // Reset the offset for new comparisons
          } else {
              // Current character is not later, just move the current index forward
              currentIndex += offset + 1;
              offset = 0; // Reset the offset for new comparisons
      // Return the substring from the start index to the end of the string
      return str.slice(startIndex);
class Solution:
   def lastSubstring(self, string: str) -> str:
       # Initialize pointer (i) for the start of the current candidate substring.
       # Initialize pointer (j) as the start of the next substring to compare.
       # Initialize (k) as the offset from both i and j during comparison.
        current_start = 0
        compare_start = 1
        offset = 0
```

```
# Iterate until the end of string is reached.
while compare_start + offset < len(string):</pre>
    # If characters at the current offset are equal, increase the offset.
    if string[current_start + offset] == string[compare_start + offset]:
        offset += 1
    # If the current character in the comparison substring is greater,
    # it becomes the new candidate. Update current_start to be compare_start.
    elif string[current_start + offset] < string[compare_start + offset]:</pre>
        current_start = max(current_start + offset + 1, compare_start)
        offset = 0
        # Ensure compare_start is always after current_start.
        if current_start >= compare_start:
            compare_start = current_start + 1
    # If the current character in the candidate substring is greater,
    # continue with the next substring by moving compare_start.
    else:
```

# Return the last substring starting from the candidate position.

size. There are no data structures that grow with the size of the input.

**Time Complexity** 

compare\_start += offset + 1

offset = 0

Time and Space Complexity

return string[current\_start:]

The time complexity of the algorithm is indeed O(n). Here's why: The indices i and j represent the starting positions of the two substrings being compared, and k tracks the current comparing position relative to i and j. The while loop will continue until j + k reaches len(s), which would happen in the worst case after 2n comparisons (when every character in the string is the same), because when s[i + k] is less than s[j + k], i is set to k + 1 steps ahead which could repeat n times in the worst case, and each time j is shifted only one step ahead, also up to n times. Therefore, the algorithm does not compare each character more than twice in the worst-case scenario.

**Space Complexity** 

The space complexity is 0(1) because the algorithm uses a fixed number of integer variables i, j, and k, regardless of the input