<https://leetcode.com/problems/longest-nice-substring/description/>

A string s is **nice** if, for every letter of the alphabet that s contains, it appears **both** in uppercase and lowercase. For example, "abABB" is nice because 'A' and 'a' appear, and 'B' and 'b' appear. However, "abA" is not because 'b' appears, but 'B' does not.

Given a string s, return *the longest****substring****of s that is****nice****. If there are multiple, return the substring of the****earliest****occurrence. If there are none, return an empty string*.

**Example 1:**

**Input:** s = "YazaAay"

**Output:** "aAa"

**Explanation:** "aAa" is a nice string because 'A/a' is the only letter of the alphabet in s, and both 'A' and 'a' appear."aAa" is the longest nice substring.

**Example 2:**

**Input:** s = "Bb"

**Output:** "Bb"

**Explanation:** "Bb" is a nice string because both 'B' and 'b' appear. The whole string is a substring.

**Example 3:**

**Input:** s = "c"

**Output:** ""

**Explanation:** There are no nice substrings.

**Constraints:**

1 <= s.length <= 100

s consists of uppercase and lowercase English letters.

**Attempt 1: 2025-04-06**

**Solution 1: Brute Force (10 min)**

**Style 1:**

class Solution {

public String longestNiceSubstring(String s) {

String result = "";

int n = s.length();

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

for(int j = i + 1; j <= n; j++) {

String str = s.substring(i, j);

if(str.length() > 1 && str.length() > result.length() && checkNice(str)) {

result = str;

}

}

}

return result;

}

private boolean checkNice(String s) {

Set<Character> set = new HashSet<Character>();

for(char c : s.toCharArray()) {

set.add(c);

}

for(char c : s.toCharArray()) {

if(set.contains(Character.toUpperCase(c)) != set.contains(Character.toLowerCase(c))) {

return false;

}

}

return true;

}

}

Time Complexity: O(n^3)

This brute force approach has O(n^3) time complexity because:

O(n^2) for generating all substrings

O(n) for checking each substring

Total: O(n^2) \* O(n) = O(n^3)

**Style 2:**

class Solution {

public String longestNiceSubstring(String s) {

String result = "";

int maxLen = 0;

// Generate all possible substrings

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

for (int j = i + 1; j <= s.length(); j++) {

String substring = s.substring(i, j);

// Check if the substring is nice

if (isNice(substring) && substring.length() > maxLen) {

maxLen = substring.length();

result = substring;

}

}

}

return result;

}

private boolean isNice(String s) {

Set<Character> lower = new HashSet<>();

Set<Character> upper = new HashSet<>();

// Separate lowercase and uppercase letters

for (char c : s.toCharArray()) {

if (Character.isLowerCase(c)) {

lower.add(c);

} else {

upper.add(Character.toLowerCase(c));

}

}

// Check if all lowercase letters have uppercase counterparts

// and vice versa (through the lowercase check)

return lower.equals(upper);

}

}

Time Complexity: O(n^3)

This brute force approach has O(n^3) time complexity because:

O(n^2) for generating all substrings

O(n) for checking each substring

Total: O(n^2) \* O(n) = O(n^3)

**Solution 2: Divide and Conquer (30 min)**

**Style 1:**

class Solution {

public String longestNiceSubstring(String s) {

if (s.length() < 2) return "";

char[] arr = s.toCharArray();

Set<Character> set = new HashSet<>();

for (char c : arr) set.add(c);

for (int i = 0; i < arr.length; i++) {

char c = arr[i];

if (set.contains(Character.toUpperCase(c)) && set.contains(Character.toLowerCase(c))) {

continue;

}

String sub1 = longestNiceSubstring(s.substring(0, i));

String sub2 = longestNiceSubstring(s.substring(i + 1));

return sub1.length() >= sub2.length() ? sub1 : sub2;

}

return s;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Style 2:**

class Solution {

    public String longestNiceSubstring(String s) {

        int[] index = helper(s, 0, s.length());

        return s.substring(index[0], index[1]);

    }

    private int[] helper(String s, int start, int end) {

        Set<Character> set = new HashSet<>();

        for(int i = start; i < end; i++) {

            set.add(s.charAt(i));

        }

        for(int i = start; i < end; i++) {

            if(!set.contains(Character.toLowerCase(s.charAt(i)))

                || !set.contains(Character.toUpperCase(s.charAt(i)))) {

                int[] prefix = helper(s, start, i);

                int[] suffix = helper(s, i + 1, end);

                return prefix[1] - prefix[0] >= suffix[1] - suffix[0] ? prefix : suffix;

            }

        }

        return new int[] {start, end};

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/longest-nice-substring/solutions/1645774/java-why-divide-conquer-explained/>

**Idea:**

Why this is a Divide & Conquer problem?

If you've solved a good variety of problems, your initial approach might be Sliding window technique, which won't work, at least not in its template form

However when you go down that route, you'll see that a string can possbily become nice later but then some character gets into the string which will make it impossible for that sub-string to become nice because that char doesn't exist in the entire string. So we know there's no point in continuing any further.

In fact this character (say z in the given example) will not let any substring become nice in the entire string, so the only way to find a nice substring is to exlude this char, which means you split the string at z, something like [start...]z[...end]

Now you have two sub-problems with the left and right partitions which ask you to find the original LNS problem, which you can do recursively. This is the very definition of [Divide-and-Conquer technique](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) - *A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.*

As long as both upper and lower case chars exists in the string, keep moving, else take the erroneous character as a partition and check for its left and right parts to be nice strings.

s is not a nice string if size(s) < 2

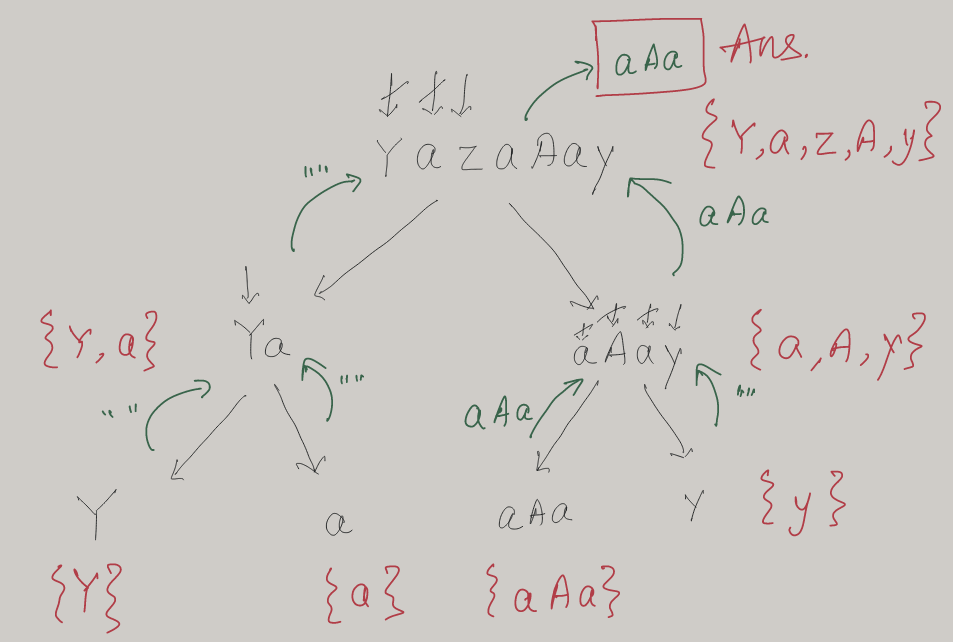
Return the longer nice sub-string from the recursive method

If you'll use > instead of >= then you'll get the nice substring which occurs last, but the question says that in case of a tie, return the earliest one, thus prefix >= suffix

Space complexity:

Size of the set can at max be 52 because allowed characters in s are a-z and A-Z, so the space of set will be O(1)

Space is O(n) because the recursive call stack can be n levels deep



**T/S:** O(n²)/O(n), where n = size(s)

**Refer to Deepseek**

**Brute Force Approach**

The brute force method involves checking all possible substrings of the input string and verifying if each one is "nice" (contains both uppercase and lowercase versions of every letter present).

**Steps:**

Generate all possible substrings of the input string

Check each substring to see if it's nice:

For every lowercase letter in the substring, its uppercase version must also be present

For every uppercase letter in the substring, its lowercase version must also be present

Track the longest nice substring found

**Java Implementation:**

class Solution {

public String longestNiceSubstring(String s) {

String result = "";

int maxLen = 0;

// Generate all possible substrings

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

for (int j = i + 1; j <= s.length(); j++) {

String substring = s.substring(i, j);

// Check if the substring is nice

if (isNice(substring) && substring.length() > maxLen) {

maxLen = substring.length();

result = substring;

}

}

}

return result;

}

private boolean isNice(String s) {

Set<Character> lower = new HashSet<>();

Set<Character> upper = new HashSet<>();

// Separate lowercase and uppercase letters

for (char c : s.toCharArray()) {

if (Character.isLowerCase(c)) {

lower.add(c);

} else {

upper.add(Character.toLowerCase(c));

}

}

// Check if all lowercase letters have uppercase counterparts

// and vice versa (through the lowercase check)

return lower.equals(upper);

}

}

**Explanation:**

Nested Loops: The outer loop sets the starting index, and the inner loop sets the ending index of each substring.

Nice Check: The isNice method:

Creates two sets: one for lowercase letters and one for uppercase letters (converted to lowercase)

Checks if these sets are identical (meaning every lowercase letter has an uppercase counterpart and vice versa)

Tracking Longest: We keep track of the longest nice substring found during the checks.

**Example:**

For the string "YazaAay":

Substring "aAa" is nice because:

Contains 'a' and 'A'

No other letters to check

Substring "azaA" is not nice because:

Contains 'z' but no 'Z'

The longest nice substring is "aAa"

**Time Complexity:**

This brute force approach has O(n³) time complexity because:

O(n²) for generating all substrings

O(n) for checking each substring

Total: O(n²) \* O(n) = O(n³)

This is why the divide and conquer approach (shown in the original solution) is more efficient for longer strings, but the brute force method is easier to understand and works fine for small inputs.

**Divide and Conquer**

A nice substring is a substring where every letter in the substring appears in both uppercase and lowercase. We need to find the longest such substring.

**Approach**

Brute Force Check: For all possible substrings, check if it's nice.

Divide and Conquer: **If we find a character that doesn't have both cases in the string, the nice substring must be either to its left or right.**

**Solution Code**

class Solution {

public String longestNiceSubstring(String s) {

if (s.length() < 2) return "";

char[] arr = s.toCharArray();

Set<Character> set = new HashSet<>();

for (char c : arr) set.add(c);

for (int i = 0; i < arr.length; i++) {

char c = arr[i];

if (set.contains(Character.toUpperCase(c)) && set.contains(Character.toLowerCase(c))) {

continue;

}

String sub1 = longestNiceSubstring(s.substring(0, i));

String sub2 = longestNiceSubstring(s.substring(i + 1));

return sub1.length() >= sub2.length() ? sub1 : sub2;

}

return s;

}

}

**Explanation**

Base Case: If the string length is less than 2, return empty string as no nice substring can exist.

Character Set: Create a set of all characters in the string for quick lookup.

Check for Nice String: For each character, check if both its uppercase and lowercase forms exist in the set.

Divide and Conquer: If a character is found that doesn't have both cases, recursively check the substrings before and after this character.

Return Result: The longest nice substring from either the left or right substring is returned.

This approach efficiently narrows down the problem by eliminating parts of the string that can't contain the longest nice substring, leading to an optimal solution.

**Time Complexity: O(N^2) (Worst Case)**

Divide Step: The algorithm splits the string whenever it finds a character that doesn't have both lowercase and uppercase versions.

Recursive Calls: Each recursive call processes a substring of the original string.

Worst Case: If the string has no "bad" characters (i.e., it's already nice), the algorithm checks all possible splits, leading to O(N²) time.

Example: "aAaAaA" (already nice) → no splits, but we still check every character.

Best Case: If the string has many "bad" characters, the recursion depth is limited, reducing time complexity.

**Space Complexity: O(N) (due to recursion stack)**

The recursion depth can go up to O(N) in the worst case (if the string is split at every position).

Each recursive call uses O(1) extra space (for the Set), but the call stack can grow up to O(N).

**Comparison with Brute Force**

|  |  |  |
| --- | --- | --- |
| Approach | Time Complexity | Space Complexity |
| Brute Force | O(N³) | O(N) |
| Divide & Conquer | O(N²) | O(N) |

**Why Divide & Conquer is Better**

Avoids checking all possible substrings redundantly.

Skips invalid regions quickly by splitting at "bad" characters.

Still O(N²) in the worst case, but much faster in practice for most inputs.

**Optimization Insight**

The worst case (O(N²)) occurs when the input is already nice (no splits happen).

However, for typical inputs with some "bad" characters, the runtime is much better than brute force.

Would you like an example walkthrough of how the recursion splits the string?

**Refer to**

[L2421.Number of Good Paths (Ref.L2506)](note://WEBe0a7e9243acd024b92c4cd8a899fb0d3)