Solution: Minimum Window Subsequence

Let's solve the Minimum Window Subsequence problem using the Sliding Window pattern.

We'll cover the following

Statement

Given two strings, `str1` and `str2`, find the shortest substring in `str1` such that `str2` is a subsequence of that substring.

A substring is defined as a contiguous sequence of characters within a string. A subsequence is a sequence that can be derived from another sequence by deleting zero or more elements without changing the order of the remaining elements.

Let's say you have the following two strings:

`str1` = "$abbcb$"

`str2` = "ac"

In this example, "$abbc$" is a substring of `str1`, from which we can derive `str2` simply by deleting both the instances of the character b. Therefore, `str2` is a subsequence of this substring. Since this substring is the shortest among all the substrings in which `str2` is present as a subsequence, the function should return this substring, that is, "$abbc$".

> If there is no substring in `str1` that covers all characters in `str2`, return an empty string.

> If there are multiple minimum-length substrings that meet the subsequence requirement, return the one with the left-most starting index.

Constraints:

- $1 \leq$ `str1.length` $\leq 2 \times 10^3$
- $1 \leq$ `str2.length` ≤ 100
- `str1` and `str2` consist of uppercase and lowercase English letters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach would be to generate all possible substrings of `str1` and then check which substrings contain `str2` as a subsequence. Out of all the substrings in `str1` that contain `str2` as a subsequence, we'll choose the one with the shortest length. Now, let's look at the cost of this solution. We need two nested loops to get all possible substrings and another loop to check whether each substring contains all the required characters. This brings the time complexity to $O(n^3)$. Since we're not using any extra space, the space complexity is $O(1)$.

Optimized approach using sliding window

To eliminate the extra traversals of the substrings, we use the sliding window approach. With this approach, we only consider the substrings that we are sure contain all the characters of `str2` in the same order. This problem can be conveniently solved using the sliding window pattern. The idea is to keep track of whether the subsequence has been found or not and to select the shortest subsequence from `str1`.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step construction

The first step of the solution is to initialize the variables. We begin by creating two variables, `sizeStr1` and `sizeStr2`, to store the lengths of `str1` and `str2`, respectively. We then initialize `minSubLen` to infinity, which will be used to store the length of the minimum subsequence.

To help us traverse the two strings, we create two indexes, `indexS1` and `indexS2`, which initially point to the first characters of `str1` and `str2`, respectively. These indexes will be incremented as we scan through the strings to find the subsequence.

Finally, we initialize `minSubsequence` to an empty string. This variable will store the output, which is the smallest possible subsequence.

Java

```java
class MinSubsequence {

    public static int[] minWindow(String str1, String str2) {

        // Save the size of str1 and str2
        int sizeStr1 = str1.length();
        int sizeStr2 = str2.length();
        // Initialize minSubLen to a very large number (infinity)
        float minSubLen = Float.POSITIVE_INFINITY;
        // Initialize pointers to zero and the minSubsequence to an empty string
        int indexS1 = 0;
        int indexS2 = 0;
        String minSubsequence = "";
        int[] arr = new int[2];
        arr[0] = sizeStr1;
        arr[1] = sizeStr2;
        return arr;
    }

    // Driver code
```

```
    public static void main(String[] args) {
        String[] str1 = {"abcdebdde", "fgrqsqsnodwmxzkzxwqegkndaa", "zxcvnhss", "alpha", "beta"};
        String[] str2 = {"bde", "kzed", "css", "la", "ab"};

        for (int i = 0; i < str1.length; i++) {
          int[] result = minWindow(str1[i], str2[i]);
          System.out.println((i + 1) + ".\t Input String: " + "(" + str1[i] + ", " + str2[i] + ")");
```

Initializing variables

Once we've initialized the variables, we start looping over `str1` using `indexS1`. In each iteration, if the current character of `str1`, pointed at by `indexS1`, and the current character of `str2`, pointed at by `indexS2`, are the same, we increment both the indexes. Otherwise, we only increment `indexS1`. Using this logic, `indexS2` will reach the end of `str2` only if each character of `str2` is found in `str1`. At this point, we have found a potential minimum window subsequence, i.e., a substring that contains `str2` as a subsequence. So, we set `minSubLen` to the length of this substring and `minSubsequence` to this substring.

Now, we need to check the rest of `str1` for a shorter substring that meets our requirement. So, we resume our search in `str1` from `indexS1` $+1$. Whenever we find a substring that meets our requirement, we compare its length with `minSubLen` and if it is shorter, we update `minSubLen` with the length of this substring and `minSubsequence` with this substring. Finally, when we have traversed the entire `str1`, we return the minimum window subsequence.

Let's take a look at the code of this solution:

Java

```
1  class MinSubsequence {
2
3      public static String minWindow(String str1, String str2) {
4
5          // Save the size of str1 and str2
6          int sizeStr1 = str1.length();
7          int sizeStr2 = str2.length();
8
9          // Initialize 'minSubLen' to a very large number (infinity)
10         float minSubLen = Float.POSITIVE_INFINITY;
11
12         // Initialize pointers to zero and the minSubsequence to an empty string
13         int indexS1 = 0;
14         int indexS2 = 0;
15         int start = 0, end = 0;
16         String minSubsequence = "";
17
18         // Process every character of str1
19         while (indexS1 < sizeStr1) {
20           // Check if the character pointed by indexS1 in str1
21           // is the same as the character pointed by indexS2 in
22           if (str1.charAt(indexS1) == str2.charAt(indexS2)) {
23             // if this was the first character of str2, mark it as the start of the substring
24             if (indexS2 == 0) {
25               start = indexS1;
26             }
27             // if the pointed character is the same in both strings increment index_s2
```

First draft of the solution

By this stage, we've found a potential solution to the problem but if we uncomment the test case provided in **lines 64 - 70** in the code widget above, we'll notice that our solution does not return the correct minimum window subsequence. The issue with our solution is that it skips over any potential minimum window subsequences that start before the last character of the substring that we just found. Let's understand this with the help of the example:

`str1` = "abcdedeaqdweq"

`str2` = "adeq"

Here, the minimum window subsequence is "aqdweq" that starts at index 7 and ends at index 12. The first potential minimum window subsequence that our code identifies is "abcdedeaq", ending at 8. If we resume the search from the end of this substring, i.e., from index 9 in `str1`, we won't be able to locate the shortest substring that satisfies our condition, since we skipped over index 7, which is the starting index of the actual minimum window subsequence.

To fix this, we simply change the index from which we resume our search and continue the iteration from the *second* character of the current substring, instead of `indexS1` $+1$.

Let's look at the code of this solution after this correction, seen in **line 41**:

Java

```java
1  class MinSubsequence {
2
3      public static String minWindow(String str1, String str2) {
4
5          // Save the size of str1 and str2
6          int sizeStr1 = str1.length();
7          int sizeStr2 = str2.length();
8          // Initialize 'minSubLen' to a very large number (infinity)
9          float minSubLen = Float.POSITIVE_INFINITY;
10         // Initialize pointers to zero and the minSubsequence to an empty string
11         int indexS1 = 0;
12         int indexS2 = 0;
13         int start = 0, end = 0;
14         String minSubsequence = "";
15
16         // Process every character of str1
17         while (indexS1 < sizeStr1) {
18           // check if the character pointed by indexS1 in str1
19           // is the same as the character pointed by indexS2 in str2
20           if (str1.charAt(indexS1) == str2.charAt(indexS2)) {
21             // if this was the first character of str2, mark it as the start of the substring
22             if (indexS2 == 0) {
23               start = indexS1;
24             }
25             // if the pointed character is the same in both strings increment indexS2
26             indexS2 += 1;
27
28             // check if indexS2 has reached the end of str2
```

Corrected solution

At this point, our solution is correct and complete, since we can see that the example where `str1` = "abcdedeaqdweq" and `str2` = "adeq", provided as the first test case in the coding widget above, give the correct output.

However, if we uncomment the test case given in **lines 58 - 64**, we'll notice that our solution times out, which means that our solution is inefficient. Let's first understand our test case. The first string, `str1`, consists of $10,000$ occurrences of the letter "f", $9,000$ occurrences of the letter "s", 500 occurrences of the letter "e", followed by one occurrence of the letter "a". The second string, `str2` is "fffessa".

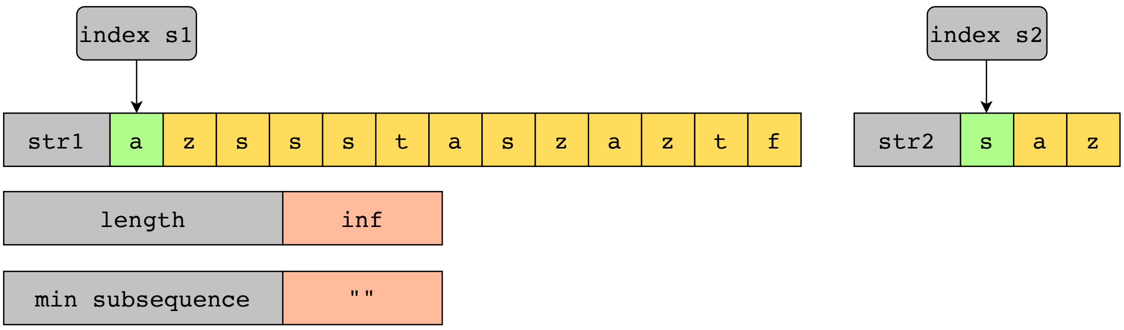
Our solution identifies all the potential minimum window subsequences starting from every instance of the character f in `str1`. The upper bound on the number of such subsequences in this case is $10000 \times 9000 \times 500$. This is obviously wasteful, since we can see from inspection that we can discard all but the last three occurrences of the letter “f”. Let’s see how we can improve our solution based on this insight.

After finding a potential subsequence, let’s try to shrink it by moving backward in `str1`, comparing it one character at a time with `str2`. We initialize two more pointer variables, `start` and `end`, to `indexS1` and point `indexS2` to the last character of `str2`. Then, we iterate backward over the potential minimum window subsequence using the `start` pointer. When we have discovered the entire `str2` in the potential minimum window subsequence in the reverse order, we update the `minSubLen` and `minSubsequence` variables accordingly.

At this point, we have found the shortest window subsequence that meets our requirement up to the index `end` in `str1`. So, now, we need to check the rest of the `str1` string. We simply resume iterating over `str1` in the forward direction, starting, as before, from `start` $+1$.

Let’s visualize the solution:

The **length** variable will store the length of the minimum window subsequence, which is initialized to **infinity**. The **min subsequence** variable will store the minimum window subsequence. Each index initially points to the first characters of **str1** and **str2**.
As the characters pointed by **index s1** and **index s2** are not the same, we simply move on to the next element of **str1**.



1 of 28

Java

```
class MinSubsequence {

    public static String minWindow(String str1, String str2) {

        // Save the size of str1 and str2
        int sizeStr1 = str1.length();
        int sizeStr2 = str2.length();

        // Initialize length to a very large number (infinity)
        float minSubLen = Float.POSITIVE_INFINITY;

        // Initialize pointers to zero and the minSubsequence to an empty string
        int indexS1 = 0;
        int indexS2 = 0;
        int start = 0, end = 0;
        String minSubsequence = "";

        // Process every character of str1
        while (indexS1 < sizeStr1) {
          // Check if the character pointed by indexS1 in str1
          // is the same as the character pointed by indexS2 in
```

```
        if (str1.charAt(indexS1) == str2.charAt(indexS2)) {
          // If the pointed character is the same
          // in both strings increment indexS2
          indexS2 += 1;
          // Check if indexS2 has reached the end of str2
          if (indexS2 == sizeStr2) {
            // At this point the str1 contains all characters of str2
```

Optimized solution

Just the code

Here's the complete solution to this problem:

Java

```java
class MinSubsequence {

    public static String minWindow(String str1, String str2) {
        int sizeStr1 = str1.length();
        int sizeStr2 = str2.length();
        float length = Float.POSITIVE_INFINITY;
        int indexS1 = 0;
        int indexS2 = 0;
        int start = 0,
        end = 0;
        String minSubsequence = "";
        while (indexS1 < sizeStr1) {
          if (str1.charAt(indexS1) == str2.charAt(indexS2)) {
            indexS2 += 1;
            if (indexS2 == sizeStr2) {
              start = indexS1;
              end = indexS1 + 1;
              indexS2 -= 1;
              while (indexS2 >= 0) {
                if (str1.charAt(start) == str2.charAt(indexS2)) {
                  indexS2 -= 1;
                }
                start -= 1;
              }
              start += 1;
              if ((end - start) < length) {
                length = end - start;
                minSubsequence = str1.substring(start, end);
```

Minimum Window Subsequence

Solution summary

1. Initialize two indexes, `indexS1` and `indexS2`, to zero for iterating both strings.
2. If the character pointed by `indexS1` in `str1` is the same as the character pointed by `indexS2` in `str2`, increment both pointers. Otherwise, only increment `indexS1`.
3. Once `indexS2` reaches the end of `str2`, initialize two new indexes (`start` and `end`). With these two indexes, we'll slide the window backward.
4. Set `start` and `end` to `indexS1`.

6. Once, `str2` has been discovered in `str1` in the backward direction, calculate the length of the substring.
7. If this length is less than the current minimum length, update the `minSubLen` variable and the `minSubsequence` string.
8. Resume the search in the forward direction from `start` $+1$ in `str1`.

9. Repeat until `indexS1` reaches the end of `str1`.

Time complexity

The outer loop iterates over the string `str1`, so the time complexity of this loop will be $O(n)$, where n is the length of string `str1`. Inside this loop, there is a `while` loop that is used to iterate back over the window once all the characters of `str2` have been found in the current window. The time complexity of this loop will be $O(m)$, where m is the length of string `str2`. Therefore, the overall time complexity of this solution is $O(n \times m)$. For example, when `str1` = “aaaaa” and `str2` = “aa”, it takes $O(n \times m)$ time.

Space complexity

Since we are not using any extra space apart from a few variables, the space complexity is $O(1)$.

← Back

Minimum Window Su...

Next →

Minimum Window Su...

Mark as Completed