727. Minimum Window Subsequence

Nov. 11, 2017 | 27.4K views

Average Rating: 1.53 (117 votes)

Given strings `S` and `T`, find the minimum (contiguous) **substring** `W` of `S`, so that `T` is a **subsequence** of `W`.

If there is no such window in `S` that covers all characters in `T`, return the empty string `""`. If there are multiple such minimum-length windows, return the one with the left-most starting index.

Example 1:

```
Input:
S = "abcdebdde", T = "bde"
Output: "bcde"
Explanation:
"bcde" is the answer because it occurs before "bdde" which has the same length.
"deb" is not a smaller window because the elements of T in the window must occur in or
```

Note:

- All the strings in the input will only contain lowercase letters.
- The length of `S` will be in the range `[1, 20000]`.
- The length of `T` will be in the range `[1, 100]`.

Approach #1: Dynamic Programming (Postfix Variation) [Accepted]

Intuition

Let's work on a simpler problem: `T = 'ab'`. Whenever we find some `'b'` in `S`, we look for the most recent `'a'` that occurred before it, and that forms a candidate window `'a' = S[i], ..., S[j] = 'b'`.

A weak solution to that problem would be to just search for `'a'` every time we find a `'b'`. With a string like `'abbb...bb'` this would be inefficient. A better approach is to remember the last `'a'` seen. Then when we see a `'b'`, we know the start of the window is where we last saw `'a'`, and the end of the window is where we are now.

How can we extend this approach to say, `T = 'abc'`? Whenever we find some `'c'` in `S`, such as `S[k] = 'c'`, we can remember the most recent window that ended at `'b'`, let's say `[i, j]`. Then our candidate window (that is, the smallest possible window ending at `k`) would be `[i, k]`.

Our approach in general works this way. We add characters to `T` one at a time, and for every `S[k] = T[-1]` we always remember the length of the candidate window ending at `k`. We can calculate this using knowledge of the length of the previous window (so we'll need to remember the last window seen). This leads to a dynamic programming solution.

Algorithm

As we iterate through `T[j]`, let's maintain `cur[e] = s` as the largest index such that `T[:j]` is a subsequence of `S[s: e+1]`, (or `-1` if impossible.) Now we want to find `new`, the largest indexes for `T[:j+1]`.

To update our knowledge as `j += 1`, if `S[i] == T[j]`, then `last` (the largest `s` we have seen so far) represents a new valid window `[s, i]`.

In Python, we use `cur` and `new`, while in Java we use `dp[j]` and `dp[~j]` to keep track of the last two rows of our dynamic programming.

At the end, we look at all the windows we have and choose the best.

Java Python Copy

```python
class Solution(object):
    def minWindow(self, S, T):
        cur = [i if x == T[0] else None
               for i, x in enumerate(S)]
        #At time j when considering T[:j+1],
        #the smallest window [s, e] where S[e] == T[j]
        #is represented by cur[e] = s.
        for j in xrange(1, len(T)):
            last = None
            new = [None] * len(S)
            #Now we would like to calculate the candidate windows
            #"new" for T[:j+1].  'last' is the last window seen.
            for i, u in enumerate(S):
                if last is not None and u == T[j]: new[i] = last
                if cur[i] is not None: last = cur[i]
            cur = new

        #Looking at the window data cur, choose the smallest length
        #window [s, e].
        ans = 0, len(S)
        for e, s in enumerate(cur):
            if s >= 0 and e - s < ans[1] - ans[0]:
                ans = s, e
        return S[ans[0]: ans[1]+1] if ans[1] < len(S) else ""
```

Complexity Analysis

- Time Complexity: $O(ST)$, where S, T are the lengths of `S, T`. We have two for-loops.
- Space Complexity: $O(S)$, the length of `dp`.

Approach #2: Dynamic Programming (Next Array Variation) [Accepted]

Intuition

Let's guess that the minimum window will start at `S[i]`. We can assume that `S[i] = T[0]`. Then, we should find the next occurrence of `T[1]` in `S[i+1:]`, say at `S[j]`. Then, we should find the next occurrence of `T[2]` in `S[j+1:]`, and so on.

Finding the next occurrence can be precomputed in linear time so that each guess becomes $O(T)$ work.

Algorithm

We can precompute (for each `i` and `letter`), `next[i][letter]`: the index of the first occurrence of `letter` in `S[i:]`, or `-1` if it is not found.

Then, we'll maintain a set of minimum windows for `T[:j]` as `j` increases. At the end, we'll take the best minimum window.

Java Python Copy

```python
class Solution(object):
    def minWindow(self, S, T):
        N = len(S)
        nxt = [None] * N
        last = [-1] * 26
        for i in xrange(N-1, -1, -1):
            last[ord(S[i]) - ord('a')] = i
            nxt[i] = tuple(last)

        windows = [[i, i] for i, c in enumerate(S) if c == T[0]]
        for j in xrange(1, len(T)):
            letter_index = ord(T[j]) - ord('a')
            windows = [[root, nxt[i+1][letter_index]]
                       for root, i in windows
                       if 0 <= i < N-1 and nxt[i+1][letter_index] >= 0]

        if not windows: return ""
        i, j = min(windows, key = lambda (i, j): j-i)
        return S[i: j+1]
```

Complexity Analysis

- Time Complexity: $O(ST)$, where S, T are the lengths of `S, T`, and assuming a fixed-sized alphabet. The precomputation of `nxt` is $O(S)$, and the other work happens in two for-loops.
- Space Complexity: $O(S)$, the size of `windows`.

Analysis written by: @awice. Approach #1 inspired by @zestypanda.

Rate this article:

Previous Next

Comments: 29

Sort By

Type comment here... (Markdown is supported)

Preview Post

sasha24 ★ 36 July 2, 2018 6:02 PM

The code is difficult to follow

36 Share Reply

Anttthea ★ 29 January 19, 2018 1:09 PM

Sorry, really low quality article

29 Share Reply

SHOW 5 REPLIES

crazygeek ★ 13 March 4, 2019 6:49 PM

Approach 2 is much easier to understand

3 Share Reply

LeonCheng ★ 262 November 12, 2017 10:46 AM

I saw your articles before, and it always explained things quite clear, but this time I am not sure what this article is talking about, hope you could explain it more thoroughly.

2 Share Reply

Galileo_Galilei ★ 437 February 7, 2019 8:31 AM

@awice Could you please explain, why we can break in lines below?

```
window[0] = window[1] = -1;
break;
```

Read More

1 Share Reply

SHOW 2 REPLIES

hello_world_cn ★ 240 October 3, 2018 5:40 AM

Should add the naive approach which is easier to follow but not space-efficient: https://leetcode.com/problems/minimum-window-subsequence/discuss/109362/Java-Super-Easy-DP-Solution-(O(mn))

1 Share Reply

SHOW 1 REPLY

elliscopef ★ 1 February 28, 2018 12:40 PM

class Solution {
public String minWindow(String S, String T) {
if(S==null || T==null) return null;
int head = 0;
int tail = 0;

Read More

1 Share Reply

Yuandong-Chen ★ 91 July 22, 2019 7:42 AM

Greedy O(26S) Time Complexity, O(26S) Space Time Complexity here, Better than official solution: https://leetcode.com/problems/minimum-window-subsequence/discuss/340667/Better-than-Official-Solution-Greedy-O(26S)-Time-Complexity-O(26S)-Space-Complexity
I think O(ST) is not so good if S*T = 2000000 at most, too large.

0 Share Reply

control_eight ★ 1 July 18, 2019 1:03 AM

Time Complexity: O(ST), Space Complexity: O(T)

```
public String minWindow(String s, String t) {
    int[] empty = {0, 0, 0};
```

Read More

0 Share Reply

icoder ★ 0 January 22, 2019 9:43 PM

Doesn't a brute force solution also have O(ST) time complexity, and O(1) space complexity? Do we really need DP for this problem?

0 Share Reply

SHOW 2 REPLIES