<https://leetcode.com/problems/is-subsequence/description/>

Given two strings s and t, return true if s is a subsequence of t, or false otherwise.

A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

**Example 1:**

**Input:** s = "abc", t = "ahbgdc"

**Output:** true

**Example 2:**

**Input:** s = "axc", t = "ahbgdc"

**Output:** false

**Constraints:**

0 <= s.length <= 100

0 <= t.length <= 10^4

s and t consist only of lowercase English letters.

**Follow up:**

Suppose there are lots of incoming s, say s1, s2, ..., sk where k >= 10^9, and you want to check one by one to see if t has its subsequence. In this scenario, how would you change your code?

**Attempt 1: 2024-12-31**

**Solution 1: Binary Search + Hash Table (30 min)**

class Solution {

    public boolean isSubsequence(String s, String t) {

        // {k, v} -> {char in t, indexes of this char in t}

        Map<Character, List<Integer>> t\_char\_indexes = new HashMap<>();

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

            t\_char\_indexes.putIfAbsent(t.charAt(i), new ArrayList<>());

            t\_char\_indexes.get(t.charAt(i)).add(i);

        }

        int prev\_char\_last\_index = -1;

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

            List<Integer> indexes = t\_char\_indexes.get(s.charAt(i));

            // Must check if current char exist in 't'

            // Test out: s = "axc", t = "ahbgdc"

            if(indexes == null) {

                return false;

            }

            // Use binary search to find current char's smallest new index

            // which should greater than previous char's last index

            int curr\_char\_new\_index = binarySearch(indexes, prev\_char\_last\_index);

            // If no valid index is found for current char, 's' is not

            // a subsequence for 't'

            if(curr\_char\_new\_index == -1) {

                return false;

            }

            // Update previous char's last index to the found index

            // of current char's smallest new index

            prev\_char\_last\_index = curr\_char\_new\_index;

        }

        return true;

    }

    // Find lower boundary

    private int binarySearch(List<Integer> indexes, int prev\_char\_last\_index) {

        int lo = 0;

        int hi = indexes.size() - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Strictly '>' since bottom line is new index one more

            // larger than given index

            if(indexes.get(mid) > prev\_char\_last\_index) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // If 'lo' is within bounds, return the index, otherwise, return -1

        if(lo >= indexes.size()) {

            return -1;

        }

        return indexes.get(lo);

    }

}

Time Complexity:

Preprocessing: O(t), where t is the length of t, to build the character index array.

Checking s: For each character in s, binary search over the list of indices for that character takes O(log⁡m), where m is the average number of occurrences of a character in t.

Total: O(s \* log⁡m), where s is the length of s.

Overall: O(t + s \* log⁡m).

Space Complexity: O(t + 26), where t is the length of t for storing character indices, and 26 is the size of the alphabet.

**Refer to chatGPT**

class Solution {

public boolean isSubsequence(String s, String t) {

// Preprocess `t` to store indices of each character

List<Integer>[] charIndices = new ArrayList[26];

for (int i = 0; i < 26; i++) {

charIndices[i] = new ArrayList<>();

}

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

charIndices[t.charAt(i) - 'a'].add(i);

}

// Check if `s` is a subsequence of `t`

int prevIndex = -1; // Tracks the previous matched index in `t`

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

List<Integer> indices = charIndices[c - 'a'];

// Use binary search to find the smallest index greater than `prevIndex`

int nextIndex = findNextIndex(indices, prevIndex);

if (nextIndex == -1) {

// If no valid index is found, `s` is not a subsequence

return false;

}

prevIndex = nextIndex; // Update `prevIndex` to the found index

}

return true; // All characters in `s` matched in order

}

// Helper function to perform binary search for the next valid index

private int findNextIndex(List<Integer> indices, int prevIndex) {

int left = 0, right = indices.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (indices.get(mid) > prevIndex) {

right = mid - 1;

} else {

left = mid + 1;

}

}

// If `left` is within bounds, return the index; otherwise, return -1

return left < indices.size() ? indices.get(left) : -1;

}

}

**Explanation**

Preprocessing **t:**

Create an array of lists (charIndices) where each list stores the indices of a specific character in t. This allows efficient lookup of where each character appears in t.

Binary Search for Subsequence Check:

For each character in s, use the helper function findNextIndex to perform a binary search over the indices of the current character in t.

findNextIndex returns the smallest index in t that is greater than prevIndex. If no such index exists, return false.

Updating **prevIndex:**

Update prevIndex to the found index after successfully matching each character in s.

Early Termination:

If any character in s cannot be matched, return false immediately.

**Complexity**

**Time Complexity:**

Preprocessing: O(t), where t is the length of t, to build the character index array.

Checking **s:**

For each character in s, binary search over the list of indices for that character takes O(log⁡m), where mmm is the average number of occurrences of a character in t.

Total: O(s ⋅ log⁡m), where s is the length of s.

Overall: O(t + s ⋅ log⁡m).

**Space Complexity:**

O(t + 26), where t is the length of t for storing character indices, and 26 is the size of the alphabet.

**Refer to**

<https://leetcode.com/problems/is-subsequence/solutions/87302/binary-search-solution-for-follow-up-with-detailed-comments/>

<https://leetcode.com/problems/is-subsequence/solutions/87302/binary-search-solution-for-follow-up-with-detailed-comments/comments/92266>

**Binary search:**

record the indexes for each character in t, if s[i] matches t[j], then s[i+1] should match a character in t with index bigger than j. This can be reduced to find the first element larger than a value in an sorted array (find upper bound), which can be achieved using binary search.

**Trie:**

For example, if s1 has been matched, s1[last char] matches t[j]. Now, s2 comes, if s1 is a prefix of s2, i.e., s1 == s2.substr[0, i-1], we can start match s2 from s2[i], right?

So, the idea is to create a Trie for all string that have been matched so far. At a node, we record the position in t which matches this char represented by the node. Now, for an incoming string s, we first search the longest prefix in the Trie, find the matching position of the last prefix-char in t, say j. Then, we can start matching the first non-prefix-char of s from j+1.

Now, if we have done the preprocessing as stated in the binary search approach, we can be even faster.

// Follow-up: O(N) time for pre-processing, O(Mlog?) for each S.

// Eg-1. s="abc", t="bahbgdca"

// idx=[a={1,7}, b={0,3}, c={6}]

// i=0 ('a'): prev=1

// i=1 ('b'): prev=3

// i=2 ('c'): prev=6 (return true)

// Eg-2. s="abc", t="bahgdcb"

// idx=[a={1}, b={0,6}, c={5}]

// i=0 ('a'): prev=1

// i=1 ('b'): prev=6

// i=2 ('c'): prev=? (return false)

public boolean isSubsequence(String s, String t) {

List<Integer>[] idx = new List[256]; // Just for clarity

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

if (idx[t.charAt(i)] == null)

idx[t.charAt(i)] = new ArrayList<>();

idx[t.charAt(i)].add(i);

}

int prev = 0;

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

if (idx[s.charAt(i)] == null) return false; // Note: char of S does NOT exist in T causing NPE

int j = Collections.binarySearch(idx[s.charAt(i)], prev);

if (j < 0) j = -j - 1;

if (j == idx[s.charAt(i)].size()) return false;

prev = idx[s.charAt(i)].get(j) + 1;

}

return true;

}

**Solution 2: Two Pointers (30 min)**

class Solution {

    public boolean isSubsequence(String s, String t) {

        if(s.equals("")) {

            return true;

        }

        int i = 0;

        int j = 0;

        while(j < t.length()) {

            if(s.charAt(i) == t.charAt(j)) {

                i++;

                if(i == s.length()) {

                    return true;

                }

            }

            j++;

        }

        return false;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/is-subsequence/solutions/87254/straight-forward-java-simple-solution/>

Just use two pointers:

public class Solution {

public boolean isSubsequence(String s, String t) {

if (s.length() == 0) return true;

int indexS = 0, indexT = 0;

while (indexT < t.length()) {

if (t.charAt(indexT) == s.charAt(indexS)) {

indexS++;

if (indexS == s.length()) return true;

}

indexT++;

}

return false;

}

}

**Refer to**

[L792.Number of Matching Subsequences (Ref.L392,L2062)](note://WEB9e9521022575941e94e9a64861e1a74b)