Previous Next

383. Ransom Note

March 15, 2020 | 17.5K views

Average Rating: 4.89 (9 votes)

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

Example 1:

```
Input: ransomNote = "a", magazine = "b"
Output: false
```

Example 2:

```
Input: ransomNote = "aa", magazine = "ab"
Output: false
```

Example 3:

```
Input: ransomNote = "aa", magazine = "aab"
Output: true
```

Constraints:

- You may assume that both strings contain only lowercase letters.

Solution

Something you might notice when you run code for this problem here on Leetcode is that *Approach 1 passes, and is the fastest*. This is because all the testcases are very small. For huge test cases though, the other approaches would beat it, and Approach 1 would be far too slow.

In an interview, it's *unlikely that Approach 1 would be sufficient to get you the job*. Interviewers will expect to see an optimized approach such as Approach 2 or 3.

Approach 1: Simulation

Intuition

To create our ransom note, for every character we have in the note, we need to take a copy of that character out of the magazine so that it can go into the note.

If a character we need isn't in the magazine, then we should stop and return `False`. Otherwise, if we manage to get all the characters we need to complete the note, then we should return `True`.

```
For each char in ransomNote:
    Find that letter in magazine.
    If it is in magazine:
        Remove it from magazine.
    Else:
        Return False
Return True
```

Note that there's no need to *explicitly* build up the ransom note; we only need to return whether or not it's possible. This can be determined simply by removing the characters we need from the magazine.

This is the most straightforward approach, but as we'll see soon, although it does pass here on Leetcode, it's not very efficient and is not likely to get you a job at a top company.

Algorithm

Strings are an **immutable** type. This means that they can't be modified, and so don't have "insert" and "delete" operations. For this reason, we instead need to repeatedly replace the magazine with a new String, that doesn't have the character we wanted to remove.

Java Python Copy

```python
def canConstruct(self, ransomNote: str, magazine: str) -> bool:
    # For each character, c,  in the ransom note.
    for c in ransomNote:
        # If there are none of c left in the String, return False.
        if c not in magazine:
            return False
        # Find the index of the first occurrence of c in the magazine.
        location = magazine.index(c)
        # Use splicing to make a new string with the characters
        # before "location" (but not including), and the characters
        #after "location".
        magazine = magazine[:location] + magazine[location + 1:]
    # If we got this far, we can successfully build the note.
    return True
```

Complexity Analysis

We'll say m is the length of the **m**agazine, and n is the length of the ransom **n**ote.

- Time Complexity : $O(m \cdot n)$.

  Finding the letter we need in the magazine has a cost of $O(m)$. This is because we need to perform a linear search of the magazine. Removing the letter we need from the magazine is *also* $O(m)$. This is because we need to make a new string to represent it. $O(m) + O(m) = O(2 \cdot m) = O(m)$ because we drop constants in big-o analysis.

  So, how many times are we performing this $O(m)$ operation? Well, we are looping through each of the n characters in the ransom note and performing it once for each letter. This is a total of n times, and so we get $n \cdot O(m) = O(m \cdot n)$.

- Space Complexity : $O(m)$.

  Creating a new magazine with one letter less requires auxillary space the length of the magazine; $O(m)$.

Approach 2: Two HashMaps

Intuition

Remember that we decided the length of the ransom **n**ote is n, and the length of the **m**agazine is m.

In an interview, you might start by describing the previous approach and determining its time complexity, but not actually implementing it. Your next goal would be to reason carefully about the implementation and its time complexity, to identify parts that could be made more efficient.

Removing the n factor from the time complexity is going to be impossible, because we need to at least look at each character in the ransom note. Otherwise, how could we possibly know whether or not we have the characters we need to make it? We might be able to avoid the need for an $O(m)$ operation for every one of the n characters in the ransom note though.

As an example, notice that if there's three `'a'` 's in the ransom note, then there needs to be *at least* three `'a'` s in the magazine. This should be fairly intuitive, as you'd encounter it if trying to make a note out of a magazine for real. The same idea applies for all the other unique characters too.

Therefore, a better way of solving the problem would be to count up how many of each letter are in both the magazine and the ransom note. We can represent the counts with a `HashMap` that has characters as keys, and counts as values. For example, the string `"leetcode is cool"` is represented as follows.

s	e	l	c	d	i	o	t
1	3	2	2	1	1	3	1

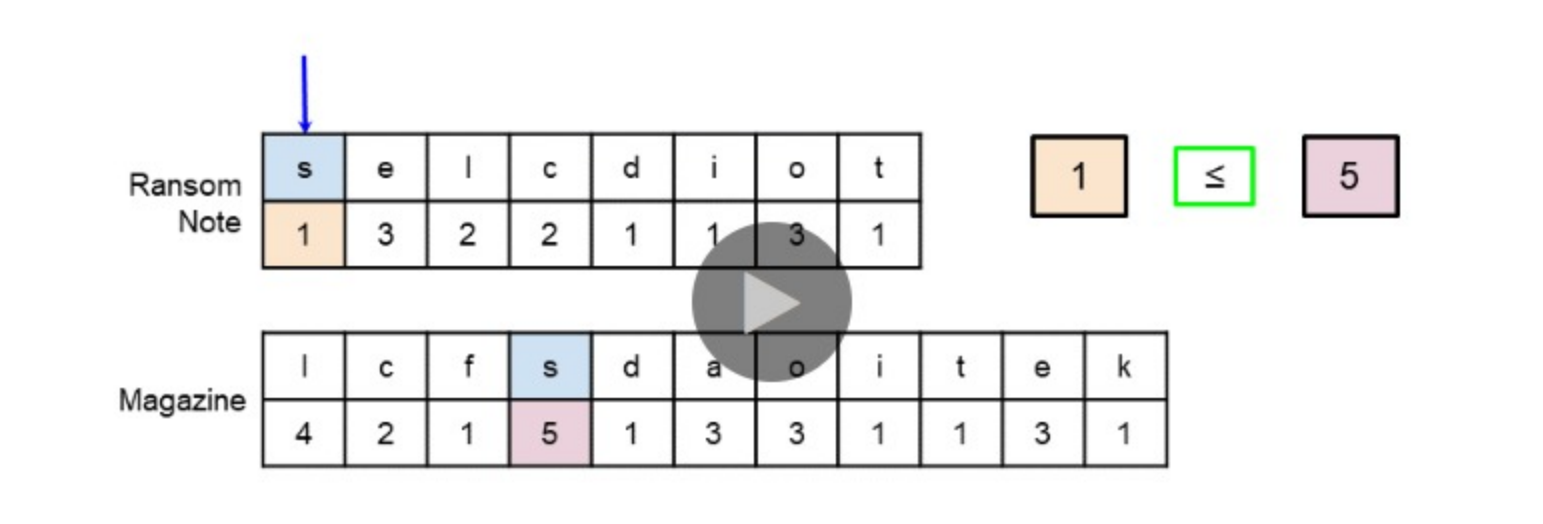
We can make two `HashMap` s; one for the magazine, and the other for the ransom note. Here is the pseudocode for making one of these "counts" `HashMap` s.

```
define function makeCountsMap(string):
    counts = a new HashMap
    for each char in string:
        if char not in counts:
            counts.put(char, 1)
        else:
            old_count = counts.get(char)
            counts.put(char, old_count + 1)
    return counts
```

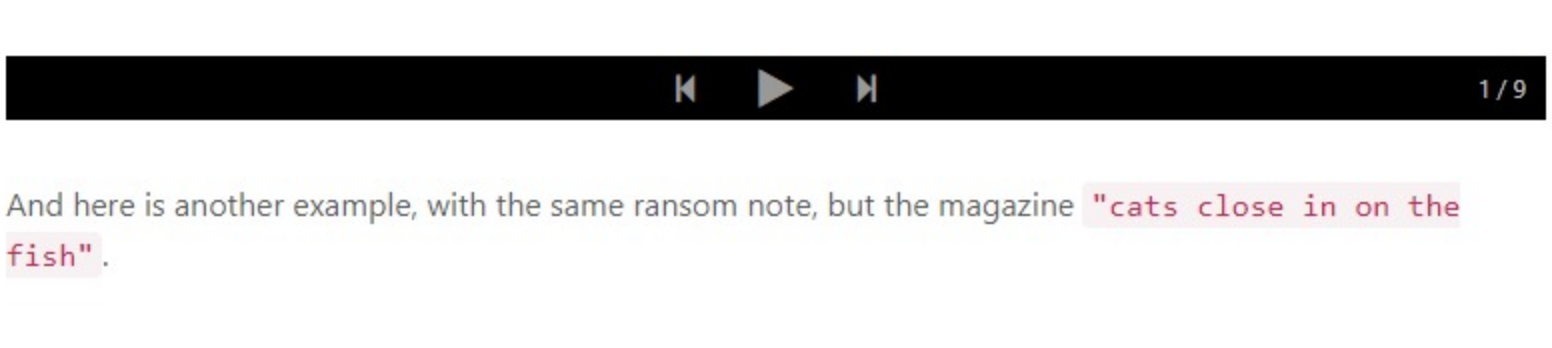
Then, to actually check whether or not the ransom note can be made using the magazine, we should loop over each character of the ransom note, checking how many of it we need, and checking that at least that many exist in the magazine, by looking it up in the magazine `HashMap`. We need to be careful of the case where the character we need isn't in the magazine *at all*; in this case we should return `False` as the number of them in the magazine is definitely smaller than the number we need. If we manage to check all the characters without `False` being returned, then we know that we must have had enough characters to complete the note, and can therefore return `True`. Here is some pseudocode for that algorithm.

```
noteCounts = makeCountsMap(ransomNote)
magazineCounts = makeCountsMap(magazine)
for each (char, count) in noteCounts:
    if char is not in magazineCounts:
        return False
    countInMagazine = magazineCounts.get(char)
    if countInMagazine < count:
        return False
return True
```

Here is an animation showing the above algorithm in action with the ransom note `"leetcode is cool"` and the magazine `"close call as fools take sides"`.



And here is another example, with the same ransom note, but the magazine `"cats close in on the fish"`.



There's one more optimization we can make. Notice that if the length of the ransom note is *longer* than the length of the magazine, then its impossible for there to be enough characters in the magazine.

Algorithm

Java Python Copy

```python
def canConstruct(self, ransomNote: str, magazine: str) -> bool:

    # Check for obvious fail case.
    if len(ransomNote) > len(magazine): return False

    # In Python, we can use the Counter class. It does all the work that the
    # makeCountsMap(...) function in our pseudocode did!
    magazine_counts = collections.Counter(magazine)
    ransom_note_counts = collections.Counter(ransomNote)

    # For each *unique* character in the ransom note:
    for char, count in ransom_note_counts.items():
        # Check that the count of char in the magazine is equal
        # or higher than the count in the ransom note.
        magazine_count = magazine_counts[char]
        if magazine_count < count:
            return False

    # If we got this far, we can successfully build the note.
    return True
```

Complexity Analysis

We'll say m is the length of the **m**agazine, and n is the length of the ransom **n**ote.

Also, let k be the number of unique characters across both the ransom note and magazine. While this is never more than 26, we'll treat it as a variable for a more accurate complexity analysis.

The basic `HashMap` operations, `get(...)` and `put(...)`, are $O(1)$ time complexity.

- Time Complexity : $O(m)$.

  When $m < n$, we immediately return `false`. Therefore, the worst case occurs when $m \geq n$.

  Creating a `HashMap` of counts for the magazine is $O(m)$, as each insertion/ count update is is $O(1)$, and is done for each of the m characters.

  Likewise, creating the `HashMap` of counts for the ransom note is $O(n)$.

  We then iterate over the ransom note `HashMap`, which contains at most n unique values, looking up their counterparts in the magazine `HashMap. This is, therefore, at *worst* $O(n)$.

  This gives us $O(n) + O(n) + O(m)$. Now, remember how we said $m \geq n$? This means that we can simplify it to $O(m) + O(m) + O(m) = 3 \cdot O(m) = O(m)$, dropping the constant of 3.

- Space Complexity : $O(k)$ / $O(1)$.

  We build two `HashMap` s of counts; each with up to k characters in them. This means that they take up $O(k)$ space.

  For this problem, because k is never more than 26, which is a constant, it'd be reasonable to say that this algorithm requires $O(1)$ space.

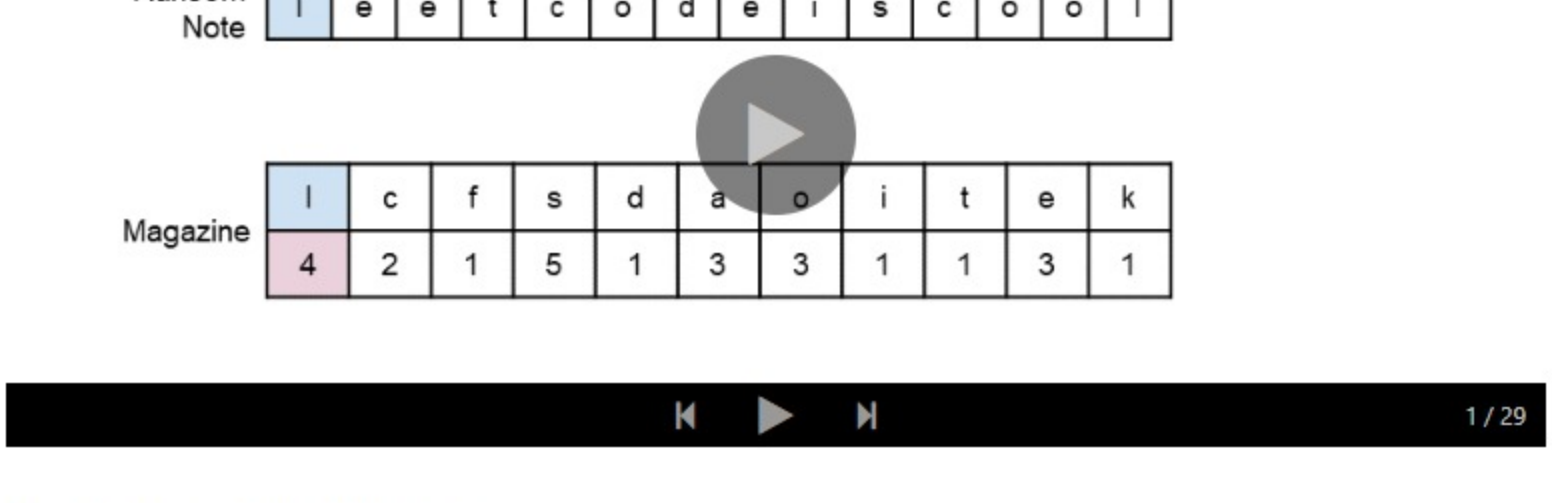
Approach 3: One HashMap

Intuition

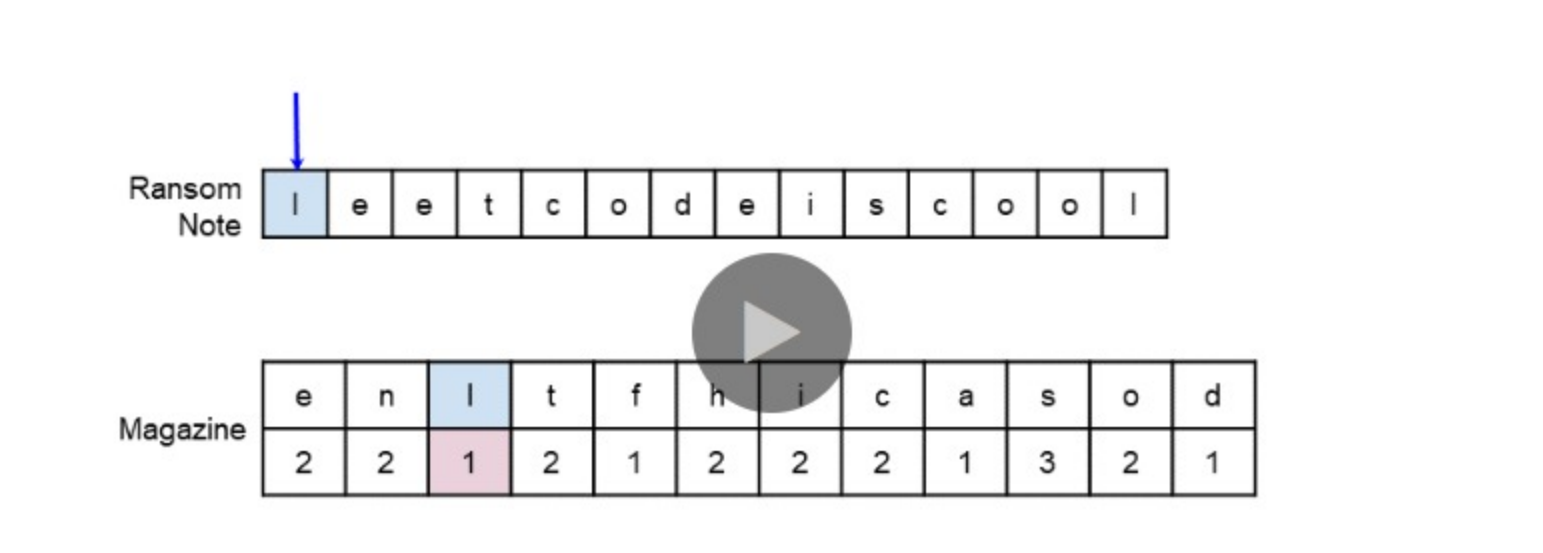
In the previous approach, we used two `HashMap` s. You might have noticed a slightly better way though; we can simply put the magazine into a `HashMap`, and then *subtract* characters from the ransom note from it. Here is the pseudocode, using our `makeCountsMap(...)` function from above.

```
magazineCounts = makeCountsMap(magazine)
for each char in ransomNote:
    countInMagazine = magazineCounts.get(char)
    if countInMagazine == 0:
        return False
    magazineCounts.put(char, countInMagazine - 1)
return True
```

Here is an animation of the algorithm on our "true" case from before.



And here's one on the "false" case.



Algorithm

Java Python Copy

```python
def canConstruct(self, ransomNote: str, magazine: str) -> bool:

    # Check for obvious fail case.
    if len(ransomNote) > len(magazine): return False

    # In Python, we can use the Counter class. It does all the work that the
    # makeCountsMap(...) function in our pseudocode did!
    letters = collections.Counter(magazine)

    # For each character, c, in the ransom note:
    for c in ransomNote:
        # If there are none of c left, return False.
        if letters[c] <= 0:
            return False
        # Remove one of c from the Counter.
        letters[c] -= 1
    # If we got this far, we can successfully build the note.
    return True
```

Complexity Analysis

We'll say m is the length of the **m**agazine, and n is the length of the ransom **n**ote.

Also, let k be the number of unique characters across both the ransom note and magazine. While this is never more than 26, we'll treat it as a variable for a more accurate complexity analysis.

The basic `HashMap` operations, `get(...)` and `put(...)`, are $O(1)$ time complexity.

- Time Complexity : $O(m)$.

  When $m < n$, we immediately return `false`. Therefore, the worst case occurs when $m \geq n$.

  Creating a `HashMap` of counts for the magazine is $O(m)$, as each insertion/ count update is is $O(1)$, and is done for each of the m characters.

  We then iterate over the ransom note, performing an $O(1)$ operation for each character in it. This has a cost of $O(n)$.

  Becuase we know that $m \geq n$, again this simplifies to $O(m)$.

- Space Complexity : $O(k)$ / $O(1)$.

  Same as above.

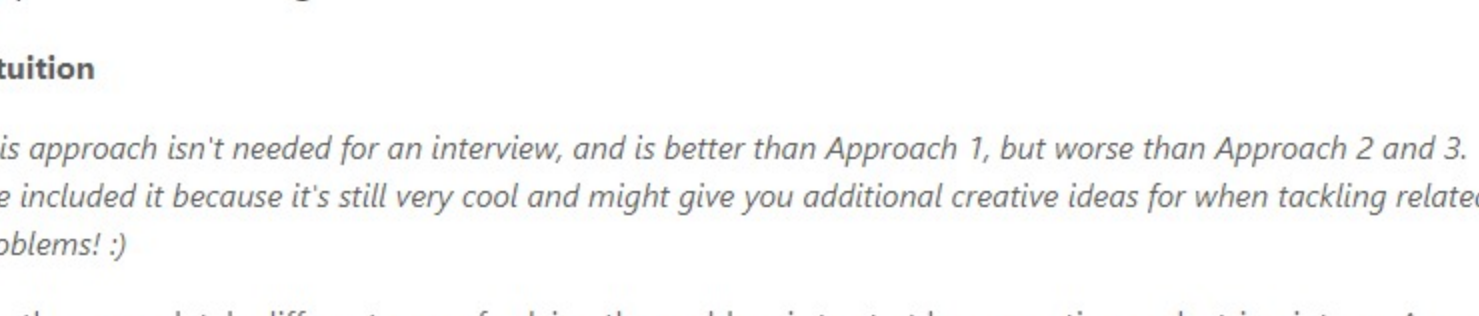
  For this problem, because k is never more than 26, which is a constant, it'd be reasonable to say that this algorithm requires $O(1)$ space.

Approach 4: Sorting and Stacks

Intuition

This approach isn't needed for an interview, and is better than Approach 1, but worse than Approach 2 and 3. I've included it because it's still very cool and might give you additional creative ideas for when tackling related problems! :)

Another, completely different, way of solving the problem is to start by converting each string into an Array of characters, and then *reverse* sorting them by alphabetical order. It's not actually necessary to reverse sort, but it will make things easier for the rest of the algorithm. For example, here's the sorted characters for the ransom note `leetcode is cool` and the magazine `close call as fools take sides`.



Now, convert each array into a stack.

Compare the tops of the stacks. There are three possibilities.

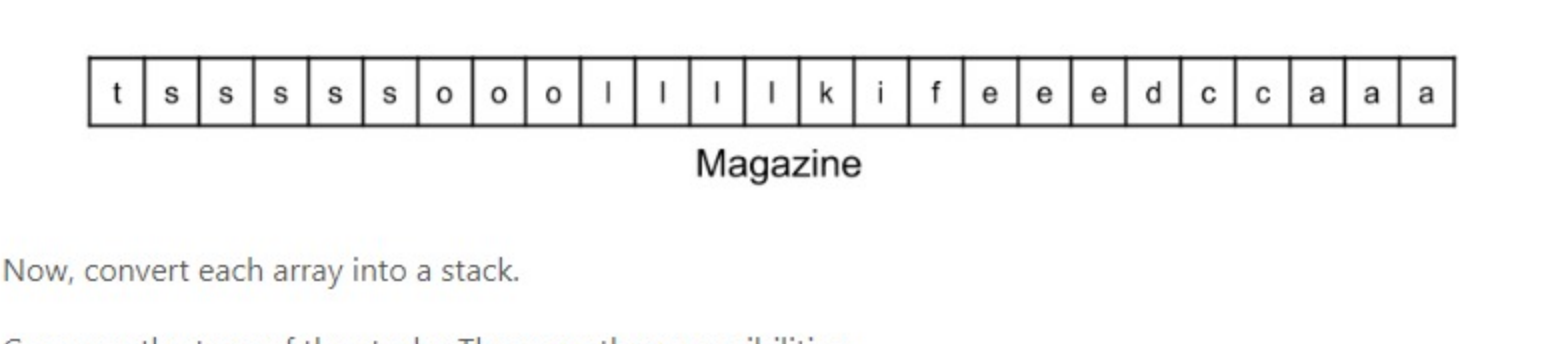
1. The characters are the same.
2. The ransom note character is *earlier* in the alphabet than the magazine character.
3. The ransom note character is *later* in the alphabet than the magazine character.

For the first possibility, we've found a copy of the letter we need in the magazine, for a letter in our ransom note. So pop the top off each stack.

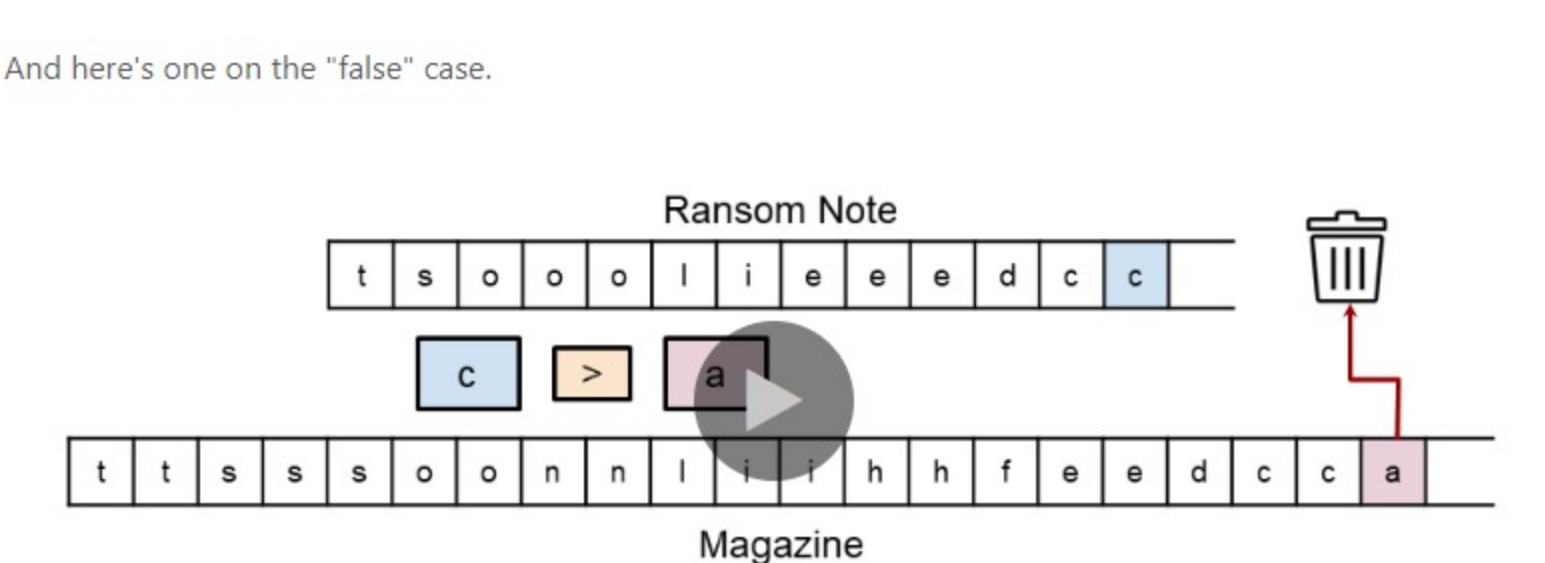
For the second possibility, we know that the letter we need *can't be on the magazine stack*. This is because all the other characters on the magazine must be even later than the top, and we needed an earlier letter. Therefore, we can return `false` now.

For the third possibility, we know that the letter on the top of the magazine stack will never be needed, as all the characters on the ransom note stack must be later than it, so we pop the top off *just* the magazine stack.

Here is an animation of the algorithm on our "true" case from before.



And here's one on the "false" case.



Algorithm

Java Python Copy

```python
def canConstruct(self, ransomNote: str, magazine: str) -> bool:

    # Check for obvious fail case.
    if len(ransomNote) > len(magazine): return False

    # Reverse sort the note and magazine. In Python, we simply
    # treat a list as a stack.
    ransomNote = sorted(ransomNote, reverse=True)
    magazine = sorted(magazine, reverse=True)

    # While there are letters left on both stacks:
    while ransomNote and magazine:
        # If the tops are the same, pop both because we have found a match.
        if ransomNote[-1] == magazine[-1]:
            ransomNote.pop()
            magazine.pop()
        # If magazine's top is earlier in the alphabet, we should remove that
        # character of magazine as we definitely won't need that letter.
        elif magazine[-1] < ransomNote[-1]:
            magazine.pop()
        # Otherwise, it's impossible for top of ransomNote to be in magazine.
        else:
            return False
    # Return true iff the entire ransomNote was built.
    return not ransomNote
```

Complexity Analysis

We'll say m is the length of the **m**agazine, and n is the length of the ransom **n**ote.

- Time Complexity : $O(m \log m)$.

  When $m < n$, we immediately return `false`. Therefore, the worst case occurs when $m \geq n$.

  Sorting the magazine is $O(m \log m)$. Inserting the contents into the stack is $O(m)$, which is insignificant. This, therefore, gives us $O(m \log m)$ for creating the magazine stack.

  Likewise, creating the ransom note stack is $O(n \log n)$.

  In total, the stacks contain $n + m$ characters. For each iteration of the loop, we are either immediately returning `false`, or removing at least one character from the stacks. This means that the stack processing loop has to use at most $O(n + m)$ time.

  This gives us $O(m \log m) + O(n \log n) + O(n + m)$. Now, remembering that $m \geq n$ it simplifies down to $O(m \log m) + O(m \log m) + O(m + m) = 2 \cdot O(m \log m) + O(2 \cdot m) = O(m \log m)$.

- Space Complexity : $O(m)$.

  The magazine stack requires $O(m)$ space, and the ransom note stack requires $O(n)$ space. Because $m \geq n$, this simplifies down to $O(m)$.

Rate this article: ★★★★★

Previous Next

Comments: 10 Sort By

Type comment here... (Markdown is supported)

Preview Post

hinslu0802 ★31 May 3, 2020 9:29 PM

I don't know why there are always so many eazy problems have a lot of over-detailed solutions and many people try to solve them in many different unnecessary ways, on the other hand, ignoring how to solve out many valuable medium or hard problems in a better way.

21 Share Reply

SHOW 2 REPLIES

LeoShi817 ★44 April 16, 2020 9:08 AM

Since inputs are lowercase, we could use int[26] as map too.

```
public boolean canConstruct(String ransomNote, String magazine) {
    int[] map = new int[26];
```

Read More

7 Share Reply

Dhruwat ★5 April 9, 2020 1:20 AM

For Approach, do you necessarily need a stack. I think you just need two pointers after you have sorted the input arrays.

4 Share Reply

SHOW 1 REPLY

prithivm ★4 March 16, 2020 10:48 PM

```
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        counter = Counter(magazine)
```

Read More

1 Share Reply

cputask ★0 June 27, 2020 6:58 PM

(1) Keep two fixed length arrays, each of size 26 to keep character occurence count in magazine and ransomNote
(2) Compare arrays

Probably, O(1) space and O(n) time solution?

Read More

0 Share Reply

praveenpurwar2004 ★9 June 7, 2020 9:37 AM

simple Python solution to this problem though not great with complexity:

```
m=sorted(magazine)
        for i in ransomNote:
```

Read More

0 Share Reply

c0re5 ★8 May 3, 2020 6:40 PM

@Hai_dee - Very nice solutions and write-up. I would probably use `Deque`, if I want to push on one side and pull out from the other side, after sorting the characters.

```
private Deque<Character> sortedCharacterStack(String s) {
```

Read More

0 Share Reply

SHOW 1 REPLY

bloomer ★160 May 3, 2020 2:07 PM

@Hai_dee Can you suggest some problems that can use approach 4 to solve questions.

0 Share Reply

ashta7 ★0 April 11, 2020 12:44 PM

In Approach 4,
Why do we do this:

```
// Return true iff the entire ransomNote was built.
```

Read More

0 Share Reply

SHOW 1 REPLY

nitinpandey ★52 March 18, 2020 11:33 AM

Approach 4, should point 2 and point 3 be interchanged?

0 Share Reply