Previous Next

459. Repeated Substring Pattern

May 16, 2020 | 3.3K views

Average Rating: 3.82 (11 votes)

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

Example 1:

```
Input: "abab"
Output: True
Explanation: It's the substring "ab" twice.
```

Example 2:

```
Input: "aba"
Output: False
```

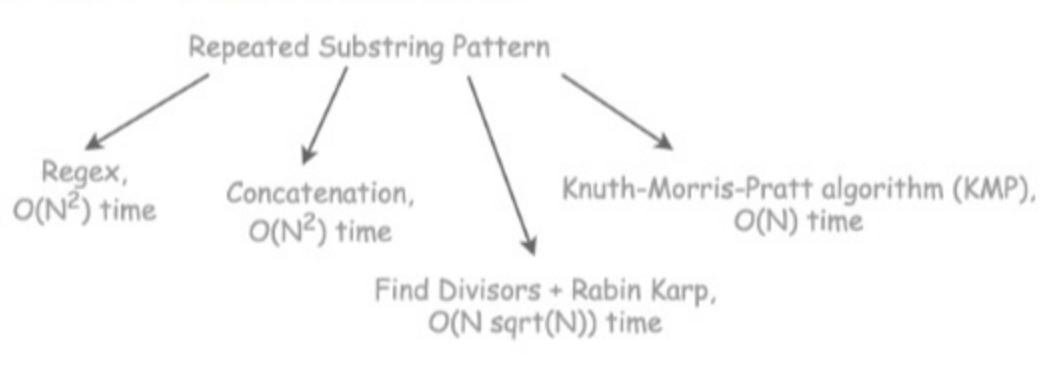
Example 3:

```
Input: "abcabcabcabc"
Output: True
Explanation: It's the substring "abc" four times. (And the substring "abcabc" twice.)
```

Solution

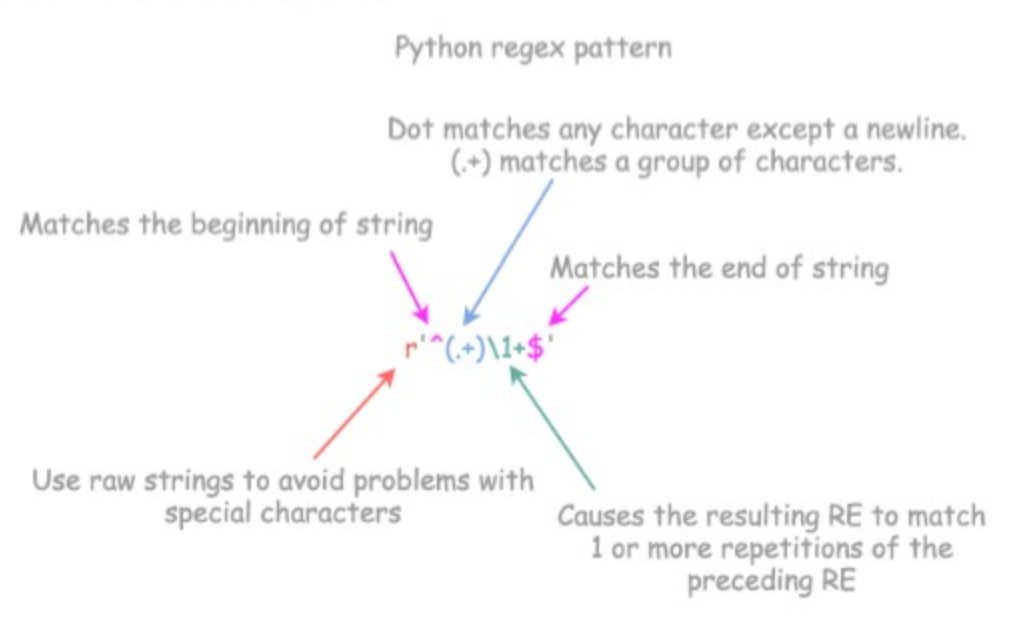
Overview

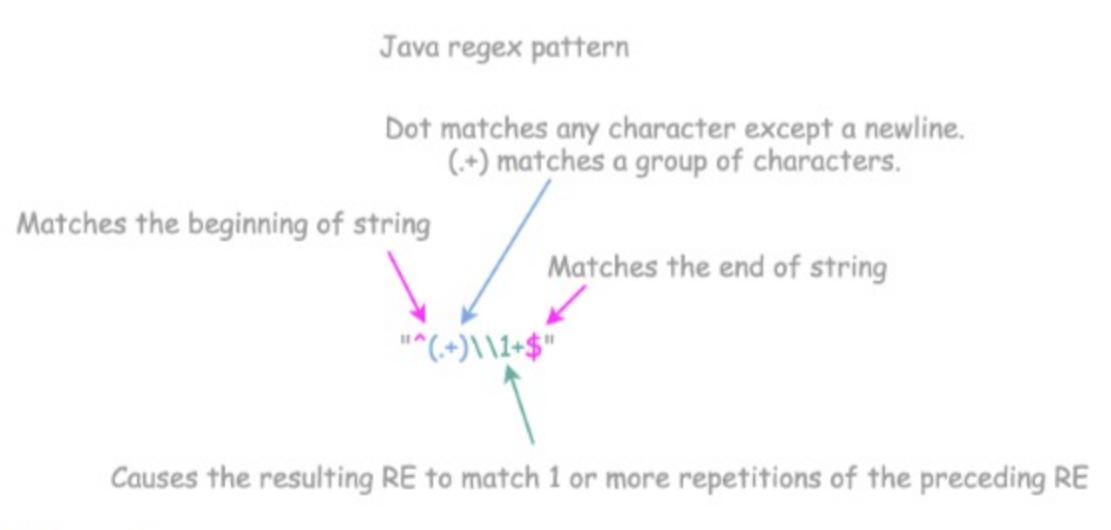
The problem could be solved in many ways. Easy approaches have $\mathcal{O}(N^2)$ time complexity, though one could improve it by using one of string searching algorithms.



Approach 1: Regex

To use regex during the interviews is like to use built-in functions, the community has no single opinion about it yet, and it's a sort of risk.





Implementation

C++ Java Python3 Copy

```python
import re
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        pattern = re.compile(r'^(.+)\1+$')
        return pattern.match(s)
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N^2)$ because we use greedy regex pattern. Once we have a `+`, the pattern is greedy.

  The difference between the greedy and the non-greedy match is the following:

  - the non-greedy match will try to match as few repetitions of the quantified pattern as possible.
  - the greedy match will try to match as many repetitions as possible.

  The worst-case situation here is to check all possible pattern lengths from `N` to `1` that would result in $\mathcal{O}(N^2)$ time complexity.

- Space complexity: $\mathcal{O}(1)$. We don't use any additional data structures, and everything depends on internal regex implementation, which is evolving quite fast nowadays. If you're interested to dig depeer, here is a famous article by Russ Cox which inspired a lot of discussions and code changes in Python community.

Approach 2: Concatenation

Repeated pattern string looks like `PatternPattern`, and the others like `Pattern1Pattern2`.

Let's double the input string:

`PatternPattern` --> `PatternPatternPatternPattern`

`Pattern1Pattern2` --> `Pattern1Pattern2Pattern1Pattern2`

Now let's cut the first and the last characters in the doubled string:

`PatternPattern` --> `*atternPatternPatternPatter*`

`Pattern1Pattern2` --> `*attern1Pattern2Pattern1Pattern*`

It's quite evident that if the new string contains the input string, the input string is a repeated pattern string.

Implementation

C++ Java Python3 Copy

```python
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        return s in (s + s)[1: -1]
```

Complexity Analysis

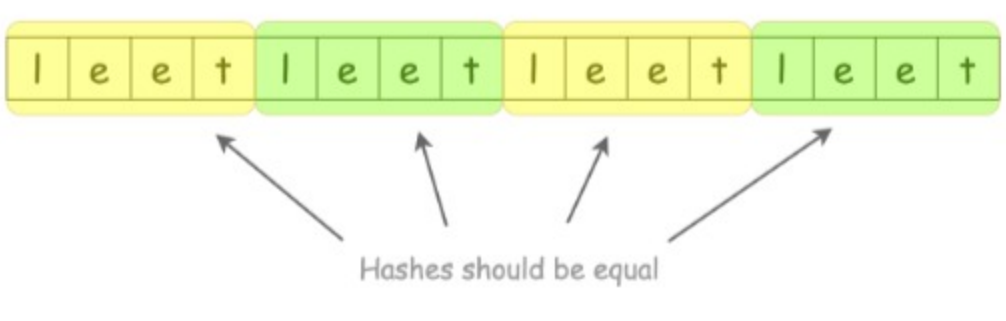
- Time complexity: $\mathcal{O}(N^2)$ because of the way `in` and `contains` are implemented.
- Space complexity: $\mathcal{O}(N)$, the space is implicitly used to keep `s + s` string.

Approach 3: Find Divisors + Rabin-Karp

Rabin-Karp

Rabin-Karp is a linear-time $\mathcal{O}(N)$ string searching algorithm:

- Move a sliding window of length L along the string of length N.
- Check hash of the string in the sliding window.



In some situations, one has to implement a particular hash algorithm to fit in a linear time, for example, we used *rolling hash algorithm* for the problem Longest Duplicate Substring.

For the current problem the standard `hash` / `hashCode` is enough because the idea is to check only lengths `L`, which are divisors of `N`. This way we're not *sliding*, we're *jumping*:

- the first string is `0..L`
- the second string is `L..2L`
- ...
- the last string is `N - L..N`

To copy characters in sliding window takes time `L`, to compute hash - time `L` as well. In total, there are `N / L` substrings, that makes it all work in a linear time $\mathcal{O}(N)$.

Find divisors

Now the only problem is to find divisors of `N`. Let's iterate to the square root of `N`, and for each identified divisor `i` calculate the paired divisor `N / i`.

Algorithm

- Deal with base cases: `n <= 2`.
- Iterate from $\sqrt{n}$ to 1.
  - For each divisor `n % i == 0`:
    - Compute paired divisor `n / i`.
    - Use Rabin-Karp to check substrings of the lengths `l = i` and `l = n / i`:
      - Take as a reference hash `first_hash` the hash of the first substring of length `l`.
      - Jump along the string with a step of length `l` while the hash of the current substring is equal to `first_hash`.
      - If the hashes of all substrings along the way are equal, the input string consists of repeated patterns of length `l`. Return True.

> Side note. The good practice is to verify the equality of two substrings after the hash match. This logic is not hard to add, and it could bring you kudos during the interview.

Implementation

C++ Java Python3 Copy

```python
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        n = len(s)
        if n < 2:
            return False
        if n == 2:
            return s[0] == s[1]

        for i in range(int(n**0.5), 0, -1):
            if n % i == 0:
                divisors = [i]
                if i != 1:
                    divisors.append(n // i)
                for l in divisors:
                    first_hash = curr_hash = hash(s[:l])
                    start = l
                    while start != n and curr_hash == first_hash:
                        curr_hash = hash(s[start:start + l])
                        start += l
                    if start == n and curr_hash == first_hash:
                        return True

        return False
```

Complexity Analysis

- Time complexity: up to $\mathcal{O}(N\sqrt{N})$. $\mathcal{O}(\sqrt{N})$ to compute all divisors and $\mathcal{O}(N)$ for each divisor "verification". That's an upper-bound estimation because divisor function grows slower than $\sqrt{N}$.
- Space complexity: up to $\mathcal{O}(\sqrt{N})$ to keep a copy of each substring during the hash computation.

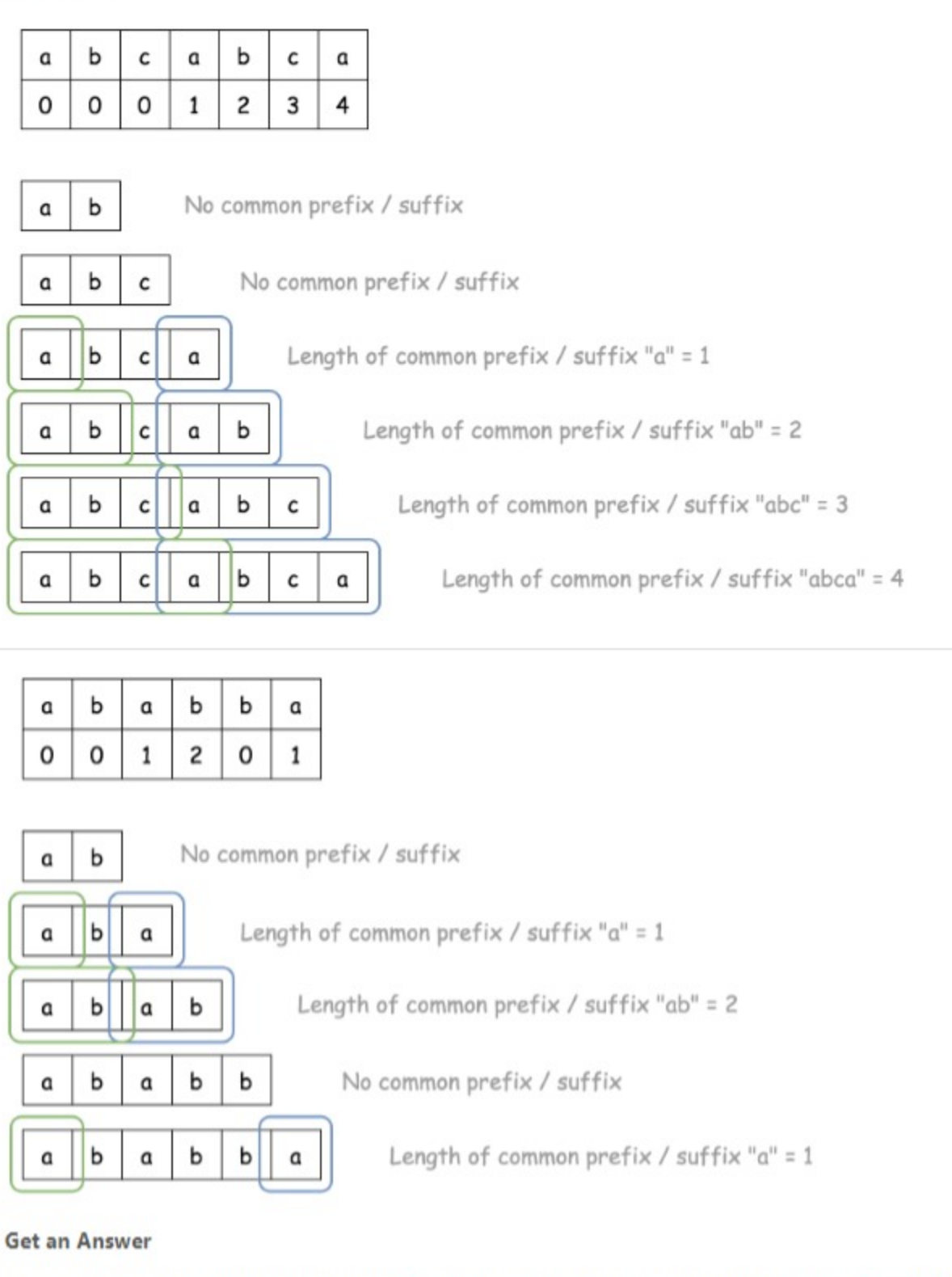
Approach 4: Knuth-Morris-Pratt Algorithm (KMP)

Lookup Table

Rabin-Karp is the best fit for the *multiple* pattern search, whereas KMP is typically used for the *single* pattern search.

The key to KMP is the *partial match table*, often called *lookup table*, or *failure function table*. **It stores the length of the longest prefix that is also a suffix.**

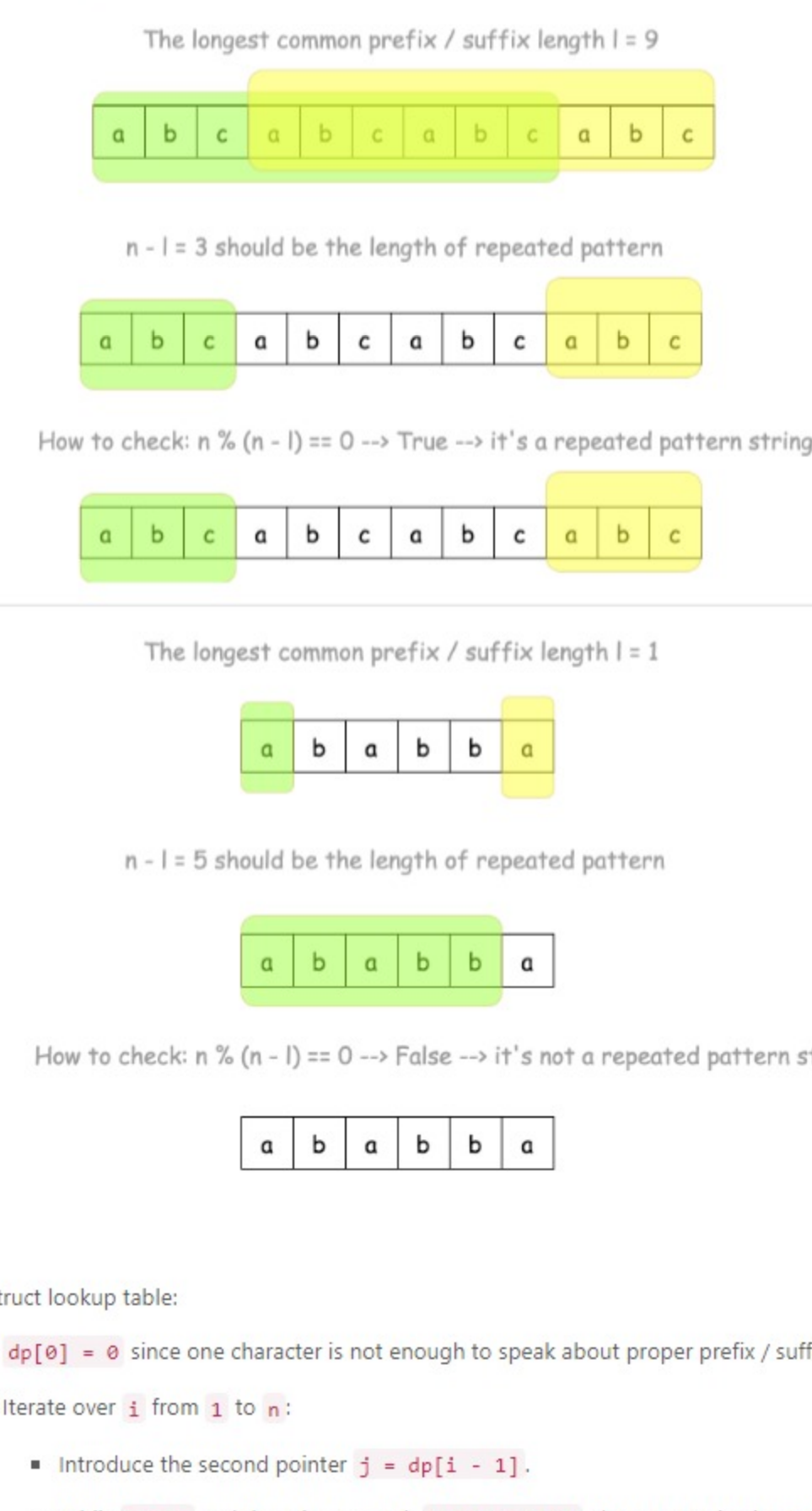
Here are two examples



How to Get an Answer

Once we have a lookup table, we know the length `l` of common prefix/suffix for the input string: `l = dp[n - 1]`.

That means that `n - l` should the length of the repeated sequence. To confirm that, one should verify if `n - l` is a divisor of `n`.



Algorithm

- Construct lookup table:
  - `dp[0] = 0` since one character is not enough to speak about proper prefix / suffix.
  - Iterate over `i` from `1` to `n`:
    - Introduce the second pointer `j = dp[i - 1]`.
    - While `j > 0` and there is no match `s[i] != s[j]`, do one step back to consider a shorter prefix: `j = dp[j - 1]`.
    - If we found a match `s[i] == s[j]`, move forward: `j += 1`
    - Write down the length of common prefix / suffix: `dp[i] = j`.
- Now we have a length of common prefix / suffix for the entire string: `l = dp[n - 1]`.
- The string is a repeated pattern string if this length is nonzero and `n - l` is a divisor of `n`. Return `l != 0 and n % (n - l) == 0`.

Implementation

C++ Java Python3 Copy

```python
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        n = len(s)
        dp = [0] * n
        # Construct partial match table (lookup table).
        # It stores the length of the proper prefix that is also a proper suffix.
        # ex. ababa --> [0, 0, 1, 2, 1]
        # ab --> the length of common prefix / suffix = 0
        # aba --> the length of common prefix / suffix = 1
        # abab --> the length of common prefix / suffix = 2
        # ababa --> the length of common prefix / suffix = 1
        for i in range(1, n):
            j = dp[i - 1]
            while j > 0 and s[i] != s[j]:
                j = dp[j - 1]
            if s[i] == s[j]:
                j += 1
            dp[i] = j

        l = dp[n - 1]
        # check if it's repeated pattern string
        return l != 0 and n % (n - l) == 0
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$. During the execution, `j` could be decreased at most N times and then increased at most N times, that makes overall execution time to be linear $\mathcal{O}(N)$.
- Space complexity: $\mathcal{O}(N)$ to keep the lookup table.

Rate this article:

Previous Next

Comments: 6 Sort By

Type comment here... (Markdown is supported)

Preview Post

leoneed ★ 17 May 17, 2020 7:03 PM

Is it still easy if we want to get O(N)?

7 Share Reply

SHOW 3 REPLIES

kamkam34 ★ 12 May 17, 2020 7:46 AM

Can you please explain why if n%(n-l) == 0 is met along with l!=0 (I understand this one) then the result is true? because (n-l) only tells us the length of the pattern, how does n%(n-l)==0 tells us the pattern is repeated? why isn't it required to check in another loop to see if the pattern is really repeated or not? Thanks

2 Share Reply

SHOW 2 REPLIES

LeetCoding_Master ★ 189 a day ago

It is not easy level for sure.

1 Share Reply

neo_coder ★ 8 June 27, 2020 11:03 AM

Can someone please help clarify:

Approach 1- (Regex) "The worst-case situation here is to check all possible pattern lengths from N to 1 that would result in N^2" <-- What's the other N, i.e. why N^2?

Read More

1 Share Reply

dilit ★ 37 July 13, 2020 2:58 AM

KMP is just brilliant!!!
The first part with j following i in the previous pattern is smart!
The backtracking part shows a true master at work. j is set to dp[j-1],
which is the pre-previous pattern's element to which s[j] had been compared.
This way, j floats to zero in the while loop!

Read More

0 Share Reply

yuewu767 ★ 0 May 28, 2020 11:37 PM

How to make sure if s is concatenations of substrings, then n-l must be the length of smallest such repeated pattern, i.e. I can't stretch further into middle of the first pattern? (I can understand if l != 0 && n % (n - l) == 0, then by transitivity it must be concatenation of substrings, but confused about the reversed direction). Thanks!

0 Share Reply