Previous Next

28. Implement strstr()

Dec. 9, 2019 | 40.3K views

Average Rating: 3.67 (30 votes)

Implement strStr().

Return the index of the first occurrence of needle in haystack, or **-1** if needle is not part of haystack.

Example 1:

```
Input: haystack = "hello", needle = "ll"
Output: 2
```

Example 2:

```
Input: haystack = "aaaaa", needle = "bba"
Output: -1
```

Clarification:

What should we return when `needle` is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when `needle` is an empty string. This is consistent to C's strstr() and Java's indexOf().

Solution

Overview

The problem is to find needle of length L in the haystack of length N.

Let's discuss three different ideas how to proceed. They are all based on sliding window idea. The key point is how to implement a window slice.

Linear time window slice $\mathcal{O}(L)$ is quite easy, move the window of length L along the haystack and compare substring in the window with the needle. Overall that would result in $\mathcal{O}((N - L)L)$ time complexity.

Could that be improved? Yes. Two pointers approach is still the case of linear time slice, though the comparison happens not for all substrings, and that improves the best time complexity up to $\mathcal{O}(N)$. The worst time complexity is still $\mathcal{O}((N - L)L)$ though.

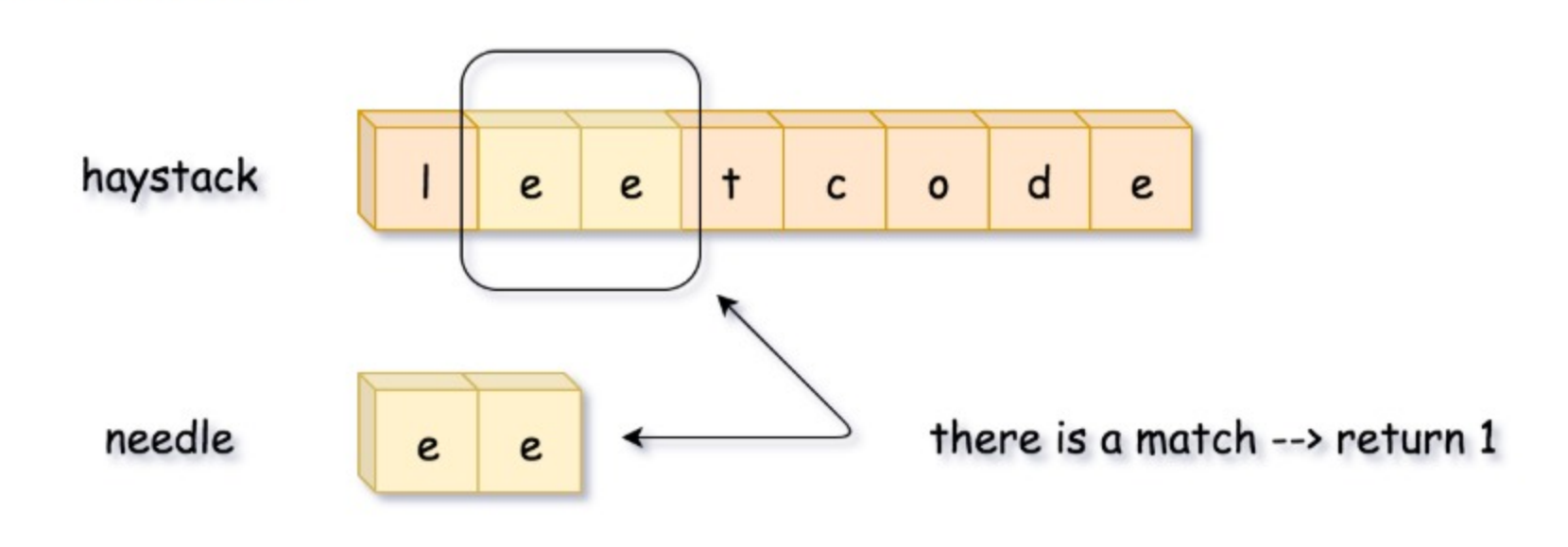
Could that be improved to $\mathcal{O}(N)$? Yes, but one has to implement constant time slice $\mathcal{O}(1)$. There are two ways to do it:

- Rabin-Karp = constant-time slice using rolling hash algorithm.
- Bit manipulation = constant-time slice using bitmasks.

Bit Manipulation approach in Java is more suitable for the short strings or strings with very limited number of characters, ex. Repeated DNA Sequences. That's a consequence of overflow issues in Java (in Python there is no such a problem). Here we deal with quite long strings and it's more simple to implement the basic version of Rabin Karp algorithm.

Approach 1: Substring: Linear Time Slice

Quite straightforward approach - move sliding window along the string and compare substring in the window with the needle.



Implementation

Java Python Copy

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        L, n = len(needle), len(haystack)

        for start in range(n - L + 1):
            if haystack[start: start + L] == needle:
                return start
        return -1
```

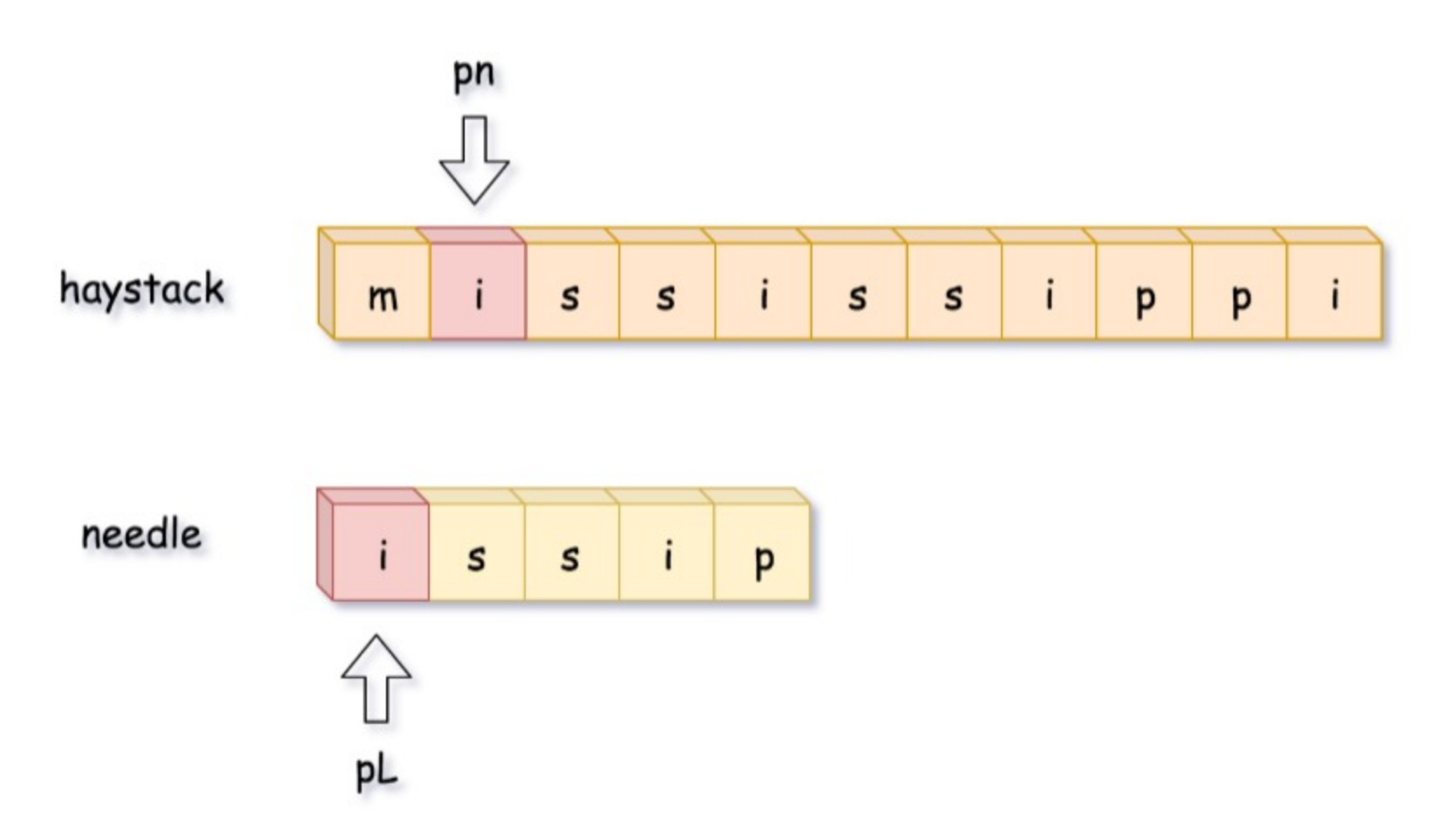
Complexity Analysis

- Time complexity: $\mathcal{O}((N - L)L)$, where N is a length of haystack and L is a length of needle. We compute a substring of length L in a loop, which is executed (N - L) times.
- Space complexity: $\mathcal{O}(1)$.

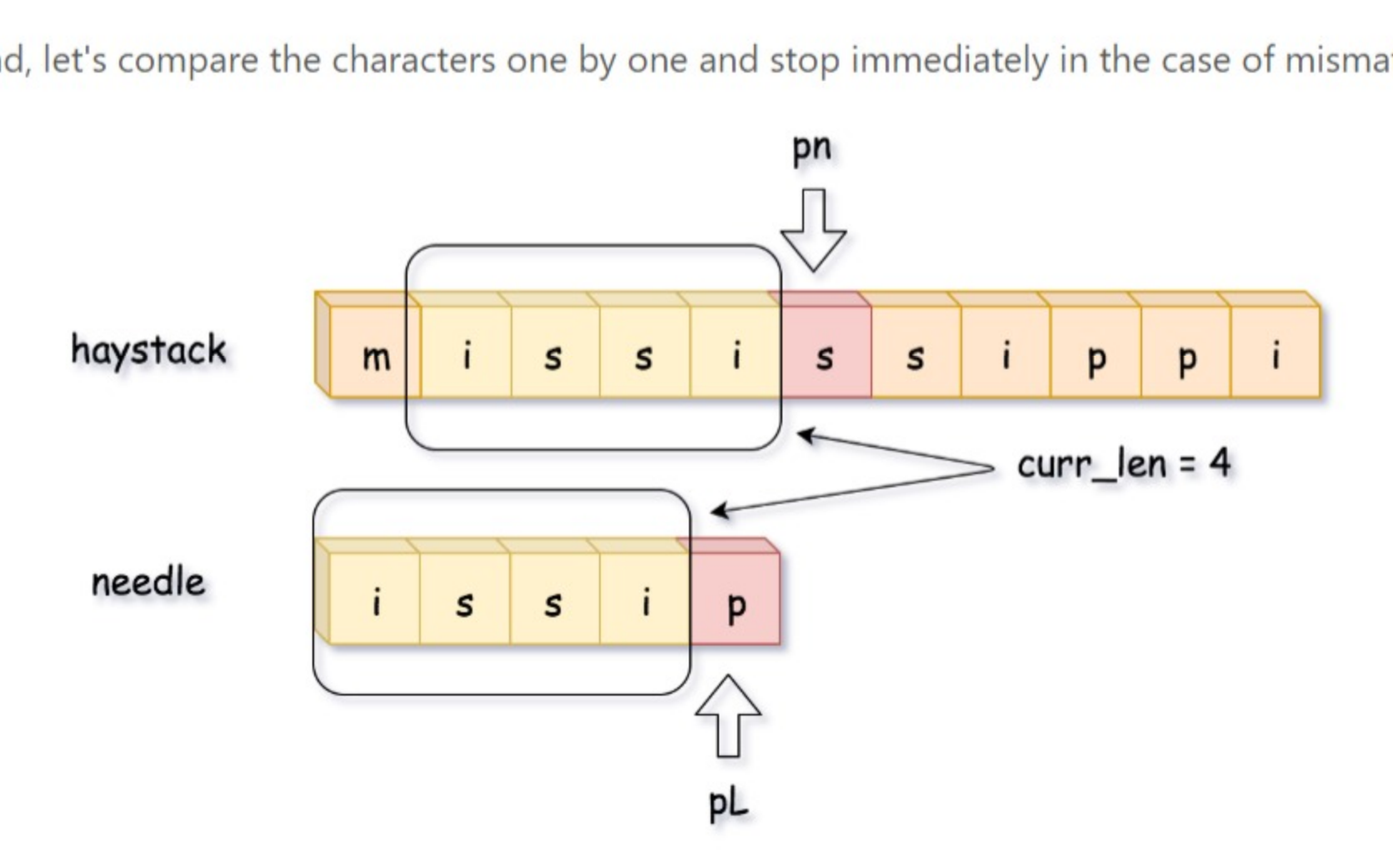
Approach 2: Two Pointers: Linear Time Slice

Drawback of the previous algorithm is that one compares absolutely all substrings of length L with the needle. There is no need to that.

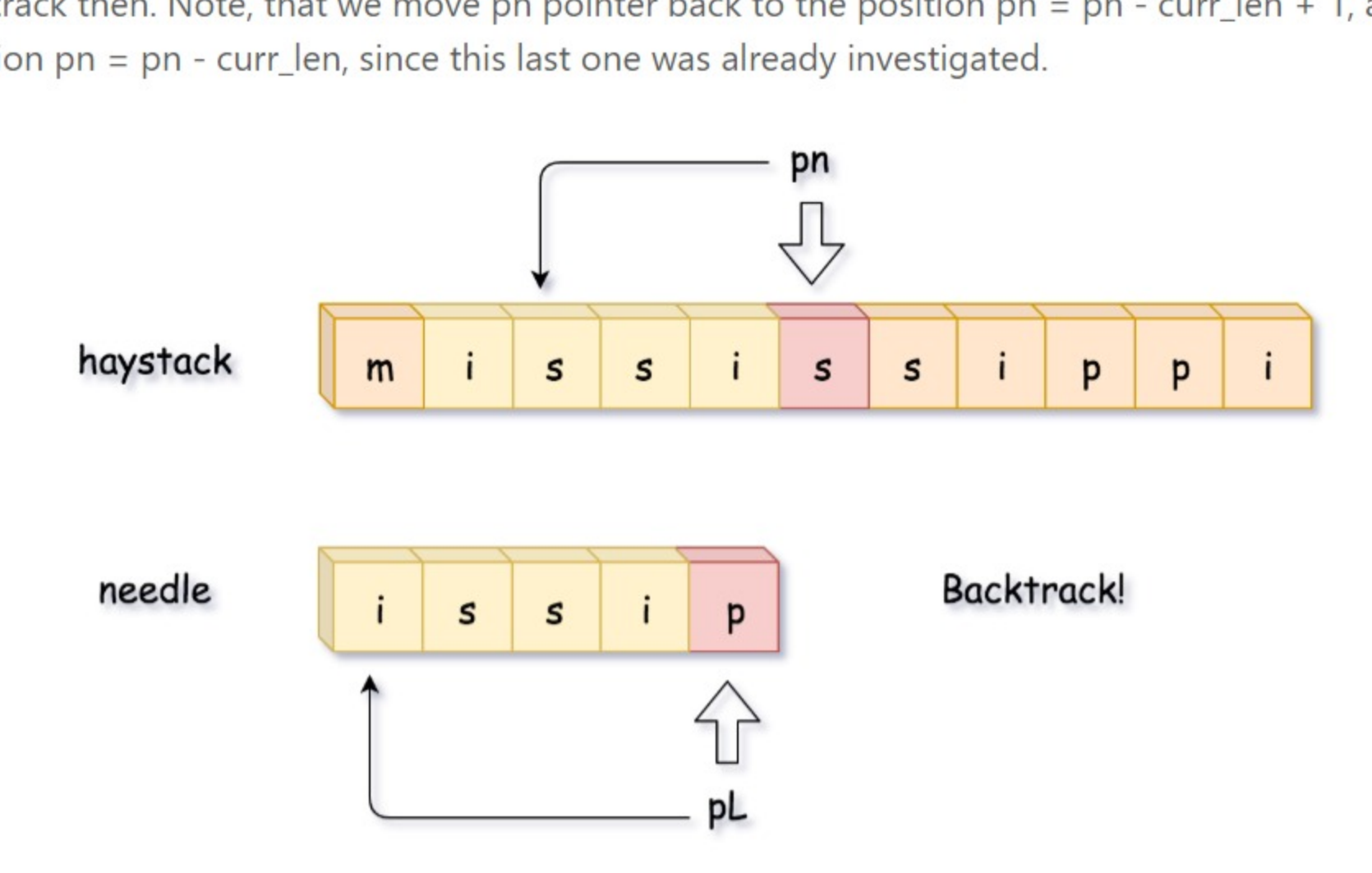
First, let's compare only substrings which starts from the first character in the needle substring.



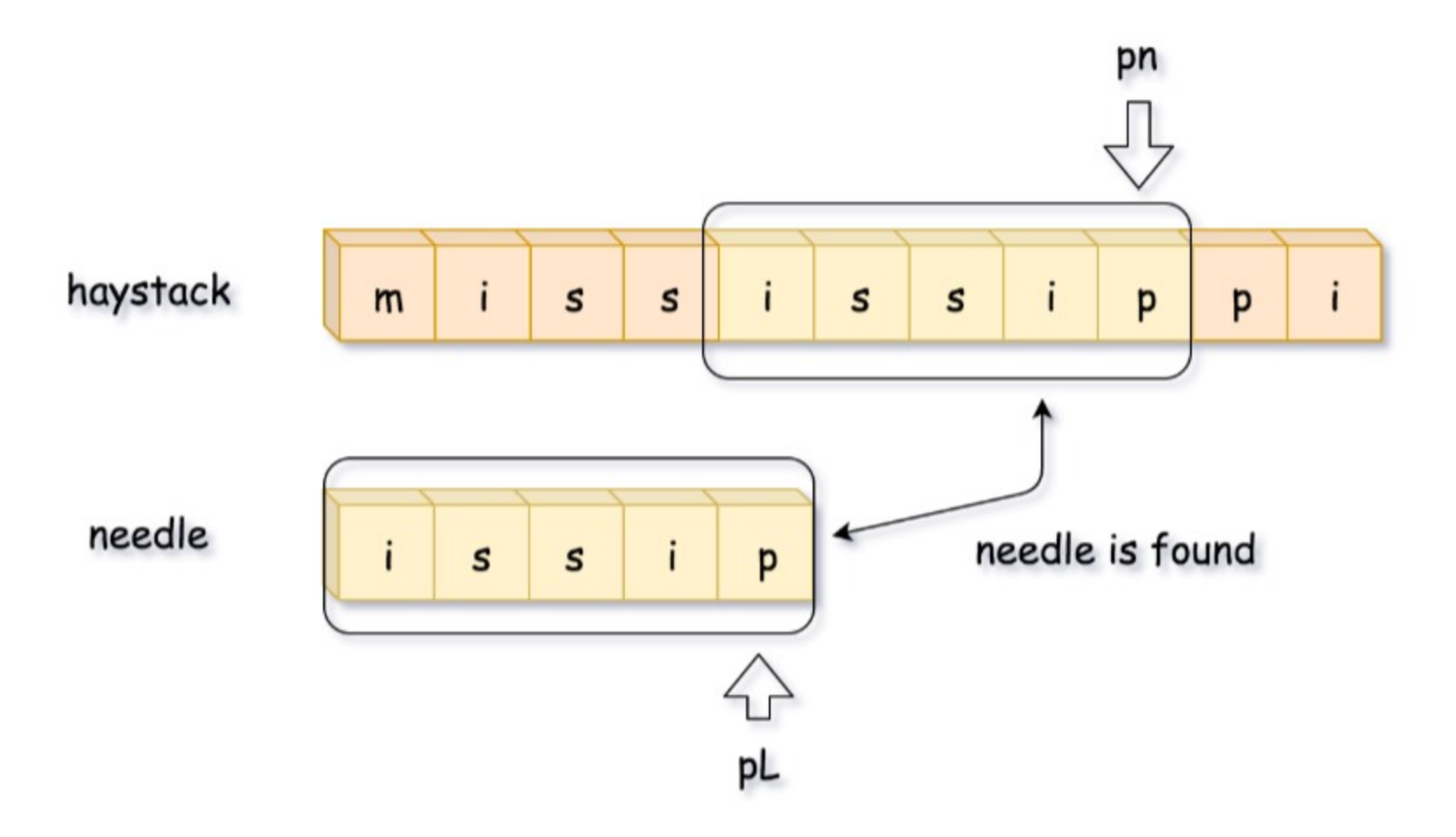
Second, let's compare the characters one by one and stop immediately in the case of mismatch.



Here it was impossible to manage the full match up to the length of needle string, which is L = 5. Let's backtrack then. Note, that we move pn pointer back to the position pn = pn - curr_len + 1, and *not* to the position pn = pn - curr_len, since this last one was already investigated.



Let's try again. Here we've managed to get the full match during the second attempt, so let's return the start position of that match, pn - L.



Algorithm

- Move pn till you'll find the first character of the needle string in the haystack.
- Compute the max string match by increasing pn, pL and curr_len in the case of equal characters.
- If you managed to get the full match, curr_len == L, return the start position of that match, pn - L.
- If you didn't, backtrack: pn = pn - curr_len + 1, pL = 0, curr_len = 0.

Implementation

Java Python Copy

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        L, n = len(needle), len(haystack)
        if L == 0:
            return 0

        pn = 0
        while pn < n - L + 1:
            # find the position of the first needle character
            # in the haystack string
            while pn < n - L + 1 and haystack[pn] != needle[0]:
                pn += 1

            # compute the max match string
            curr_len = pL = 0
            while pL < L and pn < n and haystack[pn] == needle[pL]:
                pn += 1
                pL += 1
                curr_len += 1

            # if the whole needle string is found,
            # return its start position
            if curr_len == L:
                return pn - L

            # otherwise, backtrack
            pn = pn - curr_len + 1
```

Complexity Analysis

- Time complexity: $\mathcal{O}((N - L)L)$ in the worst case of numerous almost complete false matches, and $\mathcal{O}(N)$ in the best case of one single match.
- Space complexity: $\mathcal{O}(1)$.

Approach 3: Rabin Karp: Constant Time Slice

Let's now design the algorithm with $\mathcal{O}(N)$ time complexity even in the worst case. The idea is simple: move along the string, generate hash of substring in the sliding window and compare it with the reference hash of the needle string.

There are two technical problems:

1. How to implement a string slice in a constant time?
2. How to generate substring hash in a constant time?

String slice in a constant time

Strings are immutable in Java and Python, and to move sliding window in a constant time one has to convert string to another data structure, for example, to integer array of ascii-values.

Rolling hash: hash generation in a constant time

To generate hash of array of length L, one needs $\mathcal{O}(L)$ time.

> How to have constant time of hash generation? Use the advantage of slice: only one integer in, and only one - out.

That's the idea of rolling hash. Here we'll implement the simplest one, polynomial rolling hash. Beware that's polynomial rolling hash is NOT the Rabin fingerprint.

Since one deals here with lowercase English letters, all values in the integer array are between 0 and 25 : `arr[i] = (int)S.charAt(i) - (int)'a'`.
So one could consider string `abcd` -> `[0, 1, 2, 3]` as a number in a numeral system with the base 26. Hence `abcd` -> `[0, 1, 2, 3]` could be hashed as

$$h_0 = 0 \times 26^3 + 1 \times 26^2 + 2 \times 26^1 + 3 \times 26^0$$

Let's write the same formula in a generalised way, where c_i is an integer array element and $a = 26$ is a system base.

$$h_0 = c_0 a^{L-1} + c_1 a^{L-2} + ... + c_i a^{L-1-i} + ... + c_{L-1} a^1 + c_L a^0$$

$$h_0 = \sum_{i=0}^{L-1} c_i a^{L-1-i}$$

Now let's consider the slice `abcd` -> `bcde`. For int arrays that means `[0, 1, 2, 3]` -> `[1, 2, 3, 4]`, to remove number 0 and to add number 4.

$$h_1 = (h_0 - 0 \times 26^3) \times 26 + 4 \times 26^0$$

In a generalised way

$$h_1 = (h_0 a - c_0 a^L) + c_{L+1}$$

Now hash regeneration is perfect and fits in a constant time. There is one more issue to address: possible overflow problem.

How to avoid overflow

a^L could be a large number and hence the idea is to set limits to avoid the overflow. To set limits means to limit a hash by a given number called modulus and use everywhere not hash itself but `h % modulus`.

It's quite obvious that modulus should be large enough, but how large? Here one could read more about the topic, for the problem here 2^{31} is enough.

Algorithm

- Compute the hash of substring `haystack.substring(0, L)` and reference hash of `needle.substring(0, L)`.
- Iterate over the start position of possible match: from 1 to N - L.
  - Compute rolling hash based on the previous hash value.
  - Return start position if the hash is equal to the reference one.
- Return -1, that means that needle is not found.

Implementation

Java Python Copy

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        L, n = len(needle), len(haystack)
        if L > n:
            return -1

        # base value for the rolling hash function
        a = 26
        # modulus value for the rolling hash function to avoid overflow
        modulus = 2**31

        # lambda-function to convert character to integer
        h_to_int = lambda i : ord(haystack[i]) - ord('a')
        needle_to_int = lambda i : ord(needle[i]) - ord('a')

        # compute the hash of strings haystack[:L], needle[:L]
        h = ref_h = 0
        for i in range(L):
            h = (h * a + h_to_int(i)) % modulus
            ref_h = (ref_h * a + needle_to_int(i)) % modulus
        if h == ref_h:
            return 0

        # const value to be used often : a**L % modulus
        aL = pow(a, L, modulus)
        for start in range(1, n - L + 1):
            # compute rolling hash in O(1) time
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$, one computes the reference hash of the needle string in $\mathcal{O}(L)$ time, and then runs a loop of $(N - L)$ steps with constant time operations in it.
- Space complexity: $\mathcal{O}(1)$.

Rate this article:

Previous Next