Design and Analysis of Algorithms

- ***Course Code***: BCSE204L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 3

Module:3	String Matching Algorithms	5 hours

Naïve String-matching Algorithms,
KMP algorithm,
Rabin-Karp Algorithm,
Suffix Trees.

String Matching Algorithms: Brute Force approach

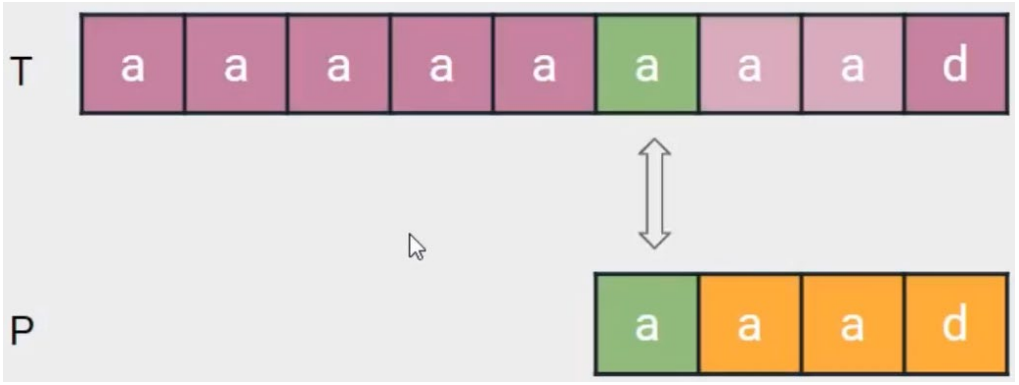
Text: abcdeabfgh
Pattern: eab

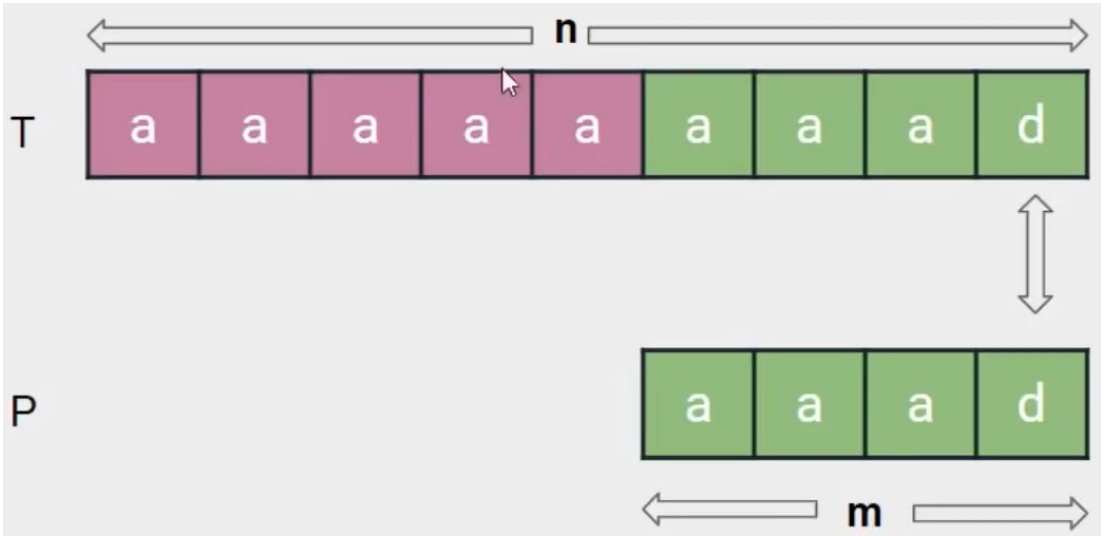
```
Step 1 :    a b c d e a b f g h
            e a b                 ...pattern is not matched.
Step 2 :    a b c d e a b f g h
              e a b               ...pattern is not matched.
Step 3 :    a b c d e a b f g h
                e a b             ...pattern is not matched.
Step 4 :    a b c d e a b f g h
                  e a b           ...pattern is not matched.
Step 4 :    a b c d e a b f g h
                    e a b         ...pattern is matched.
```

String Matching Algorithms: Brute Force approach

String Matching Algorithms: Brute Force approach

Worse Case

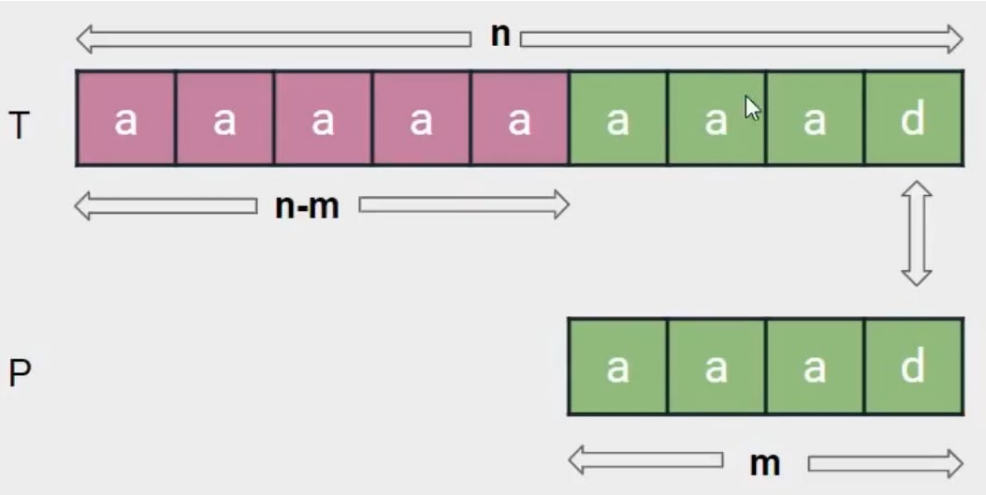




Brute force doesn't perform well when there are many repeated characters in both the text and the pattern. Let's consider a scenario where the text has a length of n and the pattern has a length of m.

String Matching Algorithms: Brute Force approach

Worse Case

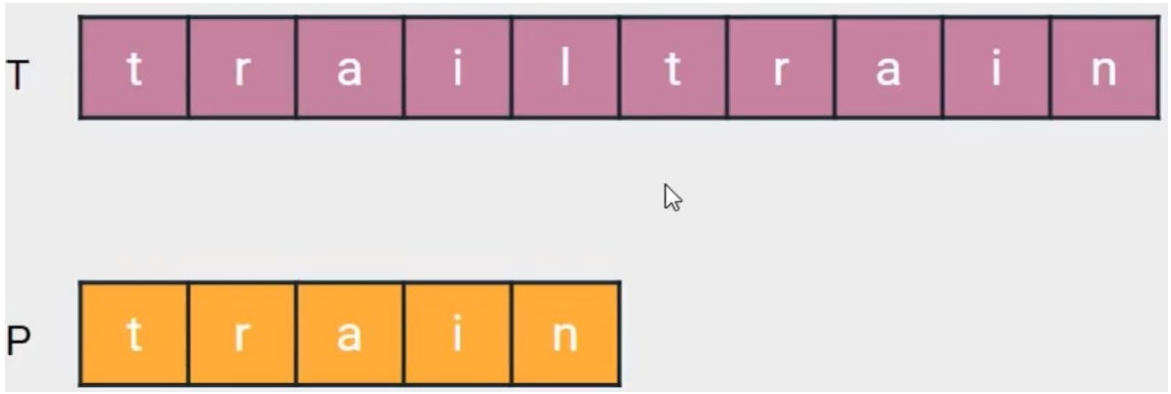


- In the worst-case scenario, we start with the first index of the text and perform m comparisons, then move to the second index of the text and do m comparisons, and so on, until we reach the index n-m (since the pattern will be truncated beyond this point).
- Thus, there are **n-m+1** possible starting positions.
- For each of these n-m+1 characters, we need to perform m comparisons. Consequently, the worst-case complexity of this analysis is significantly larger compared to m, the length of the pattern. It can be considered as O(nm).

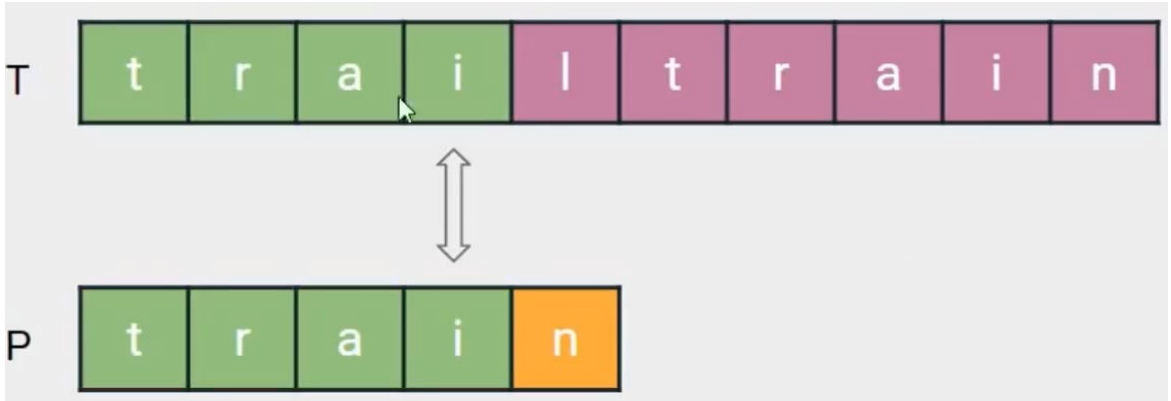
How to improve the brute force technique.

String Matching Algorithms:

How to improve the brute force technique.



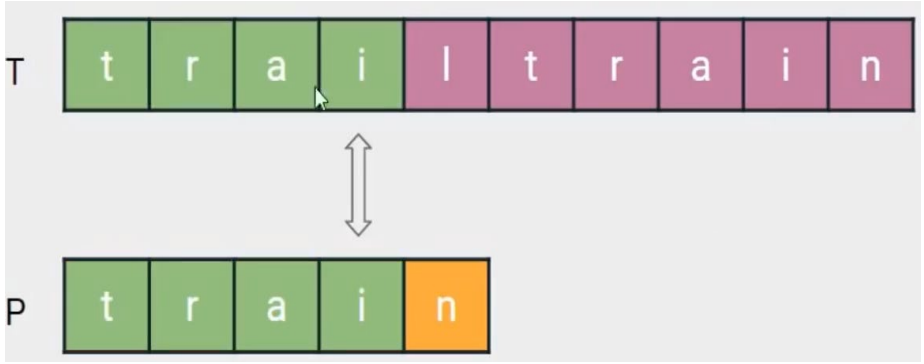
In the given example, where T=trailtrain and P=train, we observe that the first four characters match, but the last character is a mismatch.



With the brute force approach, we typically go backwards and start comparing from the second position. However, this backward step is what significantly slows down the brute force approach.

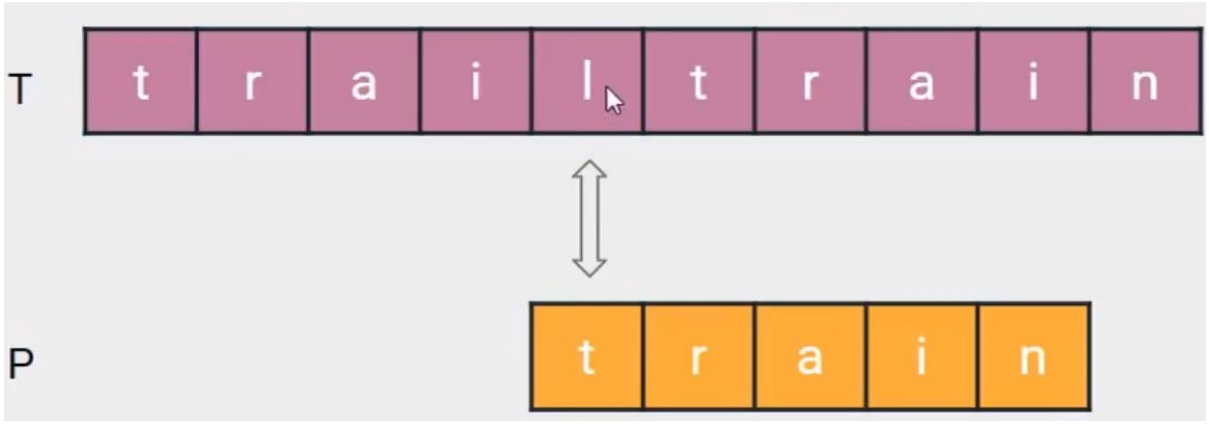
String Matching Algorithms:

How to improve the brute force technique.



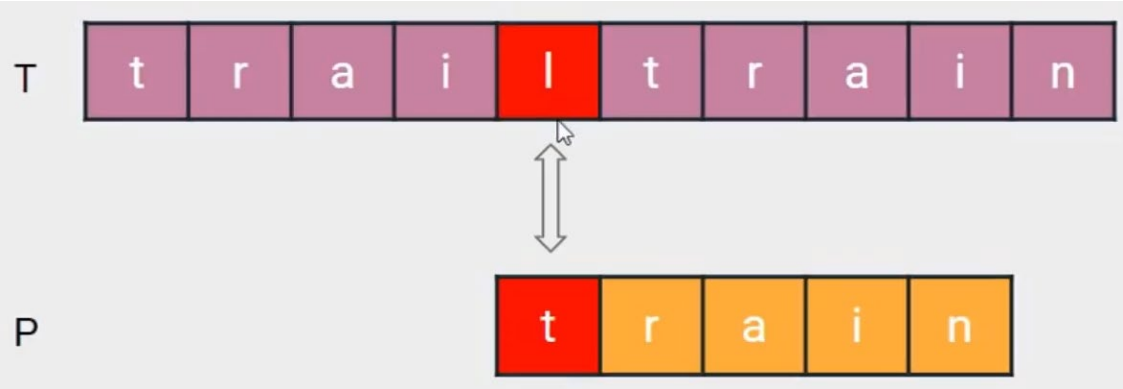
To avoid this backward step, we can utilize the fact that when there is a mismatch, such as between "**n**" and "**l**" in the pattern and text respectively, the previous characters must match.

In this example, the previous four characters of the pattern match with the previous four characters of the text. Therefore, instead of going all the way back and trying to compare again, we can simply start comparing the pattern from the position where the mismatch was found, ignoring all the previous characters of the text.

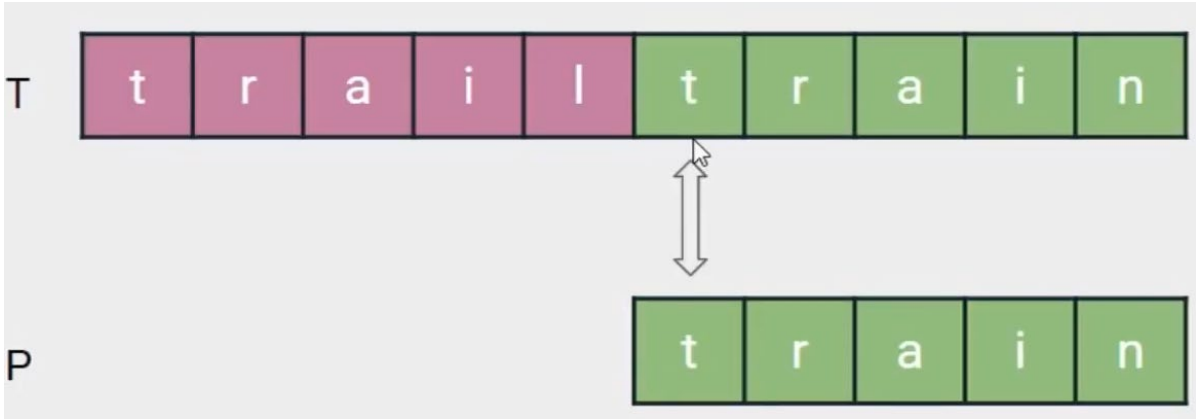


String Matching Algorithms:

How to improve the brute force technique.



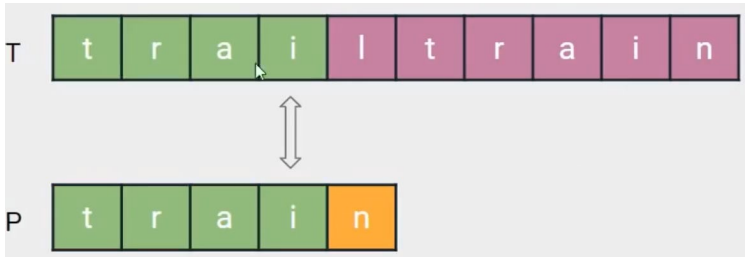
By adopting this approach, once a mismatch is encountered, we can immediately move forward to the next character in the text for comparison. This optimization allows us to efficiently search for matches without the need for unnecessary backward comparisons. In the given example, when T's "l" character and the pattern's "t" character mismatch, we can simply proceed forward to the next character, eventually matching the entire string.

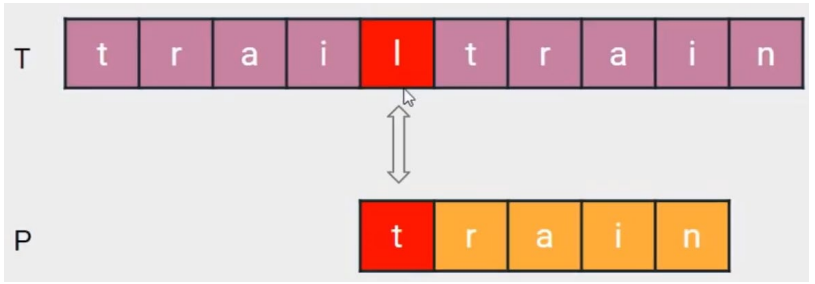


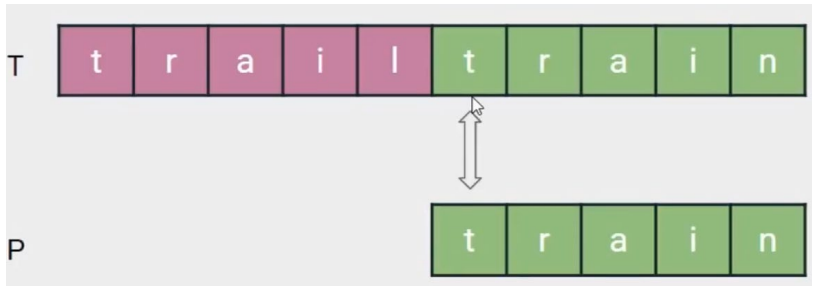
Is this approach correct, and will it work for every string?

String Matching Algorithms:

How to improve the brute force technique.







Is this approach correct, and will it work for every string?

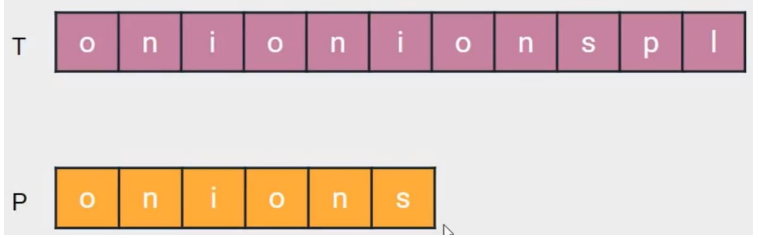
The approach of ignoring all previously matched characters and starting from the position of the mismatch will not work in all cases.

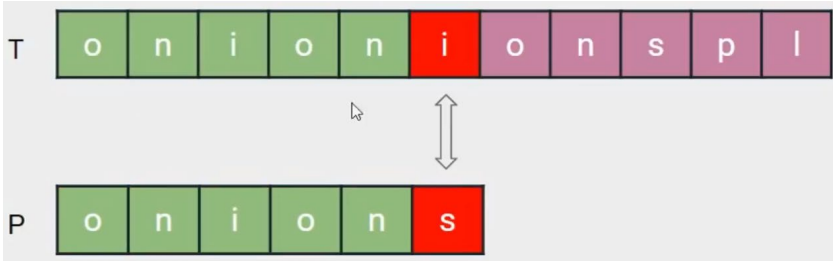
As you demonstrated in the example with T=onionionspl and P=onions, this approach can lead to missing the pattern when it occurs with overlapping characters.

Note: Here it worked because, characters in the pattern are distinct.

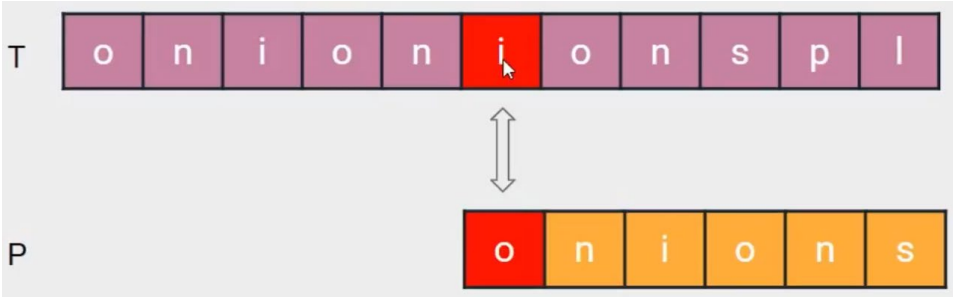
Let's consider another example:

- T=onionionspl
- P=onions

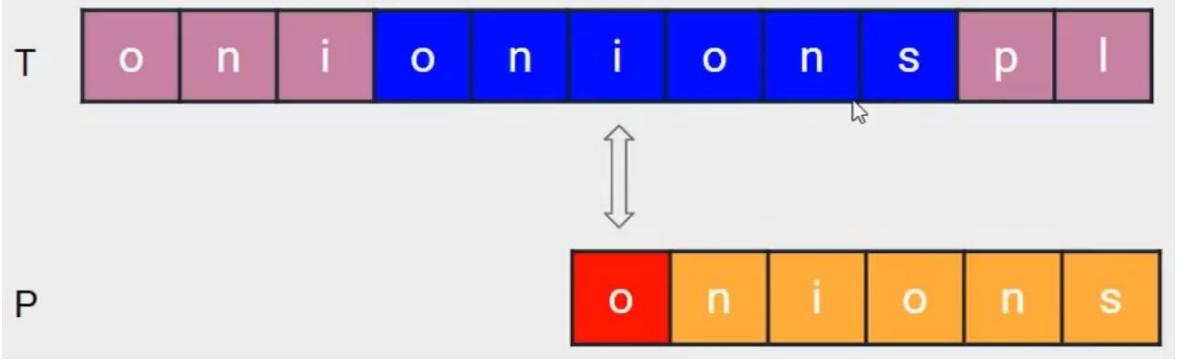




Here, the first 5 characters match, but there is a mismatch at the 6th character, i.e., 'i' in T and 's' in P.



If we follow our previous technique, which is to ignore all previously matched characters and start comparing from the mismatched position (i.e., at 'i' in T), then we will not be able to find the required pattern, which is actually present from the previously ignored position of text T, starting at index 4.



By totally ignoring the previously matched characters, we will be missing the overlapping characters like 'o' and 'n' in this case.

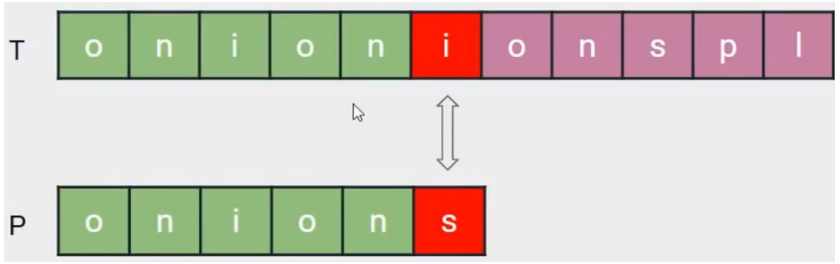
Knuth-Morris-Pratt (KMP): Introduction

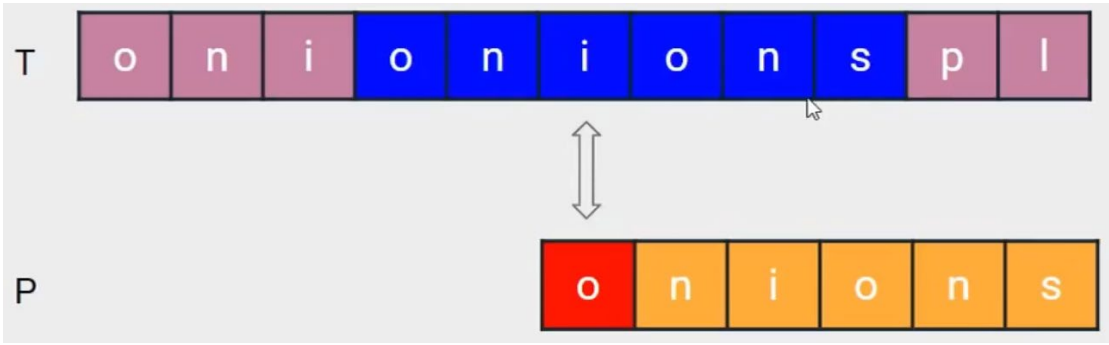
- From the previous explanation, a more robust approach is needed to handle such cases.
- Algorithms like the Knuth-Morris-Pratt (KMP) algorithm or the Boyer-Moore algorithm are designed to efficiently handle pattern matching with overlapping occurrences and distinct characters in the pattern.
- These algorithms use sophisticated techniques to avoid unnecessary comparisons while ensuring that all potential matches are correctly identified.

Knuth-Morris-Pratt (KMP): Introduction

Let's consider another example:

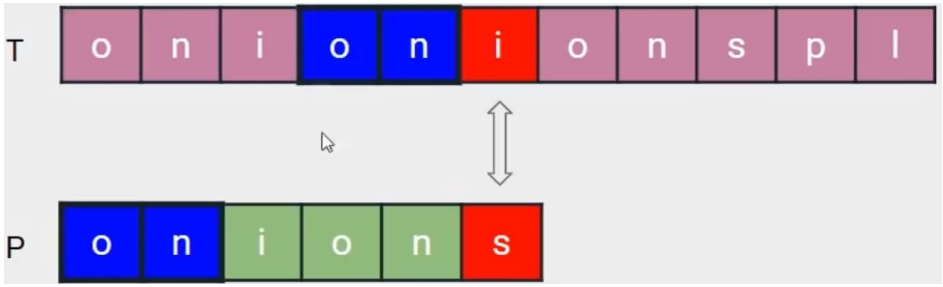
- T=onionionspl
- P=onions



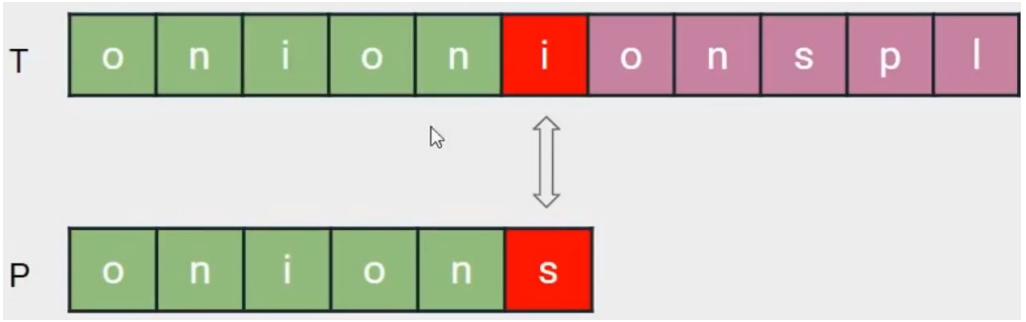


By totally ignoring the previously matched characters, we will be missing the overlapping characters like 'o' and 'n' in this case.

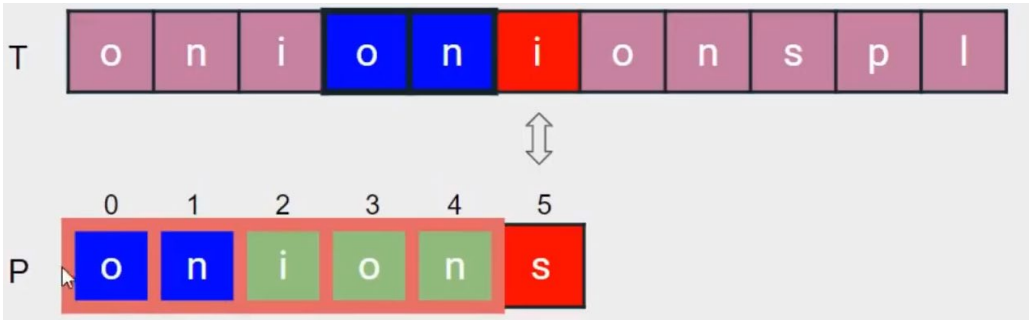
So, how exactly do we find these overlapping characters like 'o' and 'n' in this case?



Knuth-Morris-Pratt (KMP): Introduction

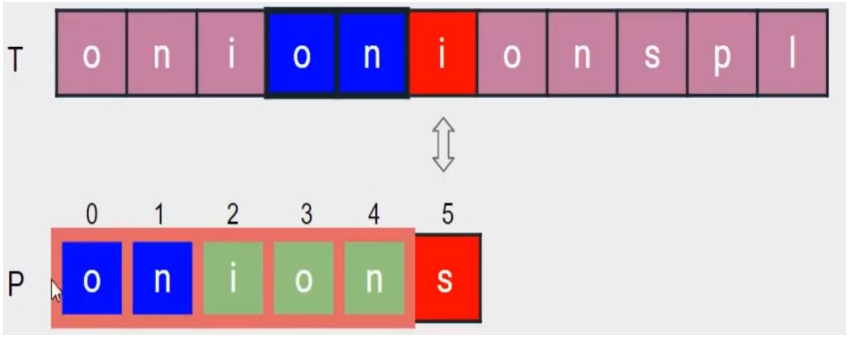


We know that the previous 5 characters of the text as well as the pattern match when there is a mismatch in the 6th character of the pattern. With this information, we can consider this particular substring of the first 5 characters.



The exact same 5 characters are present previously in the text. So in this substring, we have to find a prefix that is also a suffix. By doing this, we can match the prefix of the pattern with the suffix of the text and skip all the previous comparisons, starting from this third position or exactly the second index of the pattern. Moreover, it's not just any prefix that is also a suffix; it should be the longest prefix that is also a suffix, only then can we correctly match them and skip maximum comparisons.

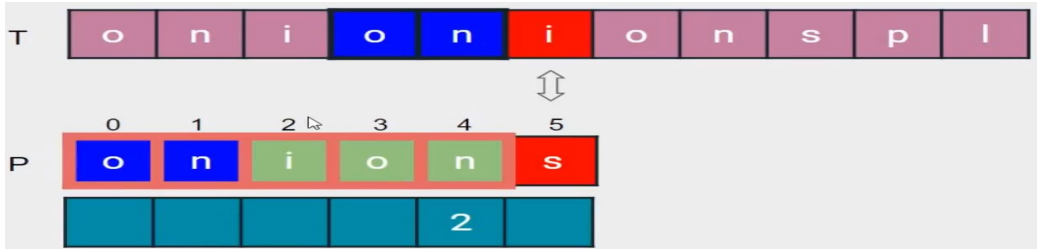
Knuth-Morris-Pratt (KMP): Introduction



The exact same 5 characters are present previously in the text. So in this substring, we have to find a prefix that is also a suffix. By doing this, we can match the prefix of the pattern with the suffix of the text and skip all the previous comparisons, starting from this third position or exactly the second index of the pattern. Moreover, it's not just any prefix that is also a suffix; it should be the longest prefix that is also a suffix, only then can we correctly match them and skip maximum comparisons.

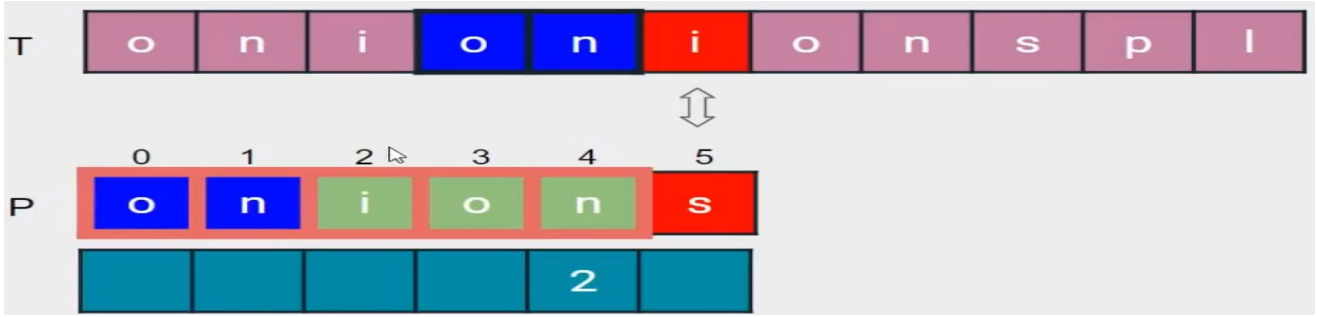
From the considered example, we have the mismatch in the 5th index of the pattern, which means the index 0 to 4 exactly matches with the previous characters of the text.

So now consider only this matched substring, i.e., index 0 to 4 of the pattern. In this substring, the longest prefix that is also a suffix is of length 2.

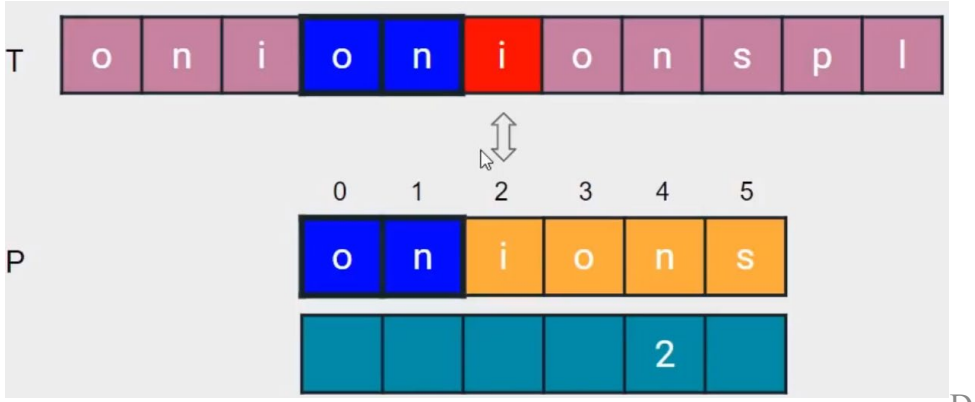


If we pre-calculate this information for each index of the pattern and store it in an array, that is, until that particular index, what is the length of the longest prefix that is also a suffix, with this information, we can easily look up that table and determine what is the next position that has to be compared whenever a mismatch occurs.

Knuth-Morris-Pratt (KMP): Introduction



In this example, the mismatch occurs in the fifth index, which means until the 4th index all the characters have matched. We go to the 4th index of the table, which gives the length of the longest prefix that is also a suffix until the fourth index. In this case, it is 2. With this information, we understand that the characters in the zeroth and the first index are already matching. With this information, if we start comparing from the second index and skip remaining comparisons without moving comparison pointers for the text, now we are able to determine from which position in the pattern we should begin our comparison.



This is the logic behind the KMP algorithm.

Knuth-Morris-Pratt (KMP): Introduction

- The Knuth-Morris-Pratt (KMP) algorithm was independently developed by Donald Knuth, Vaughan Pratt, and James H. Morris.
- It represents a significant advancement in string matching algorithms as it is the first linear-time algorithm for finding occurrences of a pattern within a text.

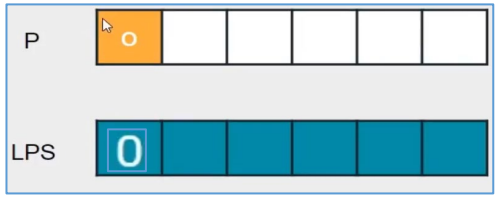
- One of the key **advantages** of the KMP algorithm over the brute force approach is its ability to avoid re-examining previously matched characters. This is achieved through the use of the failure function, also known as the "prefix function" or "pi function," which is precomputed based on the pattern.

- By efficiently skipping comparisons, the KMP algorithm reduces the time complexity to $O(n + m)$, where n is the length of the text and m is the length of the pattern, making it significantly faster than the brute force approach for large inputs.

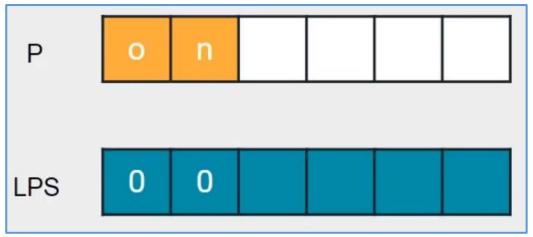
Knuth-Morris-Pratt (KMP): Algorithms

LPS arry: Longest prefix that is also a suffix. P=onions

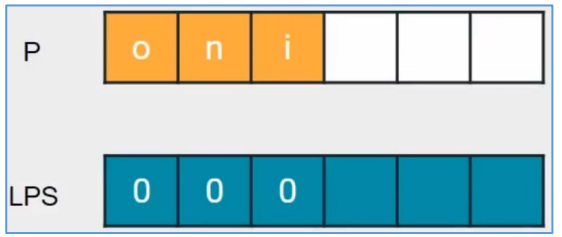
•For the zeroth index (i.e., 'o'), the length of the longest prefix that is also a suffix is zero because there are no other characters before it.



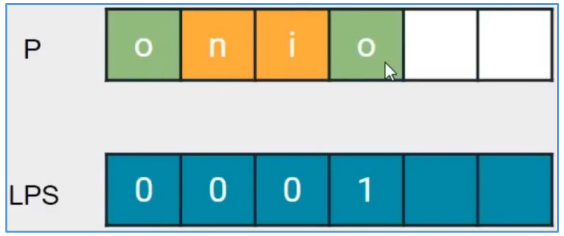
•For the first index (i.e., 'n'), we need to consider the substring "on". Here, 'o' is the prefix and 'n' is the suffix, so the length of the LPS is zero.



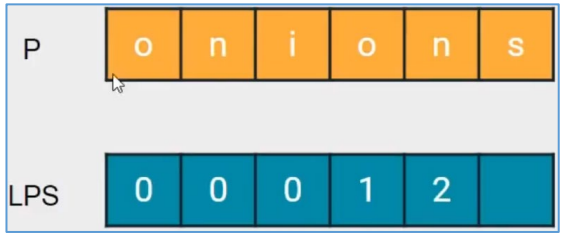
•Similarly, for the second index (i.e., 'i'), we consider the substring "oni". Again, 'o' is the prefix and 'i' is the suffix, so the length of the LPS is zero.



•For the third index (i.e., 'o'), we consider the substring "onio". Here, 'o' is both a prefix and a suffix, so the length of the LPS is one.

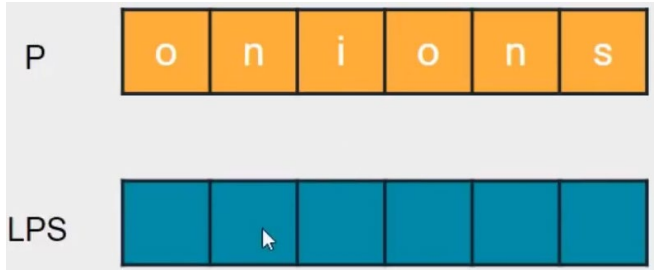


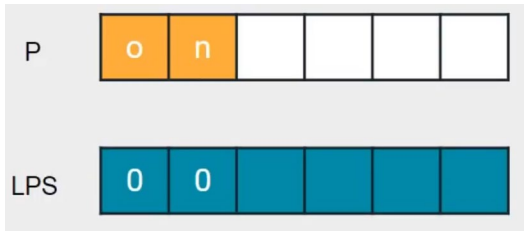
•Finally, for the fourth index (i.e., 'n'), we consider the substring "onion". In this case, "on" is a prefix and also a suffix, so the length of the LPS is two.

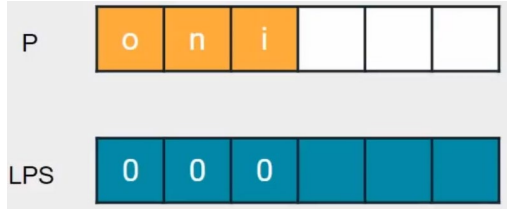


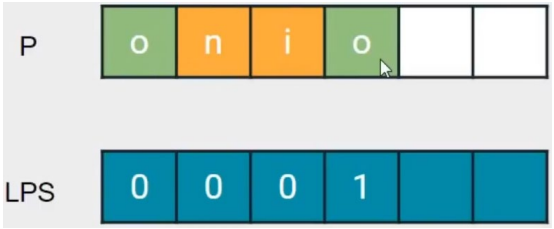
Knuth-Morris-Pratt (KMP): Algorithms

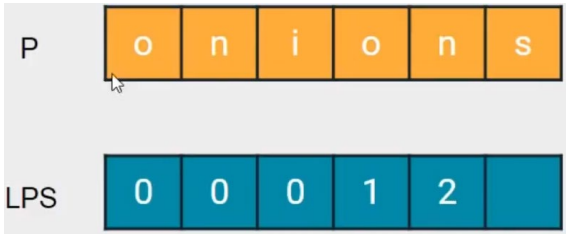
LPS arry: Longest prefix that is also a suffix.

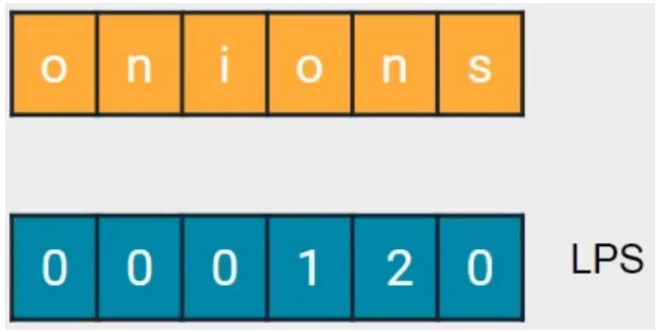






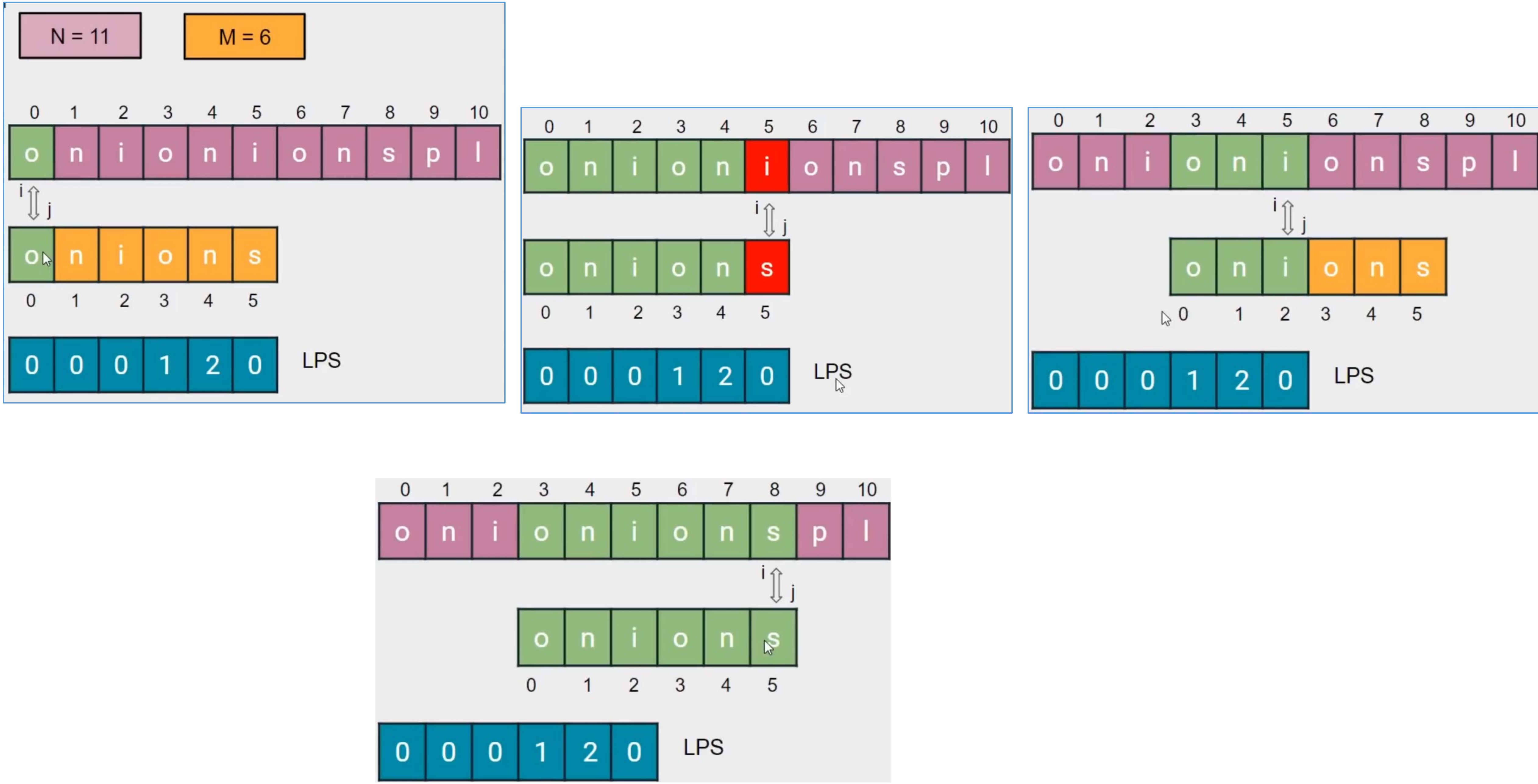






```
def KMPSearch(pat, txt):
    N = len(txt)
    M = len(pat)
    lps = [0]*M
    computeLPSArray(pat, M, lps)
    i=0
    j=0
    while i < N-M+1:
        if txt[i] == pat[j]:
            i += 1
            j += 1
        else:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
        if j == M:
            print(i-j)
            j = lps[j-1]
```

```
def computeLPSArray(pat,m,lps):
    len = 0
    i = 1
    lps[0] = 0
    while i < M:
        if pat[i] == pat[len]:
            lps[i] = len + 1
            len += 1
            i += 1
        else:
            if len != 0:
                len = lps[len-1]
            else:
                lps[i] = 0
                i += 1
```



N = 11
M = 6
0 1 2 3 4 5 6 7 8 9 10
o n i o n i o n s p l
i j
o n i o n s
0 1 2 3 4 5
0 0 0 1 2 0
LPS
0 1 2 3 4 5 6 7 8 9 10
o n i o n i o n s p l
i j
o n i o n s
0 1 2 3 4 5
0 0 0 1 2 0
LPS
0 1 2 3 4 5 6 7 8 9 10
o n i o n i o n s p l
i j
o n i o n s
0 1 2 3 4 5
0 0 0 1 2 0
LPS
0 1 2 3 4 5 6 7 8 9 10
o n i o n i o n s p l
i j
o n i o n s
0 1 2 3 4 5
0 0 0 1 2 0
LPS

Any Question?

Thank You!

FOR YOUR ATTENTION