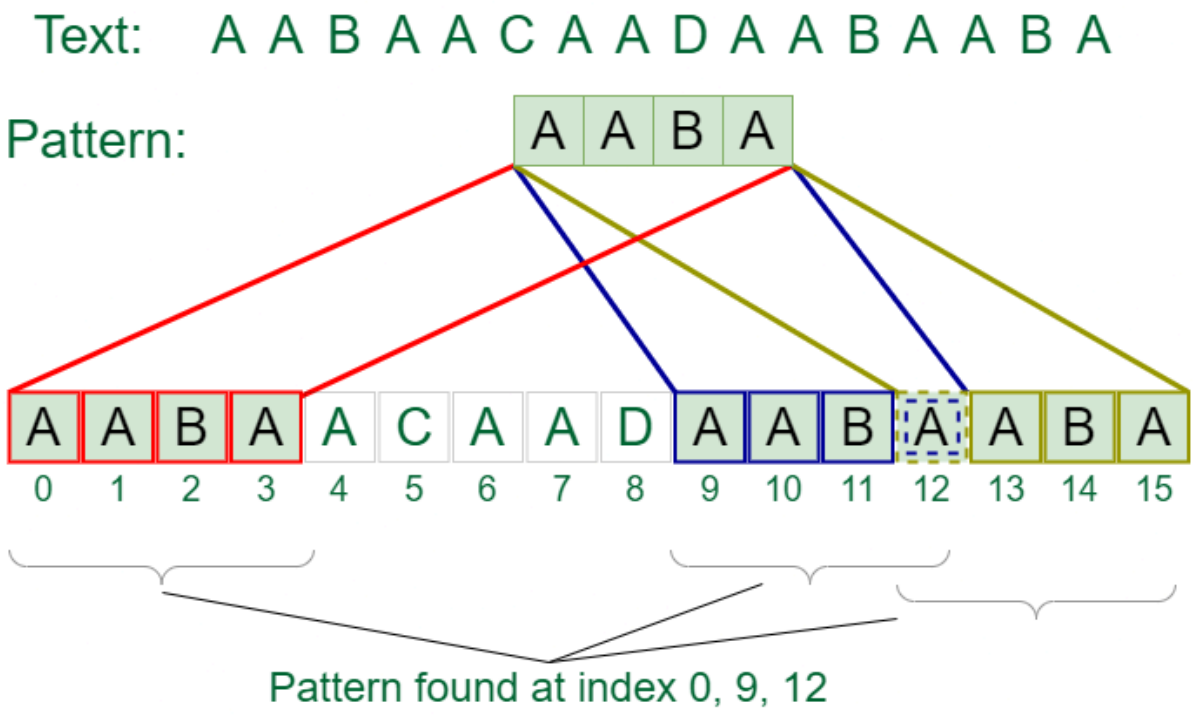
The Knuth-Morris-Pratt (KMP) algorithm is a string searching methodology which operates on the basis of previous character observations in a string may additionally be a part of the string being searched for, in the event of a mismatch. Utilizing this insight, the program harnessing this algorithm can potentially bypass both the re-examination of both previously discarded characters, and always ignore the previously found matching characters in the event of a partially-matching string.

The algorithm was conceived by James H. Harris and additionally 'discovered' a few weeks later by Donald Knuth. Morris and Vaughan Pratt published a technical report in 1970, finally publishing the algorithm in 1977 with all three members.

The initial goal behind the conception of the algorithm was to improve upon the Naive pattern searching algorithm, which simply iterates through the entire length of the string to be searched, looping through the word being searched for before proceeding. No segments of the string are skipped, which is where KMP's optimization comes in. The Naive algorithm already makes checks between indexes of both strings, so by additionally checking against the respective character of the desired string, when the compared character is not already the same as the compared character, we can know whether or not to denote that character as a possible partial-string. We then proceed to check the next character as a potential member of a partial-string, using that same index as a jump-to point for the main iteration of the searched string. The jumping behavior is provided by a calculated LPS (longest prefix which is a suffix) table, which is then referenced during operation.



A high-level view of the desired identifications to be made by a string-searching algorithm. (GfG)

KMP Algorithm Pseudo-code:

```
    input:
        string S (the text to be searched)
        string W (the word being identified)
    output:
        array of integers, P (positions in S at which W is found)
        integer, nP (number of positions)
    variables:
        integer, j = 0 (iterator for characters in S)
        integer, k = 0 (iterator for characters in W)
        array of integers, T (LPS table, calculated below)

    while j < length(S) do
        if W[k] = S[j] then
            let j += 1
            let k += 1
            if k = length(W) then
                let P[nP] = j - k, nP += 1
                let k = T[k]
        else
            let k = T[k]
            if k < 0 then
                let j += 1
                let k += 1
```

The above pseudocode for the KMP algorithm shows the iterative process of identifying desired words, with each loop starting with identifying the similarity of characters of mirrored indexes across the word being searched and the word to be identified.
In the event they are the same, both iterators are progressed, and the initial index of the word (current index - W length) is stored.
In the event they differ, the iterator for the searched-for string is set to the value defined in the LPS table, and if less than 0 (the start of a new similar substring) both iterators are increased by one.
This continues throughout the breadth of the string being scoured.

KMP LPS Table Pseudo-code:

```
    input:
        array of characters, W (the word to be analyzed)
    output:
        array of integers, T (the table to be filled)
    define variables:
        integer, pos = 1 (the current index calculated in T)
        integer, cnd = 0 (the zero-based index in W of the next
character of the current candidate substring)

    let T[0] = -1

    while pos < length(W) do
        if W[pos] = W[cnd] then
            let T[pos] = T[cnd]
        else
            let T[pos] = cnd
            while cnd ≥ 0 and W[pos] ≠ W[cnd] do
                let cnd = T[cnd]
        let pos += 1, cnd += 1
    let T[pos] = cnd
```

The above pseudo-code identifies repeating substrings throughout the word W. The effective cycle has a purely increasing counter 'pos', and counter 'cnd' which cycles between -1 and 0 while substrings have no pattern, and increasing while following a previously found substring. Upon no longer following identified substring patterns, the length of the current substring pattern (cnd) is placed in the table, and the next slot is marked as -1.
This pattern can be observed when comparing the below LPS table with the string S used in the step-by-step example.

This is the LPS table used in the step-by-step example, created by following the above code.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
-1	0	0	0	-1	0	0	3	-1	0	2	-1	0	0	3	-1	0	0	3	-1

Cont…

20	21	22
0	2	-1

Step-by-Step Example:

String `[ABC ABCDAB ABCDABCDABDE]` is being searched through for the substring `[ABCDABD]`.

`m` – The indexes of S; the string being searched.
`S` – The string being searched.
`W` – The word we are looking for matches of.
`i` – The indexes of W; the string being compared against.

```
             1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W: ABCDABD
i: 0123456
```

The characters of `S` and `W` are compared in parallel, as `S[m+i], W[i]` using `i` as an iterator. The initial three indexes match, however the fourth index `S[3]=' '` does not match `W[3]='D'`. Having previously checked `S[1],S[2]` we know neither = 'A' and set `m = 3, i = 0`.

```
             1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W:    ABCDABD
i:    0123456
```

Failing immediately, the algorithm proceeds to set `m = 4, i = 0`.

```
             1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456
```

The algorithm increments through six indexes of `W` stopping again at a mismatch between `' '` and `'D'`. However, before stopping at the tenth index, the substring `'AB'` is found, indicating that there is still a potential string match.

```
             1         2
m: 012345678901234567890123
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

Continuing, the algorithm once again stops at the tenth index, having failed the comparison between `S[10]= ' '` and `W[2]= 'C'`.
The algorithm once again compares the tenth index of `S`, now against `W[0]`.

```
             1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:            ABCDABD
i:            0123456
```

The algorithm immediately fails the check between `S[10]` and `W[0]`. `m` is incremented by one.

```
             1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:             ABCDABD
i:             0123456
```

Progressing through indexes zero through five of `W`, the algorithm then fails on the final sixth index before identifying a complete word. However, the algorithm notices that indexes fifteen and sixteen of `S` are `AB`, so `S` is set to 17, and `i` is set to 2.

```
             1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:                ABCDABD
i:                0123456
```

Finally, the algorithm fully identifies a word at index fifteen. (21, `S` Index minus 6, `W` length.)
The twenty-second index of `S` is checked, with nothing found.

The KMP algorithm has two complexities, O(k) and O(n), for the identification portion and table calculation portion. This leads to an overall complexity of O(k+n), or just O(n).
Compared against the algorithm which inspired KMP, Naive has a worst-case complexity of O(m(n-m+1)), or O(n*m). This demonstrates KMP's enhanced utility in string-identification applications.
Its use-cases are in checking for plagiarism, DNA sequencing, spell checking, spam filters, and search engine use in large databases.