KMP Pattern Searching

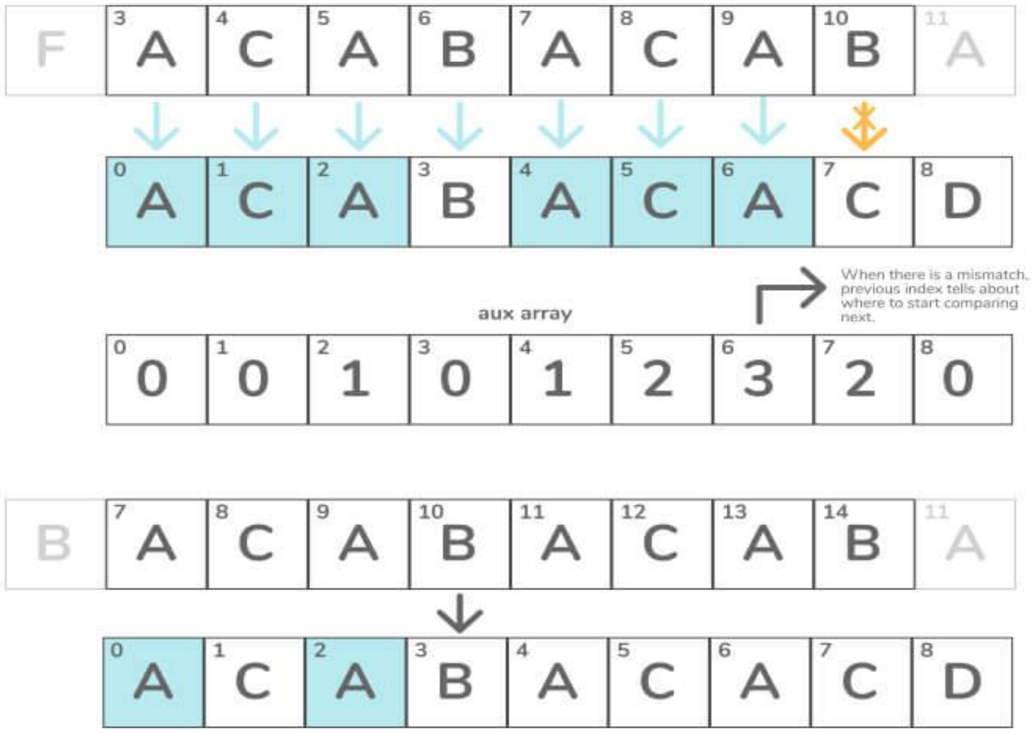
The KMP calculation utilizes the deteriorating property (design having the same sub-designs showing up more than once in the example) of the example and further develops the most pessimistic scenario running time complexity to O(n). The thought for the KMP calculation is: at whatever point the string gets mismatched, we definitely know a portion of the characters in the text of the following window. We will exploit this data to try not to coordinate with the characters that we realize will coordinate.

- The KMP algorithm pre-computes pat[] and creates an array lps[] of size m (same as the size of pattern) which is used to jump characters while matching.
- We search for lps in sub-patterns. More commonly we focus on sub-strings of patterns that are either prefixes and suffixes.
- For every sub-pattern pat[0..i] where i range from 0 to m-1, lps[i] stores the size of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

How can we utilize lps[] to decide the next positions or to know the number of characters to be jumped?

- We begin contrasting pat[j] with j = 0 with characters of the current window of text.
- We keep checking characters str[i] and pat[j] and keep incrementing i and j while pat[j] and str[i] keep matching.
- When we see there is a mismatch then,
    - It is already known that characters pat[0..j-1] are the same as str[i-j…i-1]
    - From the above points, we can conclude that lps[j-1] is the frequency of characters of pat[0…j-1] that are both proper prefixes and proper suffixes.
    - In conclusion, we don’t need to check these lps[j-1] characters with str[i-j…i-1] because we know that these characters will always match.

Dry Run Of Above Approach



InterviewBit

C++ Implementation of KMP

```
void KMPStringSearch(char* pat, char* str) {
    int M = strlen(pat)
    int N = strlen(str);

    int lps[M];
    computeLPSArray(pat, M, lps);

    int i = 0;
    int j = 0;
    while (i < N) {
        if (pat[j] == str[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }
        else if (i < N && pat[j] != str[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

void computeLPSArray(char* pat, int M, int* lps) {
        int len = 0;

    lps[0] = 0;
       int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else{
           if (len != 0) {
               len = lps[len - 1];
           }
            else                {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```