KMP Algorithm for Pattern Searching - GeeksforGeeks

Source: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund KMP Algorithm for Pattern Searching Last Updated : 10 Oct, 2025 The Knuth-Morris-Pratt (KMP) algorithm is an efficient string matching algorithm used to search for a pattern within a text. It uses a preprocessing step to handle mismatches smartly and achieves linear time complexity. KMP was developed by Donald Knuth, Vaughan Pratt, and James Morris in 1977. It is widely used in search engines, compilers, and text editors. Table of Content Naive Approach and How KMP Overcomes It LPS (Longest Prefix Suffix) Array Algorithm for Construction of LPS Array KMP Pattern Matching Algorithm Real-Life Applications Related Problems Naive Approach and How KMP Overcomes It In the naive string matching algorithm, we align the pattern at every position in the text and compare characters one by one. If a mismatch occurs, we shift the pattern by one position and start over. This can lead to rechecking the same characters multiple times, especially in cases with repeated characters. For example, searching "aaaab" in "aaaaaaaab" causes many unnecessary comparisons which lead to a time complexity of O(n × m). The KMP algorithm avoids this inefficiency by preprocessing the pattern using an auxiliary array called LPS (Longest Prefix Suffix). This array stores the length of the longest proper prefix which is also a suffix for every prefix of the pattern. When a mismatch occurs, KMP uses this information to shift the pattern intelligently, skipping over positions that are guaranteed not to match — instead of starting over. This ensures that each character in the text is compared at most once, reducing the time complexity to O(n + m). Proper Prefix : A proper prefix of a string is a prefix that is not equal to the string itself. For example, proper prefixes of "abcd" are: "", "a", "ab", and "abc". LPS (Longest Prefix Suffix) Array The LPS array stores, for every position in the pattern, the length of the longest proper prefix which is also a suffix of the substring ending at that position. It helps the KMP algorithm determine how much to shift the pattern when a mismatch occurs, without rechecking matched characters. Example of lps[] construction: Example 1: Pattern "aabaaac" At index 0: "a" → No proper prefix/suffix → lps[0] = 0 At index 1: "aa" → "a" is both prefix and suffix → lps[1] = 1 At index 2: "aab" → No prefix matches suffix → lps[2] = 0 At index 3: "aaba" → "a" is prefix and suffix → lps[3] = 1 At index 4: "aabaa" → "aa" is prefix and suffix → lps[4] = 2 At index 5: "aabaaa" → "aa" is prefix and suffix → lps[5] = 2 At index 6: "aabaaac" → Mismatch, so reset → lps[6] = 0 Final lps[]: [0, 1, 0, 1, 2, 2, 0] Example 2: Pattern "abcdabca" At index 0: lps[0] = 0 At index 1: lps[1] = 0 At index 2:lps[2] = 0 At index 3: lps[3] = 0 (no repetition in "abcd") At index 4: lps[4] = 1 ("a" repeats) At index 5: lps[5] = 2 ("ab" repeats) At index 6: lps[6] = 3 ("abc" repeats) At index 7: lps[7] = 1 (mismatch, fall back to "a") Final LPS: [0, 0, 0, 0, 1, 2, 3, 1] Note: lps[i] could also be defined as the longest prefix which is also a proper suffix. We need to use it properly in one place to make sure that the whole substring is not considered. Algorithm for Construction of LPS Array The value of lps[0] is always 0 because a string of length one has no non-empty proper prefix that is also a suffix. We maintain a variable len, initialized to 0, which keeps track of the length of the previous longest prefix suffix. As we traverse the pattern from index 1 onward, we compare the current character pat[i] with pat[len]. Based on this comparison, we have three possible cases: Case 1: pat[i] == pat[len] This means the current character continues the existing prefix-suffix match. → We increment len by 1 and assign lps[i] = len. → Then, move to the next index. Case 2: pat[i] != pat[len] and len == 0 There is no prefix that matches any suffix ending at i, and we can't fall back to any earlier matching pattern. → So we set lps[i] = 0 and simply move to the next

character. Case 3: pat[i] != pat[len] and len > 0 We cannot extend the previous matching prefix-suffix. However, there might still be a shorter prefix which is also a suffix that matches the current position. Instead of comparing all prefixes manually, we reuse previously computed LPS values. → Since pat[0...len-1] equals pat[i-len...i-1], we can fall back to lps[len - 1] and update len. → This reduces the prefix size we're matching against and avoids redundant work. We do not increment i immediately in this case — instead, we retry the current pat[i] with the new updated len. Illustration: Example of Construction of LPS Array: Try it on GfG Practice C++ vector < int > computeLPSArray (string & pattern) { int n = pattern . size (); vector < int > lps (n , 0); // length of the previous longest prefix suffix int len = 0 ; int i = 1 ; while (i < n) { if (pattern [i] == pattern [len]) { len ++ ; lps [i] = len ; i ++ ; } else { if (len != 0) { // fall back in the pattern len = lps [len - 1]; } else { lps [i] = 0 ; i ++ ; } } } return lps ; } Java class GfG { public static ArrayList < Integer > computeLPSArray (String pattern) { int n = pattern . length (); ArrayList < Integer > lps = new ArrayList <> (); for (int k = 0 ; k < n ; k ++) lps . add (0); // length of the previous longest prefix suffix int len = 0 ; int i = 1 ; while (i < n) { if (pattern . charAt (i) == pattern . charAt (len)) { len ++ ; lps . set (i , len); i ++ ; } else { if (len != 0) { // fall back in the pattern len = lps . get (len - 1); } else { lps . set (i , 0); i ++ ; } } } return lps ; } } Python def computeLPSArray (pattern): n = len (pattern) lps = [0] * n # length of the previous longest prefix suffix len = 0 i = 1 while i < n : if pattern [i] == pattern [len]: len += 1 lps [i] = len i += 1 else : if len != 0 : # fall back in the pattern len = lps [len - 1] else : lps [i] = 0 i += 1 return lps if __name__ == "__main__" : pattern = "ababcab" print (computeLPSArray (pattern)) C# class GfG { public static List < int > computeLPSArray (string pattern) { int n = pattern . Length ; List < int > lps = new List < int > (new int [n]); // length of the previous longest prefix suffix int len = 0 ; int i = 1 ; while (i < n) { if (pattern [i] == pattern [len]) { len ++ ; lps [i] = len ; i ++ ; } else { if (len != 0) { // fall back in the pattern len = lps [len - 1]; } else { lps [i] = 0 ; i ++ ; } } } return lps ; } } JavaScript function computeLPSArray (pattern) { let n = pattern . length ; let lps = new Array (n). fill (0); // length of the previous longest prefix suffix let len = 0 ; let i = 1 ; while (i < n) { if (pattern [i] === pattern [len]) { len ++ ; lps [i] = len ; i ++ ; } else { if (len !== 0) { // fall back in the pattern len = lps [len - 1]; } else { lps [i] = 0 ; i ++ ; } } } return lps ; } // Driver Code let pattern = "ababcab" ; console . log (computeLPSArray (pattern)); Time Complexity: O(n), each character in the pattern is processed at most twice — once when moving forward (i++) and possibly again when falling back using the len = lps[len - 1] step. Auxiliary Space: O(n), an extra array lps[] of size equal to the pattern is used. KMP Pattern Matching Algorithm Terminologies used in KMP Algorithm: text (txt): The main string in which we want to search for a pattern. pattern (pat): The substring we are trying to find within the text. Match: A match occurs when all characters of the pattern align exactly with a substring of the text. LPS Array (Longest Prefix Suffix): For each position i in the pattern, lps[i] stores the length of the longest proper prefix which is also a suffix in the substring pat[0...i]. Proper Prefix: A proper prefix is a prefix that is not equal to the whole string. Suffix: A suffix is a substring that ends at the current position. The LPS array helps us determine how much we can skip in the pattern when a mismatch occurs, thus avoiding redundant comparisons. Problem Statement: Given two strings: txt , representing the main text, and pat , representing the pattern to be searched. Find and return all starting indices in txt where the string pat appears as a substring. The matching should be exact, and the indices should be 0-based, meaning the first character of txt is considered to be at index 0. Examples: Input: txt = "abcab", pat = "ab" Output: [0, 3] Explanation : The string "ab" occurs twice in txt, first occurrence starts from index 0 and second from index 3. Input: txt = "aabaacaadaabaaba", pat = "aaba" Output: [0, 9, 12] Explanation: The KMP algorithm works in two main steps: 1. Preprocessing Step – Build the LPS Array First, we process the pattern to create an array called LPS (Longest Prefix Suffix). This array tells us: "If a mismatch happens at this point, how far back in the pattern can we jump without missing any potential matches?" It helps us avoid starting from the beginning of the pattern again after a mismatch. This step is done only once, before we start searching in the text. 2. Matching Step – Search the Pattern in the Text Now, we start comparing the pattern with the text, one character at a time. If the characters match: Move forward in both the text and the pattern. If the characters don’t match: => If we're not at the start of the pattern, we use the LPS value at the previous index (i.e., lps[j - 1]) to move the pattern pointer j back to that position. This means: jump to the longest prefix that is also a suffix — no need to recheck those characters. => If we're at the start (i.e., j == 0), simply move the text pointer i forward to try the next character. If we reach the end of the pattern (i.e., all characters matched), we found a match! Record the starting index and continue searching. Illustration: C++ #include <iostream> #include <string> #include <vector> using namespace std ; void constructLps (string & pat , vector < int > & lps) { // len stores the length of longest prefix which // is also a suffix for the previous index int len = 0 ; // lps[0] is always 0 lps [0] = 0 ;

int i = 1 ; while (i < pat . length ()) { // If characters match, increment the size of lps if (pat [i] == pat [len]) { len ++ ; lps [i] = len ; i ++ ; } // If there is a mismatch else { if (len != 0) { // Update len to the previous lps value // to avoid reduntant comparisons len = lps [len - 1]; } else { // If no matching prefix found, set lps[i] to 0 lps [i] = 0 ; i ++ ; } } } } vector < int > search (string & pat , string & txt) { int n = txt . length (); int m = pat . length (); vector < int > lps (m); vector < int > res ; constructLps (pat , lps); // Pointers i and j, for traversing // the text and pattern int i = 0 ; int j = 0 ; while (i < n) { // If characters match, move both pointers forward if (txt [i] == pat [j]) { i ++ ; j ++ ; // If the entire pattern is matched // store the start index in result if (j == m) { res . push_back (i - j); // Use LPS of previous index to // skip unnecessary comparisons j = lps [j - 1]; } } // If there is a mismatch else { // Use lps value of previous index // to avoid redundant comparisons if (j != 0) j = lps [j - 1]; else i ++ ; } } return res ; } int main () { string txt = "aabaacaadaabaaba" ; string pat = "aaba" ; vector < int > res = search (pat , txt); for (int i = 0 ; i < res . size (); i ++) cout << res [i] << " " ; return 0 ; } Java import java.util.ArrayList ; class GfG { static void constructLps (String pat , int [] lps) { // len stores the length of longest prefix which // is also a suffix for the previous index int len = 0 ; // lps[0] is always 0 lps [0] = 0 ; int i = 1 ; while (i < pat . length ()) { // If characters match, increment the size of lps if (pat . charAt (i) == pat . charAt (len)) { len ++ ; lps [i] = len ; i ++ ; } // If there is a mismatch else { if (len != 0) { // Update len to the previous lps value // to avoid redundant comparisons len = lps [len - 1] ; } else { // If no matching prefix found, set lps[i] to 0 lps [i] = 0 ; i ++ ; } } } } static ArrayList < Integer > search (String pat , String txt) { int n = txt . length (); int m = pat . length (); int [] lps = new int [m] ; ArrayList < Integer > res = new ArrayList <> (); constructLps (pat , lps); // Pointers i and j, for traversing // the text and pattern int i = 0 ; int j = 0 ; while (i < n) { // If characters match, move both pointers forward if (txt . charAt (i) == pat . charAt (j)) { i ++ ; j ++ ; // If the entire pattern is matched // store the start index in result if (j == m) { res . add (i - j); // Use LPS of previous index to // skip unnecessary comparisons j = lps [j - 1] ; } } // If there is a mismatch else { // Use lps value of previous index // to avoid redundant comparisons if (j != 0) j = lps [j - 1] ; else i ++ ; } } return res ; } public static void main (String [] args) { String txt = "aabaacaadaabaaba" ; String pat = "aaba" ; ArrayList < Integer > res = search (pat , txt); for (int i = 0 ; i < res . size (); i ++) System . out . print (res . get (i) + " "); } } Python def constructLps (pat , lps): # len stores the length of longest prefix which # is also a suffix for the previous index len_ = 0 m = len (pat) # lps[0] is always 0 lps [0] = 0 i = 1 while i < m : # If characters match, increment the size of lps if pat [i] == pat [len_]: len_ += 1 lps [i] = len_ i += 1 # If there is a mismatch else : if len_ != 0 : # Update len to the previous lps value # to avoid redundant comparisons len_ = lps [len_ - 1] else : # If no matching prefix found, set lps[i] to 0 lps [i] = 0 i += 1 def search (pat , txt): n = len (txt) m = len (pat) lps = [0] * m res = [] constructLps (pat , lps) # Pointers i and j, for traversing # the text and pattern i = 0 j = 0 while i < n : # If characters match, move both pointers forward if txt [i] == pat [j]: i += 1 j += 1 # If the entire pattern is matched # store the start index in result if j == m : res . append (i - j) # Use LPS of previous index to # skip unnecessary comparisons j = lps [j - 1] # If there is a mismatch else : # Use lps value of previous index # to avoid redundant comparisons if j != 0 : j = lps [j - 1] else : i += 1 return res if __name__ == "__main__" : txt = "aabaacaadaabaaba" pat = "aaba" res = search (pat , txt) for i in range (len (res)): print (res [i], end = " ") C# using System ; using System.Collections.Generic ; class GfG { static void constructLps (string pat , int [] lps) { // len stores the length of longest prefix which // is also a suffix for the previous index int len = 0 ; // lps[0] is always 0 lps [0] = 0 ; int i = 1 ; while (i < pat . Length) { // If characters match, increment the size of lps if (pat [i] == pat [len]) { len ++ ; lps [i] = len ; i ++ ; } // If there is a mismatch else { if (len != 0) { // Update len to the previous lps value // to avoid redundant comparisons len = lps [len - 1]; } else { // If no matching prefix found, set lps[i] to 0 lps [i] = 0 ; i ++ ; } } } } static List < int > search (string pat , string txt) { int n = txt . Length ; int m = pat . Length ; int [] lps = new int [m]; List < int > res = new List < int > (); constructLps (pat , lps); // Pointers i and j, for traversing // the text and pattern int i = 0 ; int j = 0 ; while (i < n) { // If characters match, move both pointers forward if (txt [i] == pat [j]) { i ++ ; j ++ ; // If the entire pattern is matched // store the start index in result if (j == m) { res . Add (i - j); // Use LPS of previous index to // skip unnecessary comparisons j = lps [j - 1]; } } // If there is a mismatch else { // Use lps value of previous index // to avoid redundant comparisons if (j != 0) j = lps [j - 1]; else i ++ ; } } return res ; } static void Main (string [] args) { string txt = "aabaacaadaabaaba" ; string pat = "aaba" ; List < int > res = search (pat , txt); for (int i = 0 ; i < res . Count ; i ++) Console . Write (res [i] + " "); } } JavaScript function constructLps (pat , lps) { // len stores the length of longest prefix which // is also a suffix for the previous index let len = 0 ; // lps[0] is always 0 lps [0] = 0 ; let i = 1 ; while (i < pat . length) { // If characters match, increment the size of lps if (pat [i] === pat [len]) { len ++ ; lps [i] = len ; i ++ ; } // If there is a mismatch else { if (len !== 0) { // Update len to the previous lps value // to avoid

redundant comparisons len = lps [len - 1]; } else { // If no matching prefix found, set lps[i] to 0 lps [i] = 0 ; i ++ ; } } } } function search (pat , txt) { const n = txt . length ; const m = pat . length ; const lps = new Array (m); const res = []; constructLps (pat , lps); // Pointers i and j, for traversing // the text and pattern let i = 0 ; let j = 0 ; while (i < n) { // If characters match, move both pointers forward if (txt [i] === pat [j]) { i ++ ; j ++ ; // If the entire pattern is matched // store the start index in result if (j === m) { res . push (i - j); // Use LPS of previous index to // skip unnecessary comparisons j = lps [j - 1]; } } // If there is a mismatch else { // Use lps value of previous index // to avoid redundant comparisons if (j !== 0) j = lps [j - 1]; else i ++ ; } } return res ; } // Driver Code const txt = "aabaacaadaabaaba" ; const pat = "aaba" ; const res = search (pat , txt); console . log (res . join (" ")); Output 0 9 12 Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern. This is because creating the LPS (Longest Prefix Suffix) array takes O(m) time, and the search through the text takes O(n) time. Auxiliary Space: O(m), as we need to store the LPS array of size m. Advantages of KMP Linear time complexity. No backtracking in the text. Efficient for large text searches (like log analysis, DNA sequencing). Real-Life Applications Text Editors (Find feature) Plagiarism Detection Bioinformatics (DNA sequence matching) Spam Detection Systems Search Engines Related Problems Case Insensitive Search Find All Occurrences of Subarray in Array Minimum Characters to Add at Front for Palindrome Check if Strings Are Rotations of Each Other Minimum Repetitions of s1 such that s2 is a substring of it Longest prefix which is also suffix Comment Article Tags: Article Tags: Strings Pattern Searching DSA Amazon Oracle Accolite Payu MAQ Software MakeMyTrip + 5 More