# Anagram Substring Search (Or Search for all permutations) - GeeksforGeeks

**Source:** https://www.geeksforgeeks.org/anagram-substring-search-search-permutations/

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 Anagram Substring Search (Or Search for all permutations) Last Updated : 23 Jul, 2025 Given a text txt and a pattern pat of size n and m respectively, the task is to find all occurrences of pat and its permutations (or anagrams) in txt . You may assume n > m. Examples: Input : txt = "BACDGABCDA", pat = "ABCD" Output : [0, 5, 6] Explanation : "BACD" is at 0, "ABCD" at 5 and "BCDA" at 6 Input : txt = "AAABABAA", pat = "AABA" Output : [0, 1, 4] Explanation: "AAAB" is at 0, "AABA" at 5 and "ABAA" at 6 Naive Solution - O((m log m) + (n-m+1)(m log m) ) Time & O(m) Space This problem is slightly different from the standard pattern-searching problem, here we need to search for anagrams as well. Therefore, we cannot directly apply standard pattern-searching algorithms like KMP , Rabin Karp , Boyer Moore , etc. The idea is to consider all the substrings of the txt with are of lengths equal to the length of pat and check whether the sorted version of substring is equal to the sorted version of pat . If they are equal then that particular substring is the permutation of the pat , else not. Illustration: Text (txt): "BACABC", Pattern (pat): "CBA" Sort pat to get sortedpat = "ABC". For each substring in txt of length equal to pat (length 3): Substring "BAC": Sort to get "ABC" (match found, add index 0). Substring "ACA": Sort to get "AAC" (no match). Substring "CAB": Sort to get "ABC" (match found, add index 2). Substring "ABC": Sort to get "ABC" (match found, add index 3). The indices of anagrams found are 0, 2, and 3. C++ #include <bits/stdc++.h> using namespace std ; vector < int > search ( string & pat , string & txt ) { int n = txt . length (), m = pat . length (); /* sortedpat stores the sorted version of pat */ string sortedpat = pat ; sort ( sortedpat . begin (), sortedpat . end ()); // to store the matching indices vector < int > res ; for ( int i = 0 ; i <= n - m ; i ++ ) { // renamed from temp to curr string curr = "" ; for ( int k = i ; k < m + i ; k ++ ) curr . push_back ( txt [ k ]); sort ( curr . begin (), curr . end ()); /* checking if sorted versions are equal */ if ( sortedpat == curr ) res . push_back ( i ); } return res ; } int main () { string txt = "BACDGABCDA" ; string pat = "ABCD" ; vector < int > result = search ( pat , txt ); for ( int idx : result ) cout << idx << " " ; return 0 ; } C #include <stdio.h> #include <string.h> #include <stdlib.h> // Function to sort characters in a string void sortString ( char * str , int len ) { for ( int i = 0 ; i < len - 1 ; i ++ ) { for ( int j = i + 1 ; j < len ; j ++ ) { if ( str [ i ] > str [ j ]) { char temp = str [ i ]; str [ i ] = str [ j ]; str [ j ] = temp ; } } } } // Function to search for anagrams of the pattern in text void search ( char * pat , char * txt , int * result , int * resCount ) { int n = strlen ( txt ); int m = strlen ( pat ); //sortedpat stores the sorted version of pat char sortedpat [ m + 1 ]; strcpy ( sortedpat , pat ); sortString ( sortedpat , m ); * resCount = 0 ; for ( int i = 0 ; i <= n - m ; i ++ ) { // renamed from temp to curr char curr [ m + 1 ]; strncpy ( curr , txt + i , m ); curr [ m ] = '\0' ; sortString ( curr , m ); //checking if sorted versions are equal if ( strcmp ( sortedpat , curr ) == 0 ) { result [ * resCount ] = i ; ( * resCount ) ++ ; } } } // Driver code int main () { char txt [] = "BACDGABCDA" ; char pat [] = "ABCD" ; int result [ 100 ], resCount ; search ( pat , txt , result , & resCount ); for ( int i = 0 ; i < resCount ; i ++ ) { printf ( "%d " , result [ i ]); } return 0 ; } Java import java.util.* ; class GfG { // Function to search for anagrams of the pattern in text static List < Integer > search ( String pat , String txt ) { int n = txt . length (), m = pat . length (); //sortedpat stores the sorted version of pat char [] sortedpatArr = pat . toCharArray (); Arrays . sort ( sortedpatArr ); String sortedpat = new String ( sortedpatArr ); // to store the matching indices List < Integer > res = new ArrayList <> (); for ( int i = 0 ; i <= n - m ; i ++ ) { // renamed from temp to curr String curr = txt . substring ( i , i + m ); char [] currArr = curr . toCharArray (); Arrays . sort ( currArr ); curr = new String ( currArr ); //checking if sorted versions are equal if ( sortedpat

. equals ( curr )) { res . add ( i ); } } return res ; } // Driver code public static void main ( String [] args ) { String txt = "BACDGABCDA" ; String pat = "ABCD" ; List < Integer > result = search ( pat , txt ); for ( int idx : result ) { System . out . print ( idx + " " ); } } } Python # Function to search for anagrams of the pattern in text def search ( pat , txt ): n = len ( txt ) m = len ( pat ) # sortedpat stores the sorted version of pat sortedpat = '' . join ( sorted ( pat )) # to store the matching indices res = [] for i in range ( n - m + 1 ): # renamed from temp to curr curr = txt [ i : i + m ] curr = '' . join ( sorted ( curr )) # checking if sorted versions are equal if sortedpat == curr : res . append ( i ) return res # Driver code txt = "BACDGABCDA" pat = "ABCD" result = search ( pat , txt ) for idx in result : print ( idx , end = " " ) C# using System ; using System.Collections.Generic ; class GfG { static List < int > Search ( string pat , string txt ) { int n = txt . Length , m = pat . Length ; /* sortedpat stores the sorted version of pat */ char [] sortedPatArr = pat . ToCharArray (); Array . Sort ( sortedPatArr ); string sortedPat = new string ( sortedPatArr ); // to store the matching indices List < int > res = new List < int > (); for ( int i = 0 ; i <= n - m ; i ++ ) { // renamed from temp to curr string curr = "" ; for ( int k = i ; k < m + i ; k ++ ) curr += txt [ k ]; char [] currArr = curr . ToCharArray (); Array . Sort ( currArr ); string sortedCurr = new string ( currArr ); /* checking if sorted versions are equal */ if ( sortedPat == sortedCurr ) res . Add ( i ); } return res ; } static void Main ( string [] args ) { string txt = "BACDGABCDA" ; string pat = "ABCD" ; List < int > result = Search ( pat , txt ); foreach ( int idx in result ) Console . Write ( idx + " " ); Console . ReadLine (); } } JavaScript // Function to search for anagrams of the pattern in text function search ( pat , txt ) { const n = txt . length , m = pat . length ; // sortedpat stores the sorted version of pat const sortedpat = pat . split ( '' ). sort (). join ( '' ); // to store the matching indices const res = []; for ( let i = 0 ; i <= n - m ; i ++ ) { // renamed from temp to curr let curr = txt . slice ( i , i + m ). split ( '' ). sort (). join ( '' ); // checking if sorted versions are equal if ( sortedpat === curr ) { res . push ( i ); } } return res ; } // Driver code const txt = "BACDGABCDA" ; const pat = "ABCD" ; const result = search ( pat , txt ); console . log ( result . join ( " " )); Output 0 5 6 Time Complexity: The for loop runs for n-m+1 times in each iteration we build a string of size m, which takes O(m) time, and sorting it takes O(m log m) time, and comparing sorted pat and sorted substring, which takes O(m) . So time complexity is O((n-m+1)*(m + m log m + m) ). Total Time is O(m log m) + O( (n-m+1)(m + mlogm + m) ) Auxiliary Space: O(m) As we are using extra space for curr string and sorted pat Hashing and Window Sliding - O (256 * (n - m) + m) Time & O(m + 256) Space Instead of checking each substring one by one like we were doing in the above naive approach, we can use two arrays to count how many times each character appears in the pattern and in the current part/window of the text we're looking at. By sliding this window across the text and updating our counts, we can quickly determine if the current substring matches the pattern. This method is much faster and allows us to handle larger texts more efficiently. For more detail about the implementation part, please refer to this article Comment Article Tags: Article Tags: Pattern Searching DSA Arrays Microsoft Amazon permutation anagram sliding-window + 4 More